JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Dipl.-Ing.
Manuel Rigger, M.Phil.

Submitted at
Institute for
System Software

Supervisor and
First Examiner
o.Univ.-Prof.
Dipl.-Ing. Dr. Dr.h.c.
Hanspeter Mössenböck

Second Examiner
Reader
Dr. Cristian Cadar

October 2018

Safe and Efficient Execution of LLVM-based Languages

Doctoral Thesis
to obtain the academic degree of
Doktor der technischen Wissenschaften
in the Doctoral Program
Technische Wissenschaften

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.
This printed thesis is identical with the electronic version submitted.

Linz, October 29, 2018

Abstract

In unsafe languages like C/C++, errors such as buffer overflows cause *Undefined Behavior*. Typically, compilers handle undefined behavior in some arbitrary way, for example, they disregard it when optimizing and omit inserting checks that could detect it. Consequently, undefined behavior often results in hard-to-find bugs and security vulnerabilities, for example, when a buffer overflow corrupts memory.

Existing bug-finding tools that instrument code to detect undefined behavior often suffer from compilers that possibly optimize code so that errors are no longer detected. Alternatively, unsafe code could be rewritten in a safe language like Java, which is well defined and where such errors are detected. However, this would incur an infeasible-high cost for many projects.

To tackle undefined behavior, we came up with an approach to execute unsafe languages on the Java Virtual Machine. We implemented this approach as *Safe Sulong*, a system that includes an interpreter for unsafe languages, which is written in Java. By relying on Java's well-definedness and its automatic run-time checks, the interpreter can detect buffer overflows and other errors during its execution and can terminate the program in such cases. Safe Sulong tracks metadata such as types and object bounds, which we provide to programmers over an introspection interface, so that they can use this data to mitigate errors and to implement additional checks. The interpreter also supports unstandardized elements in C code such as the most common inline assembly and GCC builtins. To implement them, we first studied their usage in a large number of open-source projects.

Sulong is used in GraalVM, a commercially-used multi-lingual virtual machine. Since Sulong allows the implementation of efficient native function interfaces, our safe execution mechanism could also make the execution of native extensions of other languages such as Ruby, Python, and R safer.

Kurzfassung

In unsicheren Sprachen wie C/C++ führen Fehler wie Pufferüberläufe zu *undefiniertem Verhalten*, das von Compilern in der Regel auf undefinierte Weise behandelt wird. Sie ignorieren es zum Beispiel bei Optimierungen und führen keine Überprüfungen durch, die es erkennen könnten. Das führt oft zu schwer lokalisierbaren Fehlern und Sicherheitslücken, beispielsweise wenn ein Pufferüberlauf Daten im Speicher zerstört.

Vorhandene Fehlererkennungswerkzeuge, die das Programm zum Erkennen von undefiniertem Verhalten instrumentieren, benutzen oft Compiler, die den Code so optimieren, dass Fehler nicht mehr erkannt werden. Alternativ könnten unsichere Programme in einer sicheren Sprache wie Java neu geschrieben werden, die vollständig definiert und sicher ist. Dies würde jedoch für viele Projekte zu untragbar hohen Kosten führen.

Um undefiniertes Verhalten in den Griff zu bekommen, haben wir einen Ansatz entwickelt, bei dem unsichere Sprachen auf der Java Virtual Machine ausgeführt werden. Der Ansatz wurde unter dem Namen *Safe Sulong* implementiert, einem System, das auf einem in Java geschriebenen Interpreter für unsichere Sprachen basiert. Indem der Interpreter auf die Wohldefiniertheit und die automatischen Laufzeitprüfungen von Java vertraut, kann er Pufferüberläufe und andere Fehler zur Laufzeit erkennen und das Programm in solchen Fällen abbrechen. Safe Sulong merkt sich Metadaten wie Typen und Objektgrenzen, die Programmierern über eine Introspektionsschnittstelle zur Verfügung gestellt werden, mit denen sie Fehler abfangen und zusätzliche Prüfungen implementieren können. Der Interpreter unterstützt auch unstandardisierte Elemente in C-Code wie gängige Inline-Assembly-Codestücke und GCC-Builtins. Zu diesem Zweck haben wir zunächst die Verwendung solcher Elemente in einer großen Anzahl von Open-Source-Projekten untersucht.

Sulong wird in der GraalVM verwendet, einer kommerziellen, mehrsprachigen virtuellen Maschine. Sulong erlaubt die Definition von Schnittstellen zu nativem Code, wodurch unsere Sicherheitsmechanismen auch verwendet werden könnten um die Ausführung nativer Erweiterungen anderer Sprachen wie Ruby, Python und R sicherer zu machen.

Acknowledgments

This thesis would not have been possible without the help of many people. First and foremost, I want to thank my advisor Hanspeter Mössenböck, for being an excellent mentor, for his constant feedback and support, for allowing me to freely follow my research interests, and for maintaining a successful and mutually-beneficial collaboration with Oracle Labs. I am thankful to Cristian Cadar for the time and effort he put into evaluating this work and providing feedback. I also want to thank the other members of the dissertation jury for their time, namely Armin Biere and Paul Grünbacher.

I sincerely thank Stefan Marr for teaching me much about academia and for his mentoring, which helped me to develop myself during the second half of my PhD. I want to thank Matthias Grimmer who mentored me and taught me the essential academic skills during the first year of my PhD. I am thankful to René Mayrhofer for his feedback from the viewpoint of a security researcher. I am grateful to Stephen Kell for his insightful feedback on my work, and Bram Adams for his help with improving the study on the use of GCC builtins. I want to thank Ingrid Abfalter for correcting my papers and improving my writing.

I would like to express my gratitude to Thomas Würthinger for funding my research, and his trust for letting me develop an important puzzle piece of the Graal project. I am thankful to all of the Oracle Labs employees who worked together with me, shared their knowledge, helped me debug issues, gave feedback on my work, and developed the Sulong research prototype into a product. I especially want to thank Roland Schatz, Matthias Grimmer, Lukas Stadler, Chris Seaton, and Christian Wimmer.

I want to thank my fellow PhD students, Josef Eisl, David Leopoldseder, and Benoit Daloze for their feedback and help. I want to thank the students that worked on Sulong-related topics, all of which were excellent: Jacob

Kreindl, Daniel Pekarek, Thomas Pointhuber, and Raphael Mosaner.

I am grateful for my family and friends. I want to thank Jin Ting for her love, and for her patience even when I worked on evenings and weekends. I want to thank my parents, sister, and grandparents for all their support.

Contents

Statutory Declaration **iii**

Abstract **v**

Kurzfassung **vii**

Acknowledgments **ix**

Contents **x**

I Introduction and Overview **1**

1 Introduction **3**

1.1 Problem . . . 3
1.2 State of the Art . . . 4
1.3 Remaining Challenges . . . 8
1.4 Suggested Approach . . . 9
1.5 Contributions . . . 10
1.6 Limitations . . . 14
1.7 Project Context . . . 15
1.8 Research Context . . . 18
1.9 Outline . . . 20

2 Overview **21**

2.1 Sulong . . . 21
2.2 Introspection . . . 29
2.3 Empirical Studies . . . 30

3 Publications **35**
3.1 Journal Papers 35
3.2 Conference Papers 35
3.3 Full Workshop Papers 37
3.4 Other Publications 37
3.5 Under Review 38

Bibliography **38**

II Publications **61**

4 Native Sulong **63**

5 Safe Sulong **75**

6 Lenient C **91**

7 Introspection **105**

8 Context-aware Failure-oblivious Computing **137**

9 A Study on the Use of Inline Assembly **153**

10 A Study on the Use of GCC Builtins **171**

III Future Work and Conclusion **185**

11 Future Work **187**
11.1 Safe Sulong 187
11.2 Introspection 191
11.3 Empirical Studies 192

12 Conclusion **195**

Part I

Introduction and Overview

Chapter 1

Introduction

1.1 Problem

In unsafe languages like C, the semantics of operations are only specified for valid input. For example, while dereferencing a valid pointer is well specified, the effect of dereferencing an out-of-bounds pointer, which is known as a *buffer overflow*, is not specified by the C standard.[1] Such an error is said to cause *Undefined Behavior*. Compilers are not required to produce code that detects undefined behavior while the program executes, so they do not, for example, insert bounds checks to detect buffer overflows. In fact, state-of-the-art compilers such as Clang or GCC even optimize the program based on the assumption that undefined behavior never occurs [139].

Executing programs with undefined behavior that have been compiled by Clang or GCC can yield various unintended or even disastrous results. For example, out-of-bounds reads can result in sensitive data of adjacent objects being read, even when those objects were not explicitly referenced [125]. Examples for disastrous out-of-bounds reads were Heartbleed in the OpenSSL SSL library [35] and Cloudbleed in the online service Cloudfare [46], which both allowed attackers to leak sensitive information on web servers, and affected millions of users. Attackers might not only exploit buffer overflows to read private data; out-of-bounds writes can be exploited to overwrite control-flow data, allowing attackers to divert the control flow of the program and even to take control over the process [114, 18, 9, 132]. Furthermore, a pro-

[1]If not further specified, we refer to the ISO/IEC 9899:2018 standard, which is informally known as C17 or C18.

gram fragment that causes undefined behavior can be compiled to code with unexpected behavior or can even be removed altogether by the compiler, resulting in “miscompilations” that are hard to debug [139]. Sometimes, a program that causes undefined behavior can work correctly, for example, when an operation is compiled so code that happens to behave as expected by the programmer (e.g., a signed integer addition might get compiled to a x86 `add` instruction, which wraps around on an overflow instead of causing undefined behavior). However, such bugs are still “ticking timebombs” as a compiler update might introduce an optimization that takes advantage of the undefined behavior [140].

Implementing bug-finding tools that tackle undefined behavior in languages like C/C++ is a challenge. First, parsing C/C++ requires a high implementation effort due to the alternative syntax options that predate the first ANSI C standard.[2] Furthermore, C projects not only rely on C code but also use preprocessor macros [38], compiler builtins [104], inline assembly [105], and other compiler extensions. Thus, to reduce the implementation effort, a number of bug-finding tools are based on Clang or GCC, either by instrumenting the program under test in their intermediate representations or by instrumenting the binary produced by them [117]. This is somehow undesirable, since undefined behavior can be optimized away by these compilers, and is then no longer detectable during execution.

1.2 State of the Art

Both industry and academia have been tackling undefined behavior, in particular buffer overflows, for decades; thus, a plethora of approaches exist to tackle it in various ways [20, 135, 128, 117]. In the following, we describe the most important categories of approaches and research trends, and highlight their drawbacks that we want to address. Note that the summary is necessarily incomplete.

Research trends in tackling undefined behavior can be roughly categorized into software-based and hardware-based enhancements. Important software-based approaches are instrumentation-based bug-finding tools, heuristic approaches, static analysis, and symbolic execution.

[2]ANSI C was first standardized in 1989 as ANSI X3.159-1989.

Instrumentation-based bug-finding tools Many dynamic bug-finding tools (sometimes called *sanitizers*) detect specific classes of undefined behavior by inserting instrumentation code into a program [117]. These tools detect bugs while executing the program, and thus must be supplied with concrete program inputs (e.g., program arguments or command line input). While sanitizers detect errors in the program without false positives (i.e., errors indicated by the bug-finding tool are always real bugs), one of their major drawbacks is that they only detect bugs that are triggered during execution, and overlook those that would be triggered with different input. Thus, they are often complemented with fuzzers [89], which mutate the program's input to reach a higher path coverage. Furthermore, sanitizers typically result in a high overhead compared to an unsafe execution of the program.

Compile-time instrumentation sanitizers [55, 112, 94, 95, 71, 124] insert instrumentation on the source or compiler IR level, and are typically expected to detect all errors of one or several error categories during execution of the program. Since our goal is to reliably detect all bugs during the execution of a program, we implemented Safe Sulong as a compile-time-instrumentation sanitizer. In contrast, dynamic binary instrumentation [11, 97, 85] allows the addition of checks to the executable when it is started [12, 113]. A major advantage of such approaches is that they operate solely on the executable and do not require the original source code. However, binary code often lacks information that was present on the source code level, so only a subset of undefined behavior can be detected.

Heuristic and attacker-mitigation approaches A number of approaches attempt to raise the bar for attackers to exploit undefined behavior (specifically memory errors). Data Execution Prevention [134] marks data as non-executable and was an early mechanism to restrict the injection of code into data, which is now widely used. Address Space Layout Randomization [129], which is also widely used in practice, randomly assigns the position of stack, heap, and libraries, so that exploits cannot rely on fixed addresses. Similar approaches also exist for finer-grained randomization [72], to randomize instruction sets [70] and the locations of objects in the heap [5, 98]. Pointer encryption [22] encrypts addresses stored in memory and decrypts them before an access. Similarly, Data Space Randomization [7] randomizes the representation of data in memory. Software-diversity approaches [79] aim to compile identical source code to different binaries in order to reduce the chance that an exploit runs on all systems that use the program. Multi-variant execution systems [24, 63] combine mitigation mechanisms by running multiple versions of a program to check for divergences during executions, in which case the program is terminated. Stack canaries [23] are inserted on the stack between the return address and local variables; overwrites of this address caused by a stack buffer overflow can be detected by checking if the stack canary is also overwritten. Control-flow Integrity [1, 13] can be used to verify the control flow of a program during its execution and can often be configured to control the granularity of the protection.

All heuristic and mitigation approaches do not aim to reliably detect errors and are prone to attacks. For example, Address Space Layout Randomization can be circumvented by information disclosure [125], side-channel attacks [47], and by using brute-force attacks on 32-bit systems [115]. However, many of these approaches add another level of defense that has to be tackled by attackers while only causing a low overhead [30].

Static software-based approaches Static approaches detect bugs by analyzing the source code without assuming a specific program input. Thus, they can also detect bugs that would only be rarely triggered when running an application. Static analysis tools for unsafe languages reach from simple rule checkers [69, 136, 21] to more advanced analyses that assume all potential program states to detect undefined behavior [137, 39, 62, 154, 32]. A major disadvantage of these tools is that they either need to overapproximate or underapproximate the behavior of the program. Overapproximation guarantees soundness, that is, the analyzer detects all bugs but might also yield false positives. This is a burden for the programmer, who has to determine which potential bugs need to be fixed [68]. Underapproximation guarantees completeness, that is, it yields no false positives but might overlook bugs. To balance these tradeoffs, many approaches sacrifice both soundness and completeness [39, 156, 136] and thus do not provide complete protection against undefined behavior.

Symbolic execution engines Symbolic execution is a hybrid approach that combines aspects of static and dynamic approaches [73, 4]. With symbolic execution, multiple paths through the program are explored simultaneously by assuming symbolic input values. While doing so, *path constraints*, which describe which branch an execution state took, are recorded. Finally, a solver is used to check whether the executed path is feasible and whether any bugs are triggered along the way. Modern symbolic execution engines [16, 15] typically combine symbolic execution with concrete execution to incorporate elements that are not instrumented by symbolic execution (e.g., system calls). While symbolic execution is an effective way to detect bugs, scaling issues such as *path explosion* make it infeasible to exhaustively explore real-world programs [17].

Safe Programming Languages Safe languages like Java or C# provide strict language semantics [40]. A safe systems programming language that provides memory safety by relying on ownership [131] instead of a garbage collector is Rust[3], which has recently gained much attraction. To reduce the effort of porting programs, several safer, C-like languages have been proposed such as Cyclone [67] and CCured [96]. However, they still require program changes (about 8% of the code had to be adapted for Cyclone [53]) and they have not been widely adopted in practice. Besides, there have also been proposals for assigning semantics to some instances of undefined behavior in C (e.g., *Friendly C* [25]). Due to disagreements in what the desired semantics would be, this approach has not been adopted in practice [102]. As part of our work on Safe Sulong, we also devised a C implementation that replaces instances of undefined behavior by a fixed semantics and is suitable for executing C programs on the Java Virtual Machine. Failure-oblivious computing [111] is a technique that defines semantics for out-of-bounds accesses; illegal write accesses are ignored, and illegal read accesses produce predefined values. However, returning predefined values for out-of-bounds reads cannot work in all contexts; to address this, we worked on making our approach *context aware* as part of our introspection work.

Hardware-based approaches Recent research has proposed hardware-based mechanisms to enable a safer execution, primarily with respect to

[3]`https://www.rust-lang.org`

memory safety. Hardbound [31] is an approach in which *fat pointers* were implemented in hardware, that is, pointers that additionally have base and bounds information attached. Watchdog [93] also detects use-after-free errors by associating each memory region and each pointer with an identifier that can be tagged as invalid when the memory is freed. CHERI [146, 19] is a RISC-based architecture in which pointers are treated as capabilities, which control what can be done with the pointer, to enable fine-grained memory safety and access control. Intel introduced security-related instruction-set extensions such as MPX [66], which extended the x86 architecture with instructions for tracking, manipulation, and querying bounds information [99]. If such hardware-based approaches are widely available, they potentially have a high impact, since they often can implement functionality more efficiently than software approaches that are based on general-purpose instructions. Thus, we also evaluated our introspection work using Intel MPX. However, a major disadvantage of hardware-based approaches is that such extensions require the replacement of hardware, which is costly.

1.3 Remaining Challenges

Our main goal was to improve on existing sanitizers and provide complete protection against memory errors and other categories of undefined behavior. We identified three main issues that we tackled with our research.

Unsafe optimizations Existing sanitizers overlook errors when compiler optimizations are turned on; if optimizations are turned off, the performance of these approaches is prohibitively slow. Furthermore, adding run-time checks to binaries (either by the compiler or via instrumentation) is error-prone, since instrumentation for corner cases can be forgotten, which compromises the bug-detection capabilities. Consequently, we believe that a novel approach is required that provides comprehensive protection against undefined behavior, cannot optimize undefined behavior away, but is still fast enough to be used in practice.

Missing programming interfaces Existing sanitizers track metadata as part of their runtimes to implement their checks. However, they do not expose this metadata to programmers, who could use it in their programs, for example, to check assertions. We believe that this metadata could be exposed as part of an interface that could be implemented by existing bug-finding tools.

Lack of understanding for inline assembly and compiler builtins The implementation of both static and dynamic tools for unsafe languages like C is a major effort. C programs contain, for example, unstandardized elements like inline assembly and compiler builtins. To the best of our knowledge, current compile-time instrumentation approaches fail to detect undefined behavior in them. Providing complete support for such elements would require supporting various architectures with hundreds or thousands of different instructions or builtins. Researching their usage would alleviate this problem by allowing tool developers to prioritize the implementation of error checks for such elements.

1.4 Suggested Approach

To tackle undefined behavior in C programs, we devised an approach for the execution of unsafe languages on the Java Virtual Machine. The core idea of this approach is that the semantics of an unsafe program can be mapped to the safe semantics of a Java program. By mapping an unsafe operation to a sequence of equivalent Java operations, both executions behave the same for legal inputs. However, since Java is a safe language and fully specifies the semantics of its operations, the Java code also behaves in a well-defined way for inputs that would be illegal in the C code. The Java Virtual Machine optimizes the program based on the well-defined semantics of Java. Thus, buffer overflows and other errors are not optimized away. We implemented this approach as a system called Safe Sulong.

Safe Sulong has two execution modes that react differently to undefined behavior. The first execution mode detects undefined behavior and aborts execution. It relies on the automatic run-time checks of the underlying Java Virtual Machine. This mode allows Safe Sulong to be used as a sanitizer, and is useful especially for programmers that can then fix their programs during development and testing. However, it prevents software deployers from executing programs that cause undefined behavior even when they consider it to be benign. To allow the execution of such programs, we came up with a second execution mode, based on a *Lenient C* implementation that assigns semantics to undefined behavior in a way that programmers would expect. This implementation is similar to *Friendly C*, but has been designed to be implemented on a managed runtime.

To reduce the implementation effort of Safe Sulong, we based our approach on the LLVM framework [80]. Our system executes LLVM IR, the RISC-like intermediate representation of the LLVM framework. Various LLVM frontends can parse different input languages and compile them to LLVM IR. For example, Clang can compile C/C++ to LLVM IR. When compiling an unsafe language to LLVM IR, we disable all optimizations so that operations that cause undefined behavior are not optimized away. While using Clang relieves us from dealing with preprocessor macros and the various C syntax standards, LLVM IR still contains inline assembly and compiler builtins. We conducted two empirical studies on the usage of these elements to prioritize their implementation in Sulong and other tools.

Safe Sulong is an automatic approach to tackle undefined behavior. In some cases, however, programmers would know how to better react to such behavior; for example, by checking for an erroneous condition and recovering from an error. However, C objects do not have metadata such as bounds information attached to them. Since an increasing number of bug-detection and bug-finding tools track metadata to implement additional checks, we came up with an introspection interface to allow programmers querying this metadata. Thus, programmers can complement Safe Sulong's automatic approach by implementing additional logic that queries the metadata to avoid undefined behavior. We also propose an extension to failure-oblivious computing to incorporate the semantics of the function in which the error occurs to recover from the error.

1.5 Contributions

The contributions presented as part of this thesis can be divided into scientific contributions, technical contributions, and publications.

1.5.1 Scientific Contributions

The scientific contributions of this thesis can be categorized in three parts:

Safe Sulong We present a novel approach to safely executing potentially unsafe LLVM-based languages by executing them on a Java Virtual Machine. We present a bug-finding mode that can detect out-of-bounds accesses and other errors, which we evaluated in terms of bug-finding effectiveness on open-source projects and in terms of performance [109]. Furthermore, we devised *Lenient C*, an implementation of C on the JVM that replaces undefined behavior with commonly expected behavior [107].

Introspection We present a novel approach that enables programmers to increase the robustness of their C libraries. This approach is based on an introspection interface that we designed to allow programmers querying metadata such as bounds or types. We evaluated this approach in a case study of a libc implementation for Safe Sulong [108]. We showed that introspection can also be supported in other approaches such as LLVM's AddressSanitizer [112], MPX-based bounds instrumentation [99], and SoftBound+CETS [106]. Furthermore, we evaluated the performance overhead and the effectiveness of introspection to prevent buffer overflows in a case study of real-world bugs [106].

Empirical studies We present two empirical studies on the usage of unstandardized elements in C projects, to help tool developers to prioritize these features when implementing tools for C code. In these studies, we analyzed the usage of x86-64 inline assembly [105] and the usage of GCC builtins [104] in a large number of open-source projects. In the GCC builtin study, we also analyzed the usage of builtins over the lifetime of a project. To the best of our knowledge, these are the first studies on the usage of inline assembly and compiler builtins.

1.5.2 Technical Contributions

The major technical contributions include the design and implementation of Native Sulong and Safe Sulong.

Native Sulong Native Sulong is the core of Sulong, but does not provide memory safety, since dynamically allocated objects are stored in unmanaged memory. It is available as open-source software and is highly popular on GitHub.[4] The author contributed the initial design and is the main contributor (out of 35 contributors) with almost 600 commits. Note that we released the software without the commit history of the first half year, which included only commits of the author. Other main contributors include the Oracle employees Matthias Grimmer (with almost 500 commits), who, together with Roland Schatz (with over 200 commits), worked on turning Native Sulong into a product after the paper about Native Sulong was published. From Johannes Kepler University Linz, Jacob Kreindl (with 440 commits) worked on rewriting the parser to improve its efficiency[5] and implemented source-level debugging of LLVM IR. Thomas Pointhuber (with almost 200 commits) worked on testing Sulong, and Daniel Pekarek (about 160 commits) worked on the implementation of inline assembly and the Linux syscall interface.[6] Besides other employees of the Johannes Kepler University Linz and Oracle, Colin Barrett and Swapnil Gaikwad from the University of Manchester contributed code.

Safe Sulong Safe Sulong is based on Native Sulong, but uses Java objects instead of objects in native memory. The author contributed the initial design and is the main contributor with 1012 commits out of overall 1034 commits. Safe Sulong is no longer actively maintained. Instead, a hybrid version that is based on both Native Sulong and Safe Sulong is being developed, and is maintained by Oracle Labs.

1.5.3 Supporting Publications

The results of this thesis were published at various venues. Below, the papers most relevant to the scientific contributions are listed, categorized by their contribution and sorted in chronologically ascending order. Section 3 provides a full list of publications.

[4]https://github.com/graalvm/sulong

[5]The initial parser was based on the *LLVM IR SDK* Eclipse plugin by Alon Mishne (see `https://github.com/amishne/llvm-ir-editor`). Since the plugin was designed for usage in an Integrated Development Environment and was based on the textual format of LLVM IR, we replaced it by a parser designed for efficiency and for the binary format of LLVM IR.

[6]`http://man7.org/linux/man-pages/man2/syscalls.2.html`

Sulong

Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. **Manuel Rigger**, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages* (VMIL 2016), Amsterdam, Netherlands, 2016, pp. 6–15 DOI: 10.1145/2998415.2998416

Lenient Execution of C on a Java Virtual Machine: Or: How I Learned to Stop Worrying and Run the Code. **Manuel Rigger**, Roland Schatz, Matthias Grimmer, Hanspeter Mössenböck. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes* (ManLang 2017), Prague, Czech Republic, 2017, pp. 35–47 DOI: 10.1145/3132190.3132204 (AR: 45%)

Sulong, and Thanks for All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. **Manuel Rigger**, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, Hanspeter Mössenböck. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2018), Williamsburg, VA, USA, 2018, pp. 377–391 DOI: 10.1145/3173162.3173174 (AR: 18%)

Introspection

Introspection for C and its Applications to Library Robustness. **Manuel Rigger**, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, Hanspeter Mössenböck. In *The Art, Science, and Engineering of Programming* (Programming 2018)

Preventing Buffer Overflows by Context-aware Failure-oblivious Computing. **Manuel Rigger**, Daniel Pekarek, Hanspeter Mössenböck. In *Proceedings of the 12th International Conference on Network and System Security* (NSS 2018), Hong Kong, China, 2018 (AR: 39%)

Empirical studies

An Analysis of x86-64 Inline Assembly in C Programs. **Manuel Rigger**, Stefan Marr, Stephen Kell, David Leopoldseder, Hanspeter Mössenböck. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (VEE 2018), Williamsburg, VA, USA, 2018, pp. 84–99 DOI: 10.1145/3186411.3186418 (AR: 32%)

Understanding GCC Builtins to Develop Better Tools. **Manuel Rigger**, Stefan Marr, Bram Adams, Hanspeter Mössenböck.

1.6 Limitations

Our approach of safely executing LLVM IR has various limitations, of which the most significant ones are highlighted in this section.

Side-Channel Attacks In Sulong, we did not consider side channel attacks [118], which have recently received much attention due to speculative execution vulnerabilities that have been discovered in processors [74]. Side channels allow attackers to indirectly gain information which the system leaks, for example, through timing differences [6]. Since Sulong is based on dynamic compilation, it provides a higher attack surface for side channel attacks due to profiling-based optimizations [100].

Unsupported Features Our goal for Sulong was to execute common C applications. Sulong does not provide low-level features that are needed by operating systems. For example, it does not provide any means for directly accessing memory or devices. Furthermore, Safe Sulong can only execute programs that do not depend on binaries of which neither the source code nor the LLVM IR is available. This requirement is not always met in practice (e.g., through closed-source third-party libraries) [128]. We believe that a hybrid execution approach could alleviate these issues (see Chapter 11).

Insufficient Evaluation While the performance results of Sulong are promising, we have not evaluated it on large applications such as the SPEC benchmarks[7] or browsers. This is mainly due to libc functions missing from Safe Sulong's libc. This could be addressed by running an existing complete implementation of a libc on Safe Sulong, which we have recently achieved for Native Sulong. Another reason is that Sulong needs prohibitively long to run these benchmarks because it spends a long time in interpreted loops before they are compiled. As part of future work, we want to address this with on-stack replacement, a technique that allows switching from an interpreted version of a loop to a compiled version [41]. Overall, a threat to external validity is that the performance results cannot be generalized to such applications.

1.7 Project Context

The work described in this thesis was done at the Institute for System Software at the Johannes Kepler University Linz, Austria (JKU). The institute primarily researches topics in the areas of compilers, virtual machines, programming languages, performance monitoring, and software engineering. It maintains several collaborations with industry.

Collaboration with Oracle The project described in this thesis was done as part of a long-running research collaboration with Oracle (formerly Sun Microsystems). The collaboration started in 2000, when Hanspeter Mössenböck, head of the Institute for System Software, enhanced the intermediate representation of the HotSpot client compiler as part of his sabbatical and implemented a graph-coloring register allocator [92]. As part

[7] `https://www.spec.org/benchmarks.html`

of this collaboration, several PhD students worked on enhancements of the Java HotSpot VM and its client and server compilers:

- Thomas Kotzmann et al. worked on escape analysis in HotSpot's client compiler to replace Object allocations by scalar values where possible [76, 75]. He completed his PhD thesis on *Escape Analysis in the Context of Dynamic Compilation and Deoptimization* in 2005.

- Christian Wimmer et al. worked on the inlining of objects into their referencing objects to eliminate unnecessary field accesses [142, 143, 144, 145]. He completed his PhD thesis on *Automatic Object Inlining in a Java Virtual Machine* in 2008.

- Thomas Würthinger et al. proposed a mechanism for unlimited redefinition of loaded classes at run time in the HotSpot VM [147, 152]. He completed his PhD thesis on *Dynamic Code Evolution for Java* in 2011.

- Christian Häubl et al. worked on trace-based dynamic compilation of Java code [56, 58, 59]. This work was funded by the Austrian Science Fund (FWF). He completed his PhD thesis on *Generalization of Trace Compilation for Java* in 2015.

Furthermore, the collaboration contributed a linear-scan register allocation in the client compiler [91, 141], an array-bounds-check-elimination algorithm in the client compiler [149, 151], a visualization tool for the HotSpot server compiler [150], a description of the HotSpot client compiler design [77], an optimization of the JVM's string representation [57, 65], and an efficient implementation of Java continuations [121] and Java coroutines [123].

Collaboration with Oracle Labs After Oracle Labs opened a research lab at JKU, the focus of the collaboration shifted towards the Truffle and Graal projects [148]. Truffle is a language implementation framework on which also Sulong is based. The Graal compiler is used by Truffle to compile frequently-executed functions to machine code at run time. Besides the work described in this thesis, the following PhD students have been working in this collaboration or completed their degree:

- Lukas Stadler et al. worked on the core design of Graal [119, 120] and on partial-escape analysis to enable scalar replacement [122] to increase the performance of the generated code. He completed his PhD thesis on *Partial Escape Analysis and Scalar Replacement for Java* in 2014.

- Matthias Grimmer et al. devised a language interoperability mechanism for Truffle [51] and showed how it can be used to implement native function interfaces [52]. He completed his PhD thesis on *Cross-Language Interoperability in a Multi-Language Runtime* in 2015.

- Gilles Duboscq et al. worked on Graal's core design [34], on the reduction of deoptimization metadata [33], and on loop unrolling in the Graal compiler. He completed his PhD thesis on *Aggressive Loop Optimizations in a JIT Compiler* in 2016.

- Benoit Daloze et al. came up with thread-safe and efficient data representations in dynamically-typed languages that he implemented for TruffleRuby [28, 27, 29].

- Josef Eisl et al. has been working on a new register allocation approach (trace register allocation) in the Graal compiler to balance register allocation time and the performance of the generated code [36, 37].

- David Leopoldseder et al. has been working on code duplication and loop unrolling optimizations in the Graal compiler to increase the performance of the generated code [83, 82].

Role in the Collaboration The author of this thesis joined the collaboration with Oracle Labs as a Bachelor student in 2011, when he implemented a Truffle Python interpreter. Between 2012 and 2014, he worked on a Truffle implementation for C together with Matthias Grimmer. The author described his part of the work in his Master's thesis *Truffle/C Interpreter*, and Matthias Grimmer described his part in his Master's thesis *Truffle/C Runtime Environment*. The work on Truffle/C resulted in two conference papers [49, 48]. Grimmer extended the C interpreter by memory safety as an interpreter called *ManagedC* and published promising early results in a workshop paper [50], which later motivated the implementation of Safe Sulong.

ManagedC ManagedC was an important step towards execution of unsafe languages on the JVM, and Safe Sulong is based on its initial architecture.

However, ManagedC could execute only C code since it was not based on an IR, while Safe Sulong can execute also other languages for which an LLVM front end exists. Like Safe Sulong, ManagedC provided two execution modes, a strict and a relaxed one. Safe Sulong's bug-detection mode is similar to ManagedC's strict mode. However, ManagedC could detect also accesses to uninitialized memory, but, unlike Safe Sulong, could not detect invalid free errors (e.g., calling `free()` on a stack allocation) and illegal accesses to variadic arguments. ManagedC's relaxed execution mode allowed reading uninitialized memory and reading objects assuming an incorrect type. In contrast, our Lenient C execution mode is more comprehensive and also assigns a lenient semantics for various other operations. Finally, we evaluated Safe Sulong on larger programs, which we obtained from GitHub, which required a higher level of completeness. Unlike ManagedC, we implemented our libc in C (and not in Java), added support also for libc functions that are difficult to implement (e.g., for signal handling), and implemented inline assembly instructions and GCC compiler builtins.

Impact in the Collaboration In September 2015, the author of this thesis started as a PhD student, funded by Oracle Labs, to work on the execution of LLVM IR on Truffle. After the author of this thesis implemented a prototype of Sulong and published the results, both Oracle and student researchers continued on its implementation; Sulong is now in a mature state and is distributed as part of GraalVM.[8] GraalVM is a multi-lingual VM, and contains several language implementations including JavaScript, Ruby, R, and Python. GraalVM's language interoperability mechanism allows these language implementations to use Sulong as a native function interface [52]. Sulong has also been used in other projects within Oracle. For example, Iraklis et al. developed a *smart array* data structure [101] in which Sulong is used.

1.8 Research Context

The research presented in this thesis is interdisciplinary and intersects with various computer science disciplines as outlined below and as indicated by the diverse venues of the publications.

[8]http://www.graalvm.org/

Virtual Machine Construction *Virtual Machine Construction* is concerned with the design, implementation, and evaluation of virtual machines. Virtual machines typically execute a virtual instruction set and provide services such as garbage collection and dynamic compilation for the language implemented. Many of the concepts which are used in today's virtual machines (e.g., the Hotspot JVM [77]), such as dynamic deoptimizations [61] and polymorphic inline caches [60], were pioneered in VMs for the Self language. Sulong is part of the GraalVM and exploits these VM concepts while executing LLVM-based languages. Challenges in the design of Sulong included how to exploit dynamic compilation for optimal run-time performance [103, 109], and how to represent unmanaged memory as managed objects [107, 109]. Production VMs such as the HotSpot VM have been actively developed for decades. Thus, for a thesis like this, a major *practical* challenge was to keep the implementation effort manageable by combining existing components such as LLVM, Truffle, and Graal [103] while being competitive with existing VMs and compilers.

Information Security *Information Security* is concerned with the protection of data. Buffer overflows are among the primary threats, as attackers can exploit them to read private data or corrupt it; both academia and industry have been tackling buffer overflows for decades [20]. As part of this effort, we have been working on Safe Sulong to safely execute LLVM-based languages and automatically detect buffer overflows and other errors [109]. Additionally, our work on introspection aims to provide programmers with library functions that, for example, compute buffer sizes, which can be used to prevent buffer overflows [106].

Programming Languages Programming language research is about defining and understanding programming languages. In Safe Sulong, we explored how an unsafe language can be efficiently implemented based on the semantics of a safe language. Furthermore, we proposed a modification to the C11 standard to assign fixed semantics to undefined behavior [107]. Our introspection work [108, 106] explored how programmers could use additional builtin functions to tackle buffer overflows and other errors.

Empirical Software Engineering *Software engineering* is concerned with all aspects of software production to improve the building and maintenance of software [116]. *Empirical Software Engineering* achieves this by determining the usage of software in practice. We conducted two empirical studies in which we analyzed the usage of inline assembly [105] and GCC builtins [104] in open-source C projects in order to help tool developers to prioritize the implementation of such features. Specifically, we relied on techniques from *Software Repository Mining* [54], a subdomain that aims to answer empirical questions by analyzing projects hosted on GitHub or other open-source repositories. Additionally, we analyzed the frequency of various memory-safety error categories to determine the relevance of different error classes [109].

1.9 Outline

This thesis is structured into three parts. Part I puts the dissertation into context and explains how the published papers relate to each other. It also contains a full list of the published papers. Part II includes selected publications, which are outlined in the next chapter. Part III describes future work, summarizes the thesis, and draws some conclusions.

Chapter 2

Overview

This chapter gives an overview of the thesis' contents and describes how the papers in Part II relate to each other. All papers contribute directly or indirectly (i.e., through empirical studies) towards the goal of tackling undefined behavior in unsafe languages. As Figure 2.1 shows, the publications can be grouped into three research areas that they support: the safe and efficient execution of LLVM IR on the Java Virtual Machine (Sulong), a mechanism that allows programmers to use metadata tracked by existing tools to manually tackle undefined behavior (Introspection), and studies on the usage of inline assembly and GCC compiler builtins that support the development of tools (Empirical Studies).

2.1 Sulong

Sulong is a system that executes LLVM IR on the Java Virtual Machine. Its components are displayed in Figure 2.1, which are detailed in the remainder of this section. Sulong relies on existing LLVM front ends, such as Clang for C/C++, to compile programs to an intermediate program representation called LLVM IR. We implemented a Truffle-based LLVM IR interpreter that executes this LLVM IR on the GraalVM. The interpreter can be configured to use various execution modes that differ in how they handle undefined behavior.

Unsafe Languages Sulong executes C and other unsafe languages, in which the semantics of operations with illegal input is not defined. For ex-

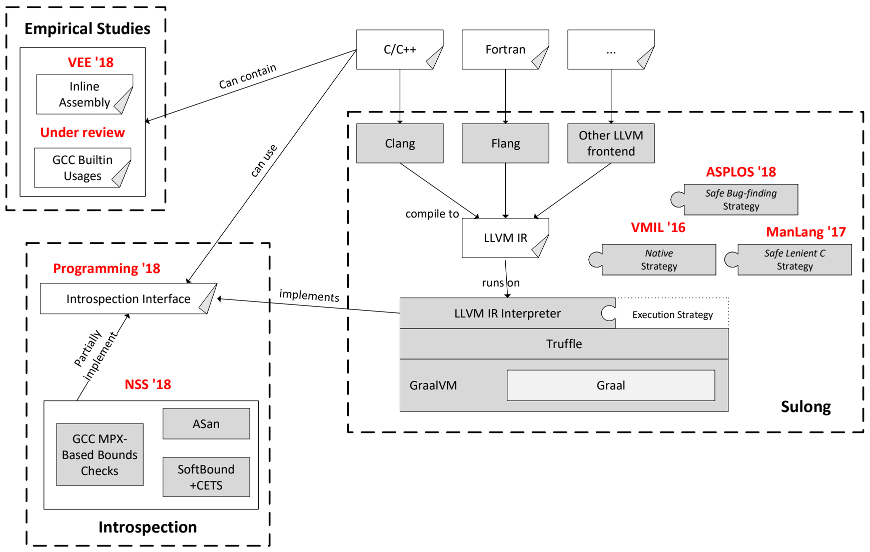

Figure 2.1: Relationship between the papers presented in this thesis; paper venues are displayed using a red and bold font.

ample, Listing 2.1 shows a C function that allocates an array with `size` elements and initializes them with random values obtained by calling `rand()`. In this example, the caller of `create_random_arr()` can cause undefined behavior depending on the `size` value that is passed. For example, if compiled on AMD64 and if `LONG_MIN` is passed, which corresponds to the value -2^{63}, then the signed multiplication `sizeof(long) * size` causes an integer underflow, the semantics of which are undefined. Generally, the C standard “imposes no requirements” on the behavior of programs with undefined behavior. However, the standard gives examples for possible outcomes:

> “Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).”

In practice, compilers attempt to detect undefined behavior statically to print warnings [126, 127]. However, this is not always possible; for the example in Listing 2.1, Clang fails to print an error message. Instead, it

Listing 2.1: A C function that produces a random array of long values

```
long* create_random_arr(long size) {
    long* numbers = malloc(sizeof(long) * size);
    for (size_t i = 0; i < size; ++i) {
        numbers[i] = rand();
    }
    return numbers;
}
```

compiles the multiplication to a left shift, which produces `0` for `LONG_MIN`, causing `malloc()` to not allocate any memory. Since `size` is promoted to an unsigned integer in the loop condition, the loop is erroneously executed 2^{63} times. Thus, the out-of-bounds writes to `number[i]` are likely to corrupt the heap and/or crash the program. Sulong's LLVM IR interpreter does not directly execute such source code programs; they first need to be compiled to LLVM IR using an LLVM front end.

LLVM Frontends Sulong can execute various programming languages by relying on existing LLVM front ends for parsing these languages [80]. This includes unsafe languages such as C, C++, and Fortran, but also safe languages such as Rust and Haskell. The Clang LLVM front end[1] can parse C and C++, while Flang[2] and DragonEgg[3] can parse Fortran. We evaluated Sulong mainly on C, as it is the most popular unsafe language according to the TIOBE index [130].

In theory, most other languages should also run on Sulong as they are all compiled to LLVM IR instructions. We evaluated Native Sulong also on Fortran (see Section 4). Although not systematically evaluated, Native Sulong is also able to run C++ programs.[4]

Safe Sulong currently only supports C programs. Supporting additional languages would require also running their standard libraries as LLVM IR code. Typically, such standard libraries rely on system calls [133], which Safe Sulong does not yet implement. To support C programs in Safe Sulong,

[1]`https://clang.llvm.org/`
[2]`https://github.com/flang-compiler/flang`
[3]`https://dragonegg.llvm.org/`
[4]To support C++, LLVM IR instructions for exception handling were implemented as part of the productizing effort.

we thus implemented libc functions in a custom libc that does not rely on system calls.

LLVM IR LLVM IR is the intermediate program representation in the LLVM framework [80]. By executing LLVM IR, Sulong can execute programs written in various programming languages with a moderate implementation effort. Listing 2.2 shows the code produced by compiling the previous C function to LLVM IR. It comprises three basic blocks that consist of sequential instructions and end with a control-flow instruction that determines which basic block to execute next. The first and the third basic block end with a `br` instruction which branches to one of two labels depending on a condition value; the second block ends with a `ret` instruction that returns a value from the function. Local variables, also called virtual registers, are preceded by a `%`. These variables are in Static Single Assignment form [26], which means that there is just a single instruction in which a value is assigned to them. To represent conditional assignments, phi functions are used that merge variants of the same variable [26]. LLVM IR does not provide instructions to allocate heap memory; the call to `malloc()` is resolved by dynamically linking it to a precompiled function that is part of libc. Since LLVM IR has been originally designed for C/C++, its operations can also cause undefined behavior [81]. For example, the `nuw` keyword in the `add` operation stands for "No Unsigned Wrap" and means that the result of the addition is undefined for integer overflows. The `store` instruction is, like in C/C++, undefined if the target address is invalid. In Sulong, we implemented these and other instructions as part of the Truffle LLVM IR interpreter.

Truffle and Graal Truffle is a language implementation framework written in Java, on which Sulong's LLVM IR interpreter is based. Truffle reduces the implementation effort of language implementations, by providing a dynamic compilation mechanism for languages implemented on top of it [148]. Language implementers in Truffle write annotated abstract syntax tree (AST) interpreters, in which each operation is implemented in an executable node [153]. Each node computes its result in an `execute()` method by executing its operands, processing their results, and returning it to the parent node. A node typically has to support operations for various data

Listing 2.2: LLVM IR function for Listing 2.1 compiled by Clang and simplified slightly for readability

```
define i64* @create_random_arr(i64) {
  %2 = shl i64 %0, 3
  %3 = call i8* @malloc(i64 %2)
  %4 = bitcast i8* %3 to i64*
  %5 = icmp eq i64 %0, 0
  br i1 %5, label %6, label %7

; <label>:6:                                        ; preds = %7, %1
  ret i64* %4

; <label>:7:                                        ; preds = %1, %7
  %8 = phi i64 [ %12, %7 ], [ 0, %1 ]
  %9 = call i32 @rand()
  %10 = sext i32 %9 to i64
  %11 = getelementptr inbounds i64, i64* %4, i64 %8
  store i64 %10, i64* %11
  %12 = add nuw i64 %8, 1
  %13 = icmp eq i64 %12, %0
  br i1 %13, label %6, label %7
}
```

types and must thus implement several `execute()` methods. To dispatch to the `execute()` method that corresponds to the inputs' data types, the Truffle framework provides a Domain Specific Language based on Java annotations that automatically generates Java code for the dispatch logic [64]. Note that an AST interpreter can only represent high-level control flow; to support the dispatch between the basic blocks of the LLVM IR, we had to come up with a hybrid interpretation approach. (see Chapter 4).

If a function, that is, its AST, has been executed often enough, Truffle compiles it to machine code. In doing so, all the `execute()` methods of the AST are inlined, effectively eliminating the overhead of using an AST interpreter by performing a *Futamura Projection* [42], and then further optimizing the code using Graal. Graal is a dynamic compiler that takes Java bytecode and profiling information as its input, and generates optimized machine code [119, 34, 33, 120, 122]. To support Sulong and other bytecode interpreters (see Chapter 4), the partial evaluation mechanism had to be extended, which was primarily done by Christian Wimmer.

Truffle nodes can implement speculative optimizations by replacing a node with some other node that implements this optimization. For example, we use this technique to speculate on constant function pointers at indirect call sites [103]. If a speculation turns out to be wrong later, the code generated by Graal is discarded, and execution falls back to the interpreter (a mechanism which is known as *deoptimization* [61, 41]). Truffle is similar to the PyPy [110] language implementation framework [86]; however, compilation in Truffle is method-based while PyPy uses a tracing compiler [10], and Truffle language implementations are implemented in Java while PyPy implementations are written in RPython [2].

Sulong's execution modes We built Sulong incrementally by adding various execution modes. The execution modes are implemented as *strategies* [44] that can be exchanged for each other in Sulong (see Figure 2.1). The strategies differ in the kind of Truffle nodes that are created to implement the semantics of LLVM IR operations. We first implemented Native Sulong, which represents user allocations in unmanaged memory, and is thus not safe with respect to memory accesses. Native Sulong was an intermediate step to a safe execution engine, which allowed us to concentrate on efficiency aspects of the implementation. Next, we implemented Safe Sulong, which represents user allocations as Java objects and provides safe semantics also for accessing them. First, we implemented a bug-finding mode that aborts the program when an invalid memory access occurs. As a second safe execution mode, we devised the Lenient C strategy that defines undefined behavior in such a way that illegal operations are executed with a defined semantics according to common expectations of programmers. This allows programs with undefined behavior to keep running.

Native Sulong (Chapter 4) Native Sulong allocates user objects in unmanaged memory. For example, when Native Sulong executes the call to `malloc()` in `create_random_arr()`, it uses a foreign function interface to call the underlying `malloc()` implementation [48], which allocates the object on the native heap. Thus, buffer overflows and other memory errors are not prevented and cause similar effects as in executables that were compiled by static compilers such as Clang or GCC. On the positive side, Native Sulong can use precompiled libraries without restrictions; for example, it can pass objects allocated by Sulong to these libraries because their memory layout conforms to the underlying machine's application binary interface [87]. The operations in Native Sulong were optimized to be efficient for legal input. As a consequence of using Java as an implementation language, however, Native Sulong treats other kinds of undefined behavior in a consistent way, similarly to the Lenient C strategy; for example, both Native Sulong and Lenient C use a Java addition to implement a signed integer addition in C, which provides wraparound semantics on integer overflow. The paper describing Native Sulong (see Chapter 4), was published in the *Workshop on Virtual Machines and Intermediate Languages 2016* [103].

Safe Sulong (Chapter 5) Safe Sulong allocates user objects on the Java heap (i.e., in managed memory). By using managed memory, Safe Sulong detects errors such as buffer overflows relying on automatic runtime checks of the underlying JVM. For example, if an out-of-bounds access occurs when accessing the array in `create_random_arr()`, the access is mapped to an out-of-bounds access to a Java array, where it causes an `ArrayIndexOutOfBoundsException`. The main paper on Safe Sulong (see Chapter 5) was published at the *International Conference on Architectural Support for Programming Languages and Operating Systems 2018* [103].

Listing 2.3: Truffle node that throws an exception on an integer overflow by using `Math.addExact()`

```
class StrictMulNode {

    @Specialization
    long execute(long left, long right) {
        return Math.multiplyExact(left, right);
    }
}
```

This paper constitutes the core contribution of this thesis; it describes a model to safely execute and optimize unsafe languages by executing them on a Java Virtual Machine. Note that we evaluated only memory errors in this paper; however, we also implemented checks for other kinds of undefined behavior. For example, a signed multiplication is implemented in Safe Sulong by a call to `Math.multiplyExact()`. In case of an integer overflow, an `ArithmeticException` would be thrown. When executing `create_random_arr()` passing `LONG_MIN` as a parameter, the multiplication node would already throw an `ArithmeticException` before the out-of-bounds access is executed.

Lenient C (Chapter 6) Safe Sulong fails to execute many programs because they cause undefined behavior and thus violate the rules set by the C standard, which has also been observed by other studies [19, 88]. To support common incorrect program patterns, we devised a Lenient C standard that defines operations for input that is illegal in the C11 standard. For example, Lenient C specifies that addition must overflow with wrap-around semantics, which we implemented by reusing Native Sulong's signed addition node. As another example, Lenient C specifies that `free()` has no effect in order to allow freed memory to be still accessed, which is another frequent programming error; instead, Lenient C assumes that a garbage collector reclaims unused memory. For buffer overflows and `NULL` pointer dereferences, we assumed that the errors were too critical to continue execution and adopted the behavior of Safe Sulong (i.e., to terminate execution). The paper on Lenient C (see Chapter 6) was published in the *International Conference on*

Listing 2.4: The `_size_right()` introspection allows programmers querying the remaining size of the object from the given pointer

```
int *arr = malloc(sizeof(int) * 10);
int *ptr = &(arr[4]);
printf("%ld\n", _size_right(ptr)); // prints 24
```

Managed Languages & Runtimes 2017 [107].

2.2 Introspection

Current run-time bug-finding tools track metadata such as bounds or types in order to detect bugs. We found that it might be useful to provide this metadata also to programmers, so that they can implement additional logic, for example, to implement assertions, validate input parameters, or handle errors that would otherwise cause undefined behavior. We explored this idea in two papers that are included in this thesis.

Introspection (Chapter 7) Based on Safe Sulong, we initially investigated which metadata could be useful for programmers, and then devised an introspection interface that allowed programmers querying it. For example, Listing 2.4 shows how the `_size_right()` introspection function can be used to obtain the remaining size of an object from the given pointer. Other introspection functions allow querying the type and memory location of an object as well as the number and types of variadic arguments. We implemented the introspection interface in Safe Sulong and evaluated possible use cases in a libc implementation. The paper, which is included in Chapter 7, was published in the *The Art, Science, and Engineering of Programming 2018* journal [108].

Context-aware failure-oblivious Computing (Chapter 8) While using our introspection interface, we informally came to the conclusion that the introspection function to query the size of an object would likely be the most useful introspection function since it is straightforward to use and can be used to prevent buffer overflows [20], which are the most common errors in unsafe languages. Thus, based on our introspection work, we devised

Listing 2.5: A strlen() implementation that avoids buffer overflows for strings that are not terminated by `'\0'` by querying the underlying buffer size and using the safer `strnlen()` function

```
size_t strlen(const char *s) {
  return strnlen(s, _size_right(s));
}
```

context-aware failure-oblivious computing, which is a specialized use case of introspection to maintain availability in the presence of buffer overflows. It achieves this by mitigating such errors in libc functions and continuing execution. Listing 2.5 shows how this concept is applied to `strlen()`. `strlen()` queries the end of the memory area allocated for string `s` using `_size_right()` to then compute the length of the string using `strnlen()`, which traverses the string at most until reaching the end of the object, which is specified by the second argument; if a string is passed that is not properly terminated by a `'\0'` character, the function nevertheless stays in bounds and returns the length of the unterminated string.

We demonstrated that our approach mitigates real-world bugs that we found in the Common Vulnerabilities and Exposures (CVE) database.[5] Furthermore, we showed that introspection is applicable also to tools other than Sulong; we implemented the `_size_right()` function for SoftBound+CETS (a bounds checker with a temporal memory error detection tool), GCC's MPX-based bounds checker instrumentation, and in LLVM's AddressSanitizer. The paper, which is included in Chapter 8, was published at the *International Conference on Network and System Security 2018* [106].

2.3 Empirical Studies

We conducted two empirical studies to prioritize the implementation of assembly instructions in inline assembly statements and compiler builtins in Sulong. Neither inline assembly, nor compiler builtins are specified as part of an official C standard; they are compiler extensions. While executing programs "from the wild", we found that a number of programs relied on them. We started to implement them in an ad-hoc fashion, but later decided

[5] `https://cve.mitre.org/`

to investigate their usage in open-source projects to systemize and prioritize their implementation and help other tool developers.

Usage of Inline Assembly (Chapter 9) Inline assembly statements embed assembly into C code. The syntax of such statements is specific to a compiler (e.g., Clang and GCC use a different syntax than the MSVC compiler), and the included instructions are specific to an architecture (e.g., x86). Listing 2.6 shows a C function that uses an inline assembly fragment in Clang/GCC syntax to include the `rdtsc` instruction, which the x86 architecture provides to query the elapsed clock cycles.

Clang parses inline assembly and partially resolves it, which freed us from implementing support for different compiler syntaxes in Sulong. For example, Listing 2.7 shows how LLVM IR represents the inline assembly fragment as a call to the `rdtsc` function, along with a string that encodes constraints (e.g., which registers are used). The example demonstrates that the inline assembly instructions are not mapped to LLVM IR, but are retained as textual statements.

For Sulong, we implemented support only for x86-64 instructions, since we assumed this architecture to be the most common one. On other architectures, projects can be cross-compiled to x86-64 and executed by Sulong; since the assembly instructions are mapped to Java code, they can be executed nevertheless. To support inline assembly, we implemented a parser that reads inline assembly statements from LLVM IR programs to map the individual instructions to Truffle nodes. For `rdtsc`, Listing 2.8 shows how we implemented a Truffle node that reads the elapsed time in milliseconds.[6] This only approximates the expected behavior, since no direct equivalent exists in Java that could be used to read the elapsed clock cycles.

In the study described in Chapter 9, we investigated over 1000 GitHub C projects in their usage of inline assembly. This allowed us to prioritize their implementation, and also to decide how to approximate the behavior of instructions in Java when no equivalent exists there. The paper, which is included in Chapter 9, was published at the *International Conference on Virtual Execution Environments 2018* [105].

[6]The full implementation is available at `https://github.com/graalvm/sulong/blob/master/projects/com.oracle.truffle.llvm.nodes/src/com/oracle/truffle/llvm/nodes/asm/LLVMAMD64RdtscNode.java`

Listing 2.6: Function with an inline assembly fragment to read the elapsed clock cycles on x86-86; `rdtsc` stores the cycles into two 32-bit registers that need to be concatenated.

```
uint64_t rdtsc() {
  uint32_t hi, lo;
  asm("rdtsc" : "=a"(lo), "=d"(hi));
  return lo | ((uint64_t) hi) << 32;
}
```

Listing 2.7: Inline assembly fragment from Listing 2.6 when compiled to LLVM IR.

```
call { i32, i32 } asm "rdtsc",
    "={ax},={dx},~{dirflag},
    ~{fpsr},~{flags}"()
```

Listing 2.8: Truffle node that implements the `rdtsc` instruction

```
class LLVMAMD64RdtscReadNode extends LLVMExpressionNode {
    long doRdtsc() {
        return System.currentTimeMillis();
    }
}
```

Usage of GCC Builtins (Chapter 10) Compiler builtins are specific to a certain compiler and provide functionality that is implemented as part of this compiler. On Linux, GCC builtins are used most frequently and are also supported by compilers such as Clang. In their usage, they are similar to functions provided by libraries (e.g., libc). Listing 2.9 shows an example of the usage of the `__builtin_expect()`, which allows communicating branch probability information to the compiler. The code in the example speculates that `error = 0`, so that the compiler optimizes the code assuming that `common_case()` is called. Clang parses such GCC builtins and typically generates equivalent LLVM intrinsics for them. For example, Listing 2.10 shows the `@llvm.expect.i64` intrinsic call that is generated for `__builtin_expect()`.

In Sulong, we implemented common LLVM intrinsics as specialized nodes. Listing 2.11 shows the node that implements the expect builtin. We incorporate the expected branch probability by creating a Truffle `ConditionProfile`. The profile checks for the expected value, and—if the check holds—code is generated that assumes that the value is always `expected`, which profits subsequent compiler optimizations. However, if the check once fails, the code deoptimizes [155] (i.e., the optimized code is discarded) and the assumption is invalidated until the program terminates; if the code is recompiled, the compiler no longer assumes the value of `expected`.

Listing 2.9: GCC builtin that communicates branch probability information to the compiler.

```
if (__builtin_expect(error, 0)) {
    exceptional_case();
} else {
    common_case();
}
```

Listing 2.10: LLVM intrinsic `@llvm.expect.i64()` that corresponds to the GCC builtin `__builtin_expect()`.

```
call i64 @llvm.expect.i64(
    i64 %3, i64 0)
```

Based on the inline assembly study, we decided to investigate the usage of GCC builtins, which are also frequently used by C projects. We improved the study design by automating the builtin extraction, which was partially done manually in the inline assembly study, and used a larger amount of projects. Furthermore, we also analyzed the usage of GCC builtins over time to find out if their usage was increasing, decreasing, or constant in these projects to predict whether builtins will be important also in programs yet to be written. The paper, which is included in Chapter 10, is currently under review [104].

Listing 2.11: Truffle node that implements the `@llvm.expect.i64()` intrinsic

```
class LLVMExpectI64 extends LLVMBuiltin {

    final ConditionProfile expectProfile =
     ↪  ConditionProfile.createBinaryProfile();

    final long expected;

    public LLVMExpectI64(long expected) {
        this.expected = expected;
    }

    @Specialization
    boolean doI64(long val) {
        if (expectProfile.profile(val == expected)) {
            return expected;
        } else {
            return val;
        }
    }
}
```

Chapter 3

Publications

Below is the full list of publications, classified by paper categories and sorted chronologically. Part II presents the most important publications, which are highlighted with a ☆ below.

3.1 Journal Papers

☆ Introspection for C and its Applications to Library Robustness. **Manuel Rigger**, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, Hanspeter Mössenböck. In *The Art, Science, and Engineering of Programming* (Programming 2018)

3.2 Conference Papers

Fast-Path Loop Unrolling of Non-Counted Loops to Enable Subsequent Compiler Optimizations. David Leopoldseder, Roland Schatz, Lukas Stadler, **Manuel Rigger**, Hanspeter Mössenböck. In *Proceedings of the 15th International Conference on Managed Languages and Runtimes* (ManLang 2018), Linz, Austria, 2018 DOI: 10.1145/3237009.3237013 (AR: 48%)

☆ Preventing Buffer Overflows by Context-aware Failure-oblivious Computing. **Manuel Rigger**, Daniel Pekarek, Hanspeter Mössenböck. In *Proceedings of the 12th International Conference on Network and System Security*

(NSS 2018), Hong Kong, China, 2018 (AR: 39%)

☆ An Analysis of x86-64 Inline Assembly in C Programs. **Manuel Rigger**, Stefan Marr, Stephen Kell, David Leopoldseder, Hanspeter Mössenböck. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (VEE 2018), Williamsburg, VA, USA, 2018, pp. 84–99 DOI: 10.1145/3186411.3186418 (AR: 32%)

☆ Sulong, and Thanks for All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. **Manuel Rigger**, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, Hanspeter Mössenböck. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2018), Williamsburg, VA, USA, 2018, pp. 377–391 DOI: 10.1145/3173162.3173174 (AR: 18%)

☆ Lenient Execution of C on a Java Virtual Machine: Or: How I Learned to Stop Worrying and Run the Code. **Manuel Rigger**, Roland Schatz, Matthias Grimmer, Hanspeter Mössenböck. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes* (ManLang 2017), Prague, Czech Republic, 2017, pp. 35–47 DOI: 10.1145/3132190.3132204 (AR: 45%)

TruffleC: Dynamic Execution of C on a Java Virtual Machine. Matthias Grimmer, **Manuel Rigger**, Roland Schatz, Lukas Stadler, Hanspeter Mössenböck. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (PPPJ 2014), Cracow, Poland, 2014, pp. 17–26 DOI: 10.1145/2647508.2647528 (AR: 42%)

An Efficient Native Function Interface for Java. Matthias Grimmer, **Manuel Rigger**, Lukas Stadler, Roland Schatz, Hanspeter Mössenböck.

In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (PPPJ 2013), Stuttgart, Germany, 2013, pp. 35–44 DOI: 10.1145/2500828.2500832 (AR: 42%)

3.3 Full Workshop Papers

A Cost Model for a Graph-Based Intermediate-Representation in a Dynamic Compiler. David Leopoldseder, Lukas Stadler, **Manuel Rigger**, Thomas Würthinger, Hanspeter Mössenböck. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (accepted for publication)* (VMIL 2018), Boston, Massachusetts, USA, 2018

☆ Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. **Manuel Rigger**, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages* (VMIL 2016), Amsterdam, Netherlands, 2016, pp. 6–15 DOI: 10.1145/2998415.2998416

3.4 Other Publications

Debugging Native Extensions of Dynamic Languages (Tool Paper). Jacob Kreindl, **Manuel Rigger**, Hanspeter Mössenböck. In *Proceedings of the 15th International Conference on Managed Languages and Runtimes* (ManLang 2018), Linz, Austria, 2018 DOI: 10.1145/3237009.3237017 (AR: 48%)

Sulong, and Thanks for All the Fish (Extended Abstract). **Manuel Rigger**, Roland Schatz, Jacob Kreindl, Cristian Häubl, Hanspeter Mössenböck. In *Workshop on Modern Language Runtimes, Ecosystems, and VMs* (MoreVMs 2018), Nice, France, 2018, pp. 35–44 DOI:

10.1145/3191697.3191726

Sandboxed Execution of C and Other Unsafe Languages on the Java Virtual Machine (Extended Abstract). **Manuel Rigger**. In *Student Research Competition at the Intl. Conf. on the Art, Science, and Engineering of Programmings* (Programming SRC 2018), Nice, France, 2018 DOI: 10.1145/3191697.3213795

Sulong: Memory Safe and Efficient Execution of LLVM-Based Languages. **Manuel Rigger**. In *ECOOP 2016 Doctoral Symposium* (ECOOP DS 2016), Rome, Italy, 2016

Sulong - Execution of LLVM-based Languages on the JVM (Position Paper). **Manuel Rigger**, Matthias Grimmer, Hanspeter Mössenböck. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (ICOOOLPS 2016), Rome, Italy, 2016, pp. 7:1–7:4 DOI: 10.1145/3012408.3012416

3.5 Under Review

☆ Understanding GCC Builtins to Develop Better Tools. **Manuel Rigger**, Stefan Marr, Bram Adams, Hanspeter Mössenböck.

On-Stack Replacement in Truffle Interpreters for Non-structured Languages. Raphael Mosaner, **Manuel Rigger**, David Leopoldseder, Roland Schatz, Hanspeter Mössenböck.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[2] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, pages 53–64, New York, NY, USA, 2007. ACM.

[3] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 47–65, New York, NY, USA, 2000. ACM.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018.

[5] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[6] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.

[7] Sandeep Bhatkar and R. Sekar. Data Space Randomization. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[8] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Commun. ACM*, 51(8):83–89, August 2008.

[9] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented Programming: A New Class of Code-reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.

[10] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[11] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[12] Derek Bruening and Qin Zhao. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 213–223, Washington, DC, USA, 2011. IEEE Computer Society.

[13] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow In-

tegrity: Precision, Security, and Performance. *ACM Comput. Surv.*, 50(1):16:1–16:33, April 2017.

[14] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. ACM.

[15] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[16] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[17] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, February 2013.

[18] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[19] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 117–130, New York, NY, USA, 2015. ACM.

[20] C. Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, pages 119–129 vol.2, 2000.

[21] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *In Proceedings of the 10th USENIX Security Symposium*, 2001.

[22] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointguardTM: Protecting Pointers from Buffer Overflow Vulnerabilities. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 7–7, Berkeley, CA, USA, 2003. USENIX Association.

[23] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[24] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant Systems: A Secretless Framework for Security Through Diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[25] Pascal Cuoq, Matthew Flatt, and John Regehr. Proposal for a Friendly Dialect of C. 2014. `https://blog.regehr.org/archives/1180`.

[26] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[27] Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. Efficient and Thread-safe Objects for Dynamically-typed

Languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 642–659, New York, NY, USA, 2016. ACM.

[28] Benoit Daloze, Chris Seaton, Daniele Bonetta, and Hanspeter Mössenböck. Techniques and Applications for Guest-language Safepoints. In *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '15, pages 8:1–8:10, New York, NY, USA, 2015. ACM.

[29] Benoit Daloze, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. Parallelization of Dynamic Languages: Synchronizing Built-in Collections. In *Proceedings of the 2018 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA'18, 2018.

[30] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 555–566, New York, NY, USA, 2015. ACM.

[31] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 103–114, New York, NY, USA, 2008. ACM.

[32] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 155–167, New York, NY, USA, 2003. ACM.

[33] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. Speculation Without Regret: Reducing Deoptimization Meta-data in

the Graal Compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 187–193, New York, NY, USA, 2014. ACM.

[34] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM.

[35] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[36] Josef Eisl, Matthias Grimmer, Doug Simon, Thomas Würthinger, and Hanspeter Mössenböck. Trace-based Register Allocation in a JIT Compiler. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16, pages 14:1–14:11, New York, NY, USA, 2016. ACM.

[37] Josef Eisl, Stefan Marr, Thomas Würthinger, and Hanspeter Mössenböck. Trace Register Allocation Policies: Compile-time vs. Performance Trade-offs. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, ManLang 2017, pages 92–104, New York, NY, USA, 2017. ACM.

[38] M. D. Ernst, G. J. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, Dec 2002.

[39] D. Evans and D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19(1):42–51, Jan 2002.

[40] Matthias Felleisen and Shriram Krishnamurthi. Safety in Programming Languages. Technical report, Rice University, 1999.

[41] S. J. Fink and Feng Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003*, pages 241–252, March 2003.

[42] Yoshihiko Futamura. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher Order Symbol. Comput.*, 12(4):381–391, December 1999.

[43] Swapnil Gaikwad, Andy Nisbet, and Mikel Luján. Performance Analysis for Languages Hosted on the Truffle Framework. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang '18, pages 5:1–5:12, New York, NY, USA, 2018. ACM.

[44] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[45] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20:20–20:27, January 2012.

[46] John Graham-Cumming. Incident report on memory leak caused by Cloudflare parser bug, 2017. `https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/`.

[47] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*, February 2017.

[48] Matthias Grimmer, Manuel Rigger, Roland Schatz, Lukas Stadler, and Hanspeter Mössenböck. TruffleC: Dynamic Execution of C on a Java Virtual Machine. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform:*

Virtual Machines, Languages, and Tools, PPPJ 2014, pages 17–26, New York, NY, USA, 2014. ACM.

[49] Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz, and Hanspeter Mössenböck. An Efficient Native Function Interface for Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ 2013, pages 35–44, New York, NY, USA, 2013. ACM.

[50] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Memory-safe Execution of C on a Java VM. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*, PLAS'15, pages 16–27, New York, NY, USA, 2015. ACM.

[51] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. High-performance Cross-language Interoperability in a Multi-language Runtime. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 78–90, New York, NY, USA, 2015. ACM.

[52] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, pages 1–13, New York, NY, USA, 2015. ACM.

[53] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM.

[54] A. E. Hassan. The road ahead for Mining Software Repositories. In *2008 Frontiers of Software Maintenance*, pages 48–57, Sept 2008.

[55] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[56] Christian Häubl and Hanspeter Mössenböck. Trace-based Compilation for the Java HotSpot Virtual Machine. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 129–138, New York, NY, USA, 2011. ACM.

[57] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Optimized Strings for the Java HotSpot Virtual Machine. In *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, PPPJ '08, pages 105–114, New York, NY, USA, 2008. ACM.

[58] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Evaluation of Trace Inlining Heuristics for Java. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1871–1876, New York, NY, USA, 2012. ACM.

[59] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Trace Transitioning and Exception Handling in a Trace-based JIT Compiler for Java. *ACM Trans. Archit. Code Optim.*, 11(1):6:1–6:26, February 2014.

[60] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag.

[61] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM.

[62] Gerard J. Holzmann. UNO: Static Source Code Checking for UserDefined Properties. In *In 6th World Conf. on Integrated Design and Process Technology, IDPT '02*, 2002.

[63] Petr Hosek and Cristian Cadar. VARAN the Unbelievable: An Efficient N-version Execution Framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming*

Languages and Operating Systems, ASPLOS '15, pages 339–353, New York, NY, USA, 2015. ACM.

[64] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A Domain-specific Language for Building Self-optimizing AST Interpreters. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 123–132, New York, NY, USA, 2014. ACM.

[65] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Compact and Efficient Strings for Java. *Science of Computer Programming*, 75(11):1077 – 1094, 2010. Special Section on the Programming Languages Track at the 23rd ACM Symposium on Applied Computing.

[66] Intel. Intel® Architecture Instruction Set Extensions and Future Features Programming Reference (Revision 34). 2018.

[67] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[68] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.

[69] S. C. Johnson. Lint, a C Program Checker. In *COMP. SCI. TECH. REP*, pages 78–1273, 1978.

[70] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-injection Attacks with Instruction-set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.

[71] Stephen Kell. Dynamically Diagnosing Type Errors in Unsafe Code. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 800–819, New York, NY, USA, 2016. ACM.

[72] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 339–348, Dec 2006.

[73] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, July 1976.

[74] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *CoRR*, abs/1801.01203, 2018.

[75] T. Kotzmann and H. Mossenbock. Run-Time Support for Optimizations Based on Escape Analysis. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 49–60, March 2007.

[76] Thomas Kotzmann and Hanspeter Mössenböck. Escape Analysis in the Context of Dynamic Compilation and Deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 111–120, New York, NY, USA, 2005. ACM.

[77] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008.

[78] Jacob Kreindl, Manuel Rigger, and Hanspeter Mössenböck. Debugging Native Extensions of Dynamic Languages (Tool Paper). In *Proceedings of the 15th International Conference on Managed Languages and Runtimes*, ManLang 2018, 2018.

[79] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated Software Diversity. In *2014 IEEE Symposium on Security and Privacy*, pages 276–291, May 2014.

[80] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[81] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. Taming Undefined Behavior in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 633–647, New York, NY, USA, 2017. ACM.

[82] David Leopoldseder, Roland Schatz, Lukas Stadler, Manuel Rigger, and Hanspeter Mössenböck. Fast-Path Loop Unrolling of Non-Counted Loops to Enable Subsequent Compiler Optimizations. In *Proceedings of the 15th International Conference on Managed Languages and Runtimes*, ManLang 2018, 2018.

[83] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. Dominance-based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 126–137, New York, NY, USA, 2018. ACM.

[84] D. Lockhart, B. Ilbeyi, and C. Batten. Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, March 2015.

[85] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[86] Stefan Marr and Stéphane Ducasse. Tracing vs. Partial Evaluation: Comparing Meta-compilation Approaches for Self-optimizing Interpreters. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 821–839, New York, NY, USA, 2015. ACM.

[87] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface. *AMD64 Architecture Processor Supplement, Draft v0*, 99, 2013.

[88] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the Depths of C: Elaborating the De Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 1–15, New York, NY, USA, 2016. ACM.

[89] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[90] Raphael Mosaner, Manuel Rigger, David Leopoldseder, Roland Schatz, and Hanspeter Mössenböck. On-Stack Replacement in Truffle Interpreters for Non-structured Languages, 2018. Under Review.

[91] Hanspeter Mössenböck and Michael Pfeiffer. Linear Scan Register Allocation in the Context of SSA Form and Register Constraints. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 229–246, London, UK, UK, 2002. Springer-Verlag.

[92] Hanspeter Mössenböck. Adding Static Single Assignment Form and a Graph Coloring Register Allocator to the Java Hotspot Client Compiler. Technical report, 2000.

[93] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *Proceedings of the 39th Annual*

International Symposium on Computer Architecture, ISCA '12, pages 189–200, Washington, DC, USA, 2012. IEEE Computer Society.

[94] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[95] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40, New York, NY, USA, 2010. ACM.

[96] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005.

[97] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[98] Gene Novark and Emery D. Berger. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 573–584, New York, NY, USA, 2010. ACM.

[99] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2):28:1–28:30, June 2018.

[100] Dan Page. A Note On Side-Channels Resulting From Dynamic Compilation. *IACR Cryptology ePrint Archive*, 2006:349, 2006.

[101] Iraklis Psaroudakis, Stefan Kaestle, Matthias Grimmer, Daniel Goodman, Jean-Pierre Lozi, and Tim Harris. Analytics with Smart Arrays: Adaptive and Efficient Language-independent Data. In *Proceedings*

of the Thirteenth EuroSys Conference, EuroSys '18, pages 17:1–17:15, New York, NY, USA, 2018. ACM.

[102] John Regehr. The Problem with Friendly C. 2015. `https://blog.regehr.org/archives/1287`.

[103] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2016, pages 6–15, New York, NY, USA, 2016. ACM.

[104] Manuel Rigger, Stefan Marr, Bram Adams, and Hanspeter Mössenböck. Understanding GCC Builtins to Develop Better Tools, 2019. Under Review.

[105] Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. An Analysis of x86-64 Inline Assembly in C Programs. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2018, pages 84–99, New York, NY, USA, 2018. ACM.

[106] Manuel Rigger, Daniel Pekarek, and Hanspeter Mössenböck. Preventing Buffer Overflows by Context-aware Failure-oblivious Computing. In *Proceedings of the 12th International Conference on Network and System Security*, NSS 2018, 2018.

[107] Manuel Rigger, Roland Schatz, Matthias Grimmer, and Hanspeter Mössenböck. Lenient Execution of C on a Java Virtual Machine: Or: How I Learned to Stop Worrying and Run the Code. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, ManLang 2017, pages 35–47, New York, NY, USA, 2017. ACM.

[108] Manuel Rigger, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. Introspection for C and its Applications to Library Robustness. *The Art, Science, and Engineering of Programming*, (2), 2018.

[109] Manuel Rigger, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. Sulong, and Thanks for All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2018, pages 377–391, New York, NY, USA, 2018. ACM.

[110] Armin Rigo and Samuele Pedroni. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953, New York, NY, USA, 2006. ACM.

[111] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing Server Availability and Security Through Failure-oblivious Computing. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[112] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[113] Julian Seward and Nicholas Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[114] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[115] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-space

Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[116] Ian Sommerville. *Software Engineering*. Pearson, 10th edition, 2015.

[117] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for Security. *IEEE Symposium on Security and Privacy (S&P'19)*. Accepted. To Appear.

[118] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard. Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices. *IEEE Communications Surveys Tutorials*, 20(1):465–488, Firstquarter 2018.

[119] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, and Thomas Würthinger. Compilation Queuing and Graph Caching for Dynamic Compilers. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '12, pages 49–58, New York, NY, USA, 2012. ACM.

[120] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 9:1–9:8, New York, NY, USA, 2013. ACM.

[121] Lukas Stadler, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose. Lazy Continuations for Java Virtual Machines. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 143–152, New York, NY, USA, 2009. ACM.

[122] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 165:165–165:174, New York, NY, USA, 2014. ACM.

[123] Lukas Stadler, Thomas Würthinger, and Christian Wimmer. Efficient Coroutines for the Java Platform. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 20–28, New York, NY, USA, 2010. ACM.

[124] E. Stepanov and K. Serebryany. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55, Feb 2015.

[125] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the Memory Secrecy Assumption. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pages 1–8, New York, NY, USA, 2009. ACM.

[126] C. Sun, V. Le, and Z. Su. Finding and Analyzing Compiler Warning Defects. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 203–213, May 2016.

[127] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward Understanding Compiler Bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 294–305, New York, NY, USA, 2016. ACM.

[128] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[129] PaX Team. Address space layout randomization. 2003.

[130] TIOBE. TIOBE Index for June 2018, 2018. `http://www.tiobe.com/tiobe-index/`.

[131] Jesse A. Tov and Riccardo Pucella. Practical Affine Types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 447–458, New York, NY, USA, 2011. ACM.

[132] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the Expressiveness of Return-into-libc Attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 121–141, Berlin, Heidelberg, 2011. Springer-Verlag.

[133] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A Study of Modern Linux API Usage and Compatibility: What to Support when You'Re Supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 16:1–16:16, New York, NY, USA, 2016. ACM.

[134] Arjan van de Ven and Ingo Molnar. Exec shield. 2004.

[135] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory Errors: The Past, the Present, and the Future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*, RAID'12, pages 86–106, Berlin, Heidelberg, 2012. Springer-Verlag.

[136] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference*, pages 257–267, Dec 2000.

[137] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *NDSS*, pages 3–17, 2000.

[138] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *NDSS*, 2009.

[139] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined Behavior: What Happened to My Code? In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, pages 9:1–9:7, New York, NY, USA, 2012. ACM.

[140] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards Optimization-safe Systems: Analyzing the Impact

of Undefined Behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 260–275, New York, NY, USA, 2013. ACM.

[141] Christian Wimmer and Hanspeter Mössenböck. Optimized Interval Splitting in a Linear Scan Register Allocator. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 132–141, New York, NY, USA, 2005. ACM.

[142] Christian Wimmer and Hanspeter Mössenböck. Automatic Object Colocation Based on Read Barriers. In *Modular Programming Languages*, pages 326–345, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[143] Christian Wimmer and Hanspeter Mössenböck. Automatic Feedback-directed Object Inlining in the Java Hotspot™Virtual Machine. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 12–21, New York, NY, USA, 2007. ACM.

[144] Christian Wimmer and Hanspeter Mössenböck. Automatic Array Inlining in Java Virtual Machines. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 14–23, New York, NY, USA, 2008. ACM.

[145] Christian Wimmer and Hanspeter Mössenbösck. Automatic Feedback-directed Object Fusing. *ACM Trans. Archit. Code Optim.*, 7(2):7:1–7:35, October 2010.

[146] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.

[147] Thomas Würthinger, Danilo Ansaloni, Walter Binder, Christian Wimmer, and Hanspeter Mössenböck. Safe and Atomic Run-time Code

Evolution for Java and Its Application to Dynamic AOP. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 825–844, New York, NY, USA, 2011. ACM.

[148] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 662–676, New York, NY, USA, 2017. ACM.

[149] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array Bounds Check Elimination for the Java HotSpot&Trade; Client Compiler. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, pages 125–133, New York, NY, USA, 2007. ACM.

[150] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Visualization of Program Dependence Graphs. In Laurie Hendren, editor, *Compiler Construction*, pages 193–196, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[151] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array Bounds Check Elimination in the Context of Deoptimization. *Sci. Comput. Program.*, 74(5-6):279–295, March 2009.

[152] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Unrestricted and Safe Dynamic Code Evolution for Java. *Sci. Comput. Program.*, 78(5):481–498, May 2013.

[153] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM.

[154] Yichen Xie, Andy Chou, and Dawson Engler. ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. In *Proceedings of the 9th European Software Engineering Conference Held*

Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-11, pages 327–336, New York, NY, USA, 2003. ACM.

[155] Yudi Zheng, Lubomír Bulej, and Walter Binder. An Empirical Study on Deoptimization in the Graal Compiler. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:30, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[156] Misha Zitser, Richard Lippmann, and Tim Leek. Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 97–106, New York, NY, USA, 2004. ACM.

Part II

Publications

Chapter 4

Native Sulong

This chapter includes the paper that describes and evaluates Native Sulong, which is also the core of Safe Sulong.

Paper: Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2016, pages 6–15, New York, NY, USA, 2016. ACM

Note: Christian Wimmer contributed the implementation of the partial evaluation mechanism in Graal, which is described abstractly as part of this paper.

Bringing Low-Level Languages to the JVM: Efficient Execution of LLVM IR on Truffle

Manuel Rigger
Johannes Kepler University, Austria
manuel.rigger@jku.at

Matthias Grimmer
Johannes Kepler University, Austria
matthias.grimmer@jku.at

Christian Wimmer
Oracle Labs
christian.wimmer@oracle.com

Thomas Würthinger
Oracle Labs
thomas.wuerthinger@oracle.com

Hanspeter Mössenböck
Johannes Kepler University, Austria
hanspeter.moessenboeck@jku.at

Abstract

Although the Java platform has been used as a multi-language platform, most of the low-level languages (such as C, Fortran, and C++) cannot be executed efficiently on the JVM. We propose Sulong, a system that can execute LLVM-based languages on the JVM. By targeting LLVM IR, Sulong is able to execute C, Fortran, and other languages that can be compiled to LLVM IR. Sulong combines LLVM's static optimizations with dynamic compilation to reach a peak performance that is near to the performance achievable with static compilers. For C benchmarks, Sulong's peak runtime performance is on average 1.39× slower (0.79× to 2.45×) compared to the performance of executables compiled by Clang O3. For Fortran benchmarks, Sulong is 2.63× slower (1.43× to 4.96×) than the performance of executables compiled by GCC O3. This low overhead makes Sulong an alternative to Java's native function interfaces. More importantly, it also allows other JVM language implementations to use Sulong for implementing their native interfaces.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - Run-time environments, Code generation, Interpreters, Compilers, Optimization

Keywords LLVM, JVM, Sulong, dynamic compilation

1. Introduction

The Java Virtual Machine (JVM) has been recently used as a platform for not only Java and Scala, but also for dynamic languages including Ruby, Python, and JavaScript (Rose 2009). Having the JVM as a common platform enables cross-language interoperability so that Java code can call functions or methods written in other languages. Language implementation frameworks such as Truffle (Würthinger et al. 2013) feature a mechanism for cross-language interoperability, which allows writing efficient multi-language applications (Grimmer et al. 2015b). However, except from a C implementation (Grimmer et al. 2014, 2015a), there are no efficient Truffle implementations of lower-level languages, e.g., Fortran, C++, and others. To call functions written in such languages, developers have to resort to the Java Native Interface (JNI, Liang 1999) or other native function interfaces. These native function interfaces add runtime overhead since data structures have to be converted or (un)marshalled when transferring data between Java and the target language. Also, language boundaries are compilation boundaries, so a compiler cannot, for example, apply function inlining across languages.[1]

In this paper we present Sulong, a system that enriches the JVM with a variety of new languages by executing LLVM IR on the JVM. Sulong includes a new LLVM IR interpreter, which allows it to execute all languages that have an LLVM IR front end, including C/C++, Fortran, Ada, and Haskell. Developers can use the interpreter as a Java library to execute these languages on the JVM. We implemented the LLVM IR interpreter in Java on top of the Truffle framework (Würthinger et al. 2013), so that Sulong does not only interface with Java but also provides seamless interoperability with other Truffle language implementations (Grimmer et al. 2015b) such as R (Stadler et al. 2016), Ruby (Seaton 2015), and JavaScript (Würthinger et al. 2013). This in-

[1] Stepanian et al. (Stepanian et al. 2005) show that inlining native C code into Java is important and improves performance significantly. However, they convert the native code to the same intermediate language as the JIT compiler uses while we want to directly run low-level code on the JVM.

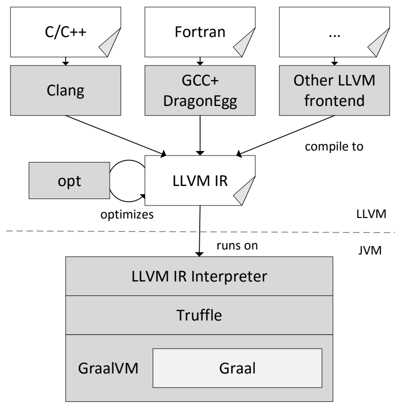

Figure 1. System overview.

teroperability mechanism allows optimizations across language boundaries such as cross-language function inlining. Furthermore, by having Truffle as a common base, other language implementations can use Sulong to implement their Native Function Interfaces (NFIs). For example, JRuby+Truffle (a Truffle implementation of Ruby, Seaton 2015) already uses Sulong to implement C extension support and FastR (a Truffle implementation of R, Stadler et al. 2016) experiments implementing support for native extensions with it.

Efficiency is very important to make Sulong an alternative to NFIs for both Java developers and Truffle language implementers. To achieve the necessary performance, Sulong combines LLVM's static optimizations at compile-time with a dynamic compiler at run-time. We use the Graal dynamic compiler (Duboscq et al. 2013; Stadler et al. 2014) to compile frequently executed LLVM IR functions to native code. This allows Sulong to reach peak performance that is near to the performance of code produced by industrial-strength compilers such as Clang.

In summary, this paper contributes the following:

- We describe how we bring a variety of languages to the JVM by using LLVM front ends and implementing a self-optimizing LLVM IR interpreter.
- We present a novel compilation approach to dynamically compile LLVM IR.
- We describe how we use static optimizations in combination with dynamic compilation to generate efficient machine code and demonstrate its peak performance on a range of C and Fortran benchmarks.

2. System Overview

Sulong is a modularized system that uses parts of LLVM and the JVM (see Figure 1). In this section we describe LLVM and Truffle + Graal, which are the basis of Sulong.

```
void processRequests() {
    int i = 0;
    do {
        processPacket();
        i++;
    } while (i < 10000);
}
```

Figure 2. A small C program containing a loop.

2.1 LLVM

LLVM (Lattner and Adve 2004) is a modular static compilation framework that consists of a standardized IR (called LLVM IR or bitcode) and a set of libraries. LLVM front ends translate a source program to an LLVM IR program. LLVM's official front-end is Clang which can compile C, C++, Objective C, and Objective C++. To enable GCC to compile its supported languages including Ada, Fortran, and Go to LLVM IR, one can use the DragonEgg plugin.[2] After compilation, a user can decide to further process the LLVM IR file, e.g., by using the LLVM static optimization tool *opt* to optimize the program. To get an executable from the LLVM IR file one can use the LLVM linker and assembler to link the LLVM IR files and to compile them to machine code. Sulong consists of a Truffle interpreter that we use to execute this IR on the JVM (see Section 3).

Figure 2 shows a C program, and Figure 3 the corresponding LLVM IR program in textual form. In LLVM IR, as in most IRs, a function comprises basic blocks that consist of sequential instructions and end with a terminator instruction that transfers control to the next basic block. For example, `br label %1` is an unconditional branch to the basic block labeled %1, `br i1 %3, label %1, label %4` is a conditional branch (depending on the boolean value %3) to the basic blocks labelled %1 or %4, and `ret void` is a return from the function. These branches transfer control between basic blocks, similar as in non-structured programming languages that use `goto`. The biggest challenge for Sulong's LLVM IR interpreter is to efficiently interpret and dynamically compile the dispatch between basic blocks.

LLVM IR is in Static Single Assignment form (SSA, Cytron et al. 1991), i.e., each variable is only assigned once. In LLVM IR, these variables are called virtual registers and are prefixed with %. To merge assignments to the same variable after branches, LLVM IR uses phi functions. For example, in `%i.0 = phi i32 [ 0, %0 ], [ %2, %1 ]`, the value assigned to `%i.0` is `0` when the predecessor block is %0 and %2 if the predecessor block is %1. Sulong's LLVM IR interpreter needs to implement these virtual registers of LLVM IR as well as the native memory access that low-level languages use.

[2] `http://dragonegg.llvm.org/`

```
define void @processRequests() #0 {
; (basic block 0)
  br label %1

; <label>:1 (basic block 1)
  %i.0 = phi i32 [ 0, %0 ], [ %2, %1 ]
  call void @processPacket()
  %2 = add nsw i32 %i.0, 1
  %3 = icmp slt i32 %2, 10000
  br i1 %3, label %1, label %4

; <label>:4 (basic block 2)
  ret void
}
```

Figure 3. LLVM IR of the C program in Figure 2.

2.2 Truffle

Truffle (Würthinger et al. 2013) is a language implementation framework to build high-performance Abstract Syntax Tree (AST) interpreters on the JVM. Each node in a Truffle AST has an *execute* method in which it executes its children and returns its own result. Truffle AST interpreters are self-optimizing (Würthinger et al. 2012) in the sense that AST nodes can speculatively rewrite themselves with *specialized* variants at run time, e.g., based on profile information obtained during execution such as type information. For example, our LLVM IR interpreter can optimize indirect function calls by rewriting the indirect call node to a specialized node that speculates on a constant call target and can thus build polymorphic inline caches (Hölzle et al. 1991). In turn, this optimization enables speculative function inlining of indirect calls.

If these speculative assumptions turn out to be wrong, the specialized tree can be reverted to a more generic version that provides functionality for all possible cases. Truffle guest languages use self-optimization via tree rewriting as a general mechanism for dynamically optimizing code at run-time. For example, if an indirect function call is highly polymorphic, Truffle languages rewrite the polymorphic inline cache to a node that performs the lookup and calls the function.

2.3 Graal

When the execution count of a Truffle AST reaches a predefined threshold, Truffle uses the dynamic Graal compiler (Duboscq et al. 2013; Stadler et al. 2014) to compile the AST to machine code. The compiler assumes that the AST is stable and inlines node execution methods of a hot AST into a single method (known as partial evaluation, Futamura 1999) and performs aggressive optimizations over the whole tree. Graal inserts deoptimization points (Hölzle et al. 1992) in the machine code where the speculative assumptions are checked. If they turn out to be wrong, control is transferred back from compiled code to the interpreted AST, where specialized nodes can be reverted to a more generic version.

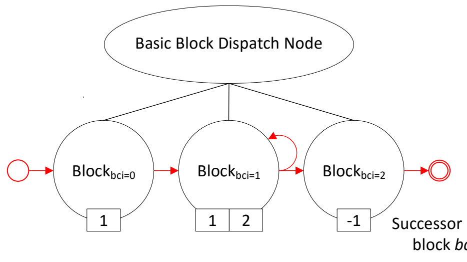

Figure 4. Basic block dispatch node for Figure 3

2.4 Sulong

Sulong uses LLVM front ends to compile source languages such as C/C++ or Fortran to LLVM IR and interprets it on the JVM. To simplify and optimize an LLVM IR program prior to interpretation, Sulong uses LLVM's static optimizers. Sulong then executes the LLVM IR on the JVM using a new Truffle interpreter. This Truffle language implementation brings all LLVM languages to the JVM, and makes them accessible to other Truffle language implementations.

Sulong's interpreter optimizes the AST based on the profile feedback that it observes at run time. Eventually, Truffle uses Graal as a dynamic compiler to compile the program to machine code, from which execution continues with native speeds. This architecture allows Sulong to profit from both static optimizations by LLVM, and dynamic optimizations by Truffle and Graal.

3. Execution of Unstructured Control Flow

The LLVM IR interpreter is different from previous language implementations on top of Truffle since it has to deal with unstructured control flow that cannot easily be handled in an AST interpreter. Support for unstructured control flow is the key for enabling the execution of LLVM IR, both in the interpreter and in the dynamically compiled code.

3.1 Interpreter

To support unstructured control flow in the interpreter we follow a mixed AST execution and bytecode interpretation approach. Basic blocks only contain sequential instructions, hence, we build ASTs for them. We do not build ASTs to implement transferring control between the basic blocks, since unstructured control flow cannot be directly modeled using ASTs. We could convert the unstructured LLVM IR programs to structured programs (Erosa and Hendren 1994), at the expense of making the implementation more complicated and removing the direct correspondence between LLVM IR instructions and Truffle nodes. Instead, we use a basic block dispatch node to transfer control between the basic blocks (and also add support for its compilation, see Section 3.2). Each function has such a basic block dispatch node. In the loop of the basic block dispatch node (see Fig-

```
int bci = 0;
while (bci != -1)
    bci = blocks[bci].execute();
```

Figure 5. Sulong's basic block dispatch node.

ure 5), we execute a basic block in each iteration, starting from a bitcode index of zero (bci = 0). Each node that represents a basic block contains an `int[]` array with the bcis of its successor blocks, which allows the compiler to see all possible successors of a block, i.e., the successor bcis are compile-time constants. The compiler needs this information to compile the basic block dispatch node (see Section 3.2). When executing a basic block, the basic block computes an index into this successor array, which it uses to return the next bci. Execution of basic blocks continues until bci = -1 which signals a return statement.

For the program in Figure 3, the basic block dispatch node transfers execution between three basic blocks that have consecutive indices from 0 to 2. Figure 4 shows the basic block dispatch node for this program and illustrates the control flow between the basic blocks with red arrows. Execution starts with the first basic block $\text{block}_{bci=0}$. $\text{Block}_{bci=0}$ has only one possible successor ($\text{block}_{bci=1}$), therefore its successor array contains only one element, namely bci = 1. The basic block dispatch node executes $\text{block}_{bci=0}$, and reads the next bci = 1 from its successor array. $\text{Block}_{bci=1}$ has two possible successors ($\text{block}_{bci=1}$, the loop body; and $\text{block}_{bci=2}$, the loop exit), therefore the successor array contains two elements, namely bci = 1 and bci = 2. Again, the basic block dispatch node executes $\text{block}_{bci=1}$, and returns either bci = 1 or bci = 2 from its successor array. The successor of $\text{block}_{bci=2}$ is bci = -1, which signals a return from the function.

3.2 Compilation

When compiling an AST, the Graal compiler has to recursively inline the execution methods of all AST nodes. While this is trivial for a regular AST, Graal has to treat the basic block dispatch node differently. For the basic block dispatch node, the compiler unrolls the loop (`while (bci != -1)`, see Figure 5) until all paths through the program are expanded. With respect to the program in Figure 3, the compiler starts with a bci = 0 and determines all successors of $\text{block}_{bci=0}$. The successor of $\text{block}_{bci=0}$ is $\text{block}_{bci=1}$. The compiler can peel the first iteration, and thus moves the execution of $\text{block}_{bci=0}$ out of the loop. Figure 6 illustrates this first step of the loop expansion in pseudo code; note that the first loop iteration (the execution of $\text{block}_{bci=0}$) is peeled. Next, the compiler determines the successors of $\text{block}_{bci=1}$, which are $\text{block}_{bci=1}$ (i.e., the loop body) and $\text{block}_{bci=2}$ (i.e., the loop exit). The compiler detects when a path has already been expanded and merges it with the existing path, which guarantees that the loop expansion terminates. In our

```
  blocks[0].execute();           // bci = 1
  bci = blocks[1].execute();     // to be expanded
```

Figure 6. Step 1: Unrolling the loop of the basic block dispatch node.

```
  blocks[0].execute();           // bci = 1
merge1:
  bci = blocks[1].execute();     // bci = 1 or 2
  if (bci == 1)
      goto merge1;
  else
      bci = blocks[2].execute(); // to be expanded
```

Figure 7. Step 2: Unrolling the loop of the basic block dispatch node.

```
  blocks[0].execute();          // bci = 1
merge1:
  bci = blocks[1].execute();    // bci = 1 or 2
  if (bci == 1)
      goto merge1;
  else
      blocks[2].execute();      // bci = -1
      return;
```

Figure 8. Final state: Unrolled loop of the basic block dispatch node.

example, the compiler sees that it has already expanded $\text{block}_{bci=1}$, and inserts a backjump ($\text{block}_{bci=1}$ has itself as a successor, so the compiler detected a loop). The second successor of $\text{block}_{bci=1}$ is $\text{block}_{bci=2}$, which the compiler expands. Figure 7 shows how the successors of $\text{block}_{bci=1}$ are expanded; note that the compiler inserts a jump (`goto merge1`) if it detects a path that has already been expanded ($\text{block}_{bci=1}$ has itself as a successor). Finally, the compiler expands the successors of $\text{block}_{bci=2}$, of which there are none (indicated by bci = -1). The bci = -1 terminates the loop and the compiler has finished loop unrolling. Figure 8 shows how the successors of $\text{block}_{bci=2}$ are expanded; note that the compiler inserts code to return from the function (`return`) if it detects a path that lets the basic block dispatch loop terminate. The Graal compiler then further optimizes the graph obtained by this partial evaluation.

4. Native Calls and Memory Management

One concern for Sulong is seamless and efficient interoperability with native shared libraries such as the C standard library. Reusing existing code in low-level languages such as C/C++ is commonly done by linking user programs against a shared native library that is present as a machine code binary but not available as source code (e.g. the C standard library). Sulong uses the Graal Native Function Interface (Graal NFI, Grimmer et al. 2013) to call native functions of such a library. When Graal compiles the AST to machine code, the compiled Java code directly (i.e., without overhead) calls the native function.

To be interoperable with native functions, the Graal NFI expects its caller to either pass primitive values (by value) or unmanaged objects such as structs or arrays (by reference). Sulong aligns LLVM IR objects (structs, arrays, and vectors) using the same layout as in executables produced by static compilers. It reads this layout information from the bitcode file. When Sulong calls a native function, this native function can directly operate on allocations provided by Sulong, since they match the platform's Application Binary Interface. Thus, Sulong does not need to marshal or convert objects when calling shared library functions, and can call native functions with zero overhead when compared to native to native calls in executables. Following the object layout of static compilers also allows programmers to not only rely on standard C, but even to run programs that rely on undefined aspects of the memory layout when accessing native memory. This is useful in practice, since many programmers rely on what today's compilers do and not what ISO C specifies (Memarian et al. 2016). Sulong allocates, deallocates, and accesses unmanaged memory using the JDK internal *sun.misc.Unsafe* API.

To execute LLVM IR, Sulong has to support two types of unmanaged memory:

Stack: LLVM IR has an *alloca* instruction to allocate stack memory. To implement stack memory, Sulong allocates a block of memory at the start of the program and assigns its address to a stack pointer. The implementation of the *alloca* instruction then increments this stack pointer to allocate memory on the stack.

Heap: LLVM IR can allocate heap memory using external calls to a library function such as *malloc* from the C standard library. Heap memory allocation is transparent for Sulong and is handled like any other external call to a shared library.

5. Static and Dynamic Optimizations

By default, LLVM front ends such as Clang compile local variables in C/C++ to LLVM IR instructions that allocate the variables on the stack. Once a local variable is needed, it is loaded from memory and assigned to a virtual register. Thus, unoptimized LLVM IR programs have many stack allocations and memory accesses that could be avoided by keeping variables in virtual registers as long as their addresses are not needed and the variables have a primitive type. Storing local variables in memory is especially a problem for Sulong: The Graal compiler does not optimize allocations and accesses to unmanaged memory since Java programs mostly use managed memory. To overcome this shortfall, Sulong uses static LLVM optimizations to reduce the number of allocations and accesses to unmanaged memory. LLVM offers the *mem2reg* optimization which attempts to lift such stack allocations to virtual registers or constants. Sulong applies this optimization to reduce native memory accesses which enables the Graal compiler to produce more efficient machine code. Sulong's LLVM IR interpreter efficiently represents virtual registers (see Section 2.1) as Java objects that Graal can optimize well. In compiled code, virtual registers map to machine registers, or are allocated on the stack.

Besides *mem2reg*, LLVM provides other optimizations that reduce memory accesses such as dead store elimination, promote "by reference" arguments to scalars, and handle loop invariant code motion.

In addition to the static optimizations by LLVM Sulong performs several dynamic optimizations that cannot be performed by classic static compilers. On the Truffle level Sulong performs the following optimizations:

Runtime Inlining: Truffle performs profiling-based inlining during run-time. While we could use LLVM to perform static inlining we defer inlining to the run time since Truffle can exploit profiling feedback such as function call counts that can lead to better inlining decisions.

Dynamic Dead Code Elimination: We profile the probability of basic block successors in our basic block dispatch node. Graal will not compile a basic block that has never been executed and instead inserts a deoptimization point. This effectively results in a dynamic dead-instruction elimination (Butts and Sohi 2002), since Graal only considers those nodes for compilation that have been executed by the Sulong interpreter. Additionally, the successor probability profiling helps Graal during optimization and enables re-ordering of basic blocks based on the frequency of their execution.

Value profiling: We identify run-time-invariant memory values (Calder et al. 1997) by observing if a loaded memory value does not change, and replace such a load node by a node that checks if the value is still the same and returns the cached constant. When Graal compiles the node, it can propagate the profiled constant through constant folding and other optimizations. This optimization is especially beneficial for global variables that are set at the beginning of a program (e.g., configuration values) and do not change afterwards.

Polymorphic inline caches: We construct polymorphic inline caches (Hölzle et al. 1991) for function pointer calls. The first time we indirectly call a function the call site caches the target function up to a certain cache size. Subsequent calls then first check if the current function pointer is one of the cached target functions, and if so, perform a direct call to the function. Guarded direct calling enables Truffle to inline function pointer calls which eliminates the call overhead and enables optimizations on a larger range of code. If the number of cached functions exceeds a predefined threshold, we perform a nor-

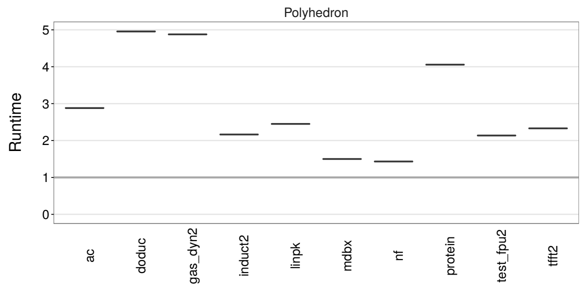

Figure 10. Polyhedron benchmark suite; peak performance (lower is better, relative to *GCC O3*)

mal indirect call since the inlining benefits are not likely to amortize the additional checks.

6. Evaluation

To evaluate Sulong, we choose C and Fortran as two LLVM languages. We do not evaluate C++ since we do not yet support LLVM IR exception handling. We use LLVM's official front end Clang to compile C to LLVM IR. Since Clang cannot compile Fortran, we use GCC with the DragonEgg plugin to compile Fortran to LLVM IR.

6.1 Benchmarks

To evaluate Sulong, we use all single-threaded C benchmarks from the *Computer Language Benchmark game* (shootouts)[3]. The shootouts are small benchmarks (66-453 LOC[4]) designed to compare the performance of different languages. They are useful as a base for the comparison of language implementations, since language implementers commonly use them as an optimization target (Barrett et al. 2016; Marr et al. 2016). We also include the whetstone[5], deltablue[6], and richards[7] benchmarks (239 to 839 LOC) since they are similarly popular small benchmarks for C.

Sulong is still a prototype and in an early stage. It cannot yet execute all SPEC CPU benchmarks. However, we want to also present performance numbers on real world applications. Sulong can already execute an application for compression using bzip2 (5k LOC) and gzip (5K LOC), and an application that converts an audio file using oggenc (48K LOC). These benchmarks are part of the *Large scale compilation-unit C programs* [8].

The same is true when executing Fortan on top of Sulong. Sulong can run 10 benchmarks from the *Polyhedron Benchmark Suite*[9], which in total consists of 17 mixed-size (161 LOC - 27K LOC) benchmarks to evaluate Fortran compiler implementations.

The benchmarks from SPEC CPU and the Polyhedron Benchmark Suite that are not part of our evaluation cannot be executed by Sulong. Sulong either fails parsing their LLVM IR, crashes because of implementation bugs, or reports an unimplemented feature. We are convinced that the implementation of missing features and resolving the known issues is possible with reasonable effort in the future.

6.2 Experimental Setup

To account for the adaptive compilation techniques of Truffle and Graal, we set up a harness that warms up the benchmarks. After the warm-up iterations, every benchmark reaches a steady state such that subsequent iterations are identically and independently distributed. We execute each C benchmark 100 times and use the last 50 iterations to compute the runtime. Since the Fortran benchmarks warm up faster and run longer, we execute them 20 times and use the last 10 iterations to compute the runtime.

We measure the peak performance of C and Fortran code on top of Sulong and then compare it with the performance of executables generated by the static compilers Clang (for C), and GCC (for Fortran). We focus this evaluation on peak performance of long-running applications where the startup performance plays a minor role. Hence, we neglect the startup time and present performance numbers after an initial warm-up.

We executed the benchmarks on a quad-core Intel Core i7-6700HQ CPU at 2.60GHz on Ubuntu version 14.04 (4.3.0-040300rc3-generic) with 16 GB of memory. We use *Sulong* revision *ad56c6f*, which is publicly available at `https://github.com/graalvm/sulong`, that uses LLVM 3.3 (we currently cannot use a newer version due to parser limitations), and the Graal version that will be contained in the GraalVM 0.17 release. When compiling Fortran files to LLVM IR, *Sulong* uses GCC 4.6, the version that is expected to work best with the DragonEgg plugin. When compiling C or Fortran benchmarks for *Sulong* we use the following static optimization parameters to *opt*: *-mem2reg -globalopt -simplifycfg -constprop -instcombine -dse -loop-simplify -reassociate -licm -gvn*. We consider a systematic evaluation of combinations of static and dynamic optimizations on Sulong as future work.

We use *Clang O3* (*-O3* LLVM optimizations) for C, and *GCC O3* (*-O3* GCC optimizations) for Fortran to get a static compilation upper performance boundary. For comparability, *Clang O3* and *GCC O3* use the same LLVM and GCC versions as Sulong. We visualize the peak performance runtime of the benchmarks using box plots. The y-axis shows Sulong's run-time (lower is better) relative to *Clang O3*'s and *GCC O3*'s runtime which is normalized to *1*.

[3] `http://benchmarksgame.alioth.debian.org/`

[4] We used *cloc* to get the lines of code (LOC) without blank lines and comments.

[5] `http://www.netlib.org/benchmark/whetstone.c`

[6] `https://github.com/xxgreg/deltablue/blob/master/deltablue.c`

[7] `http://www.cl.cam.ac.uk/~mr10/Bench.html`

[8] `http://people.csail.mit.edu/smcc/projects/single-file-programs/`

[9] `http://www.polyhedron.com`

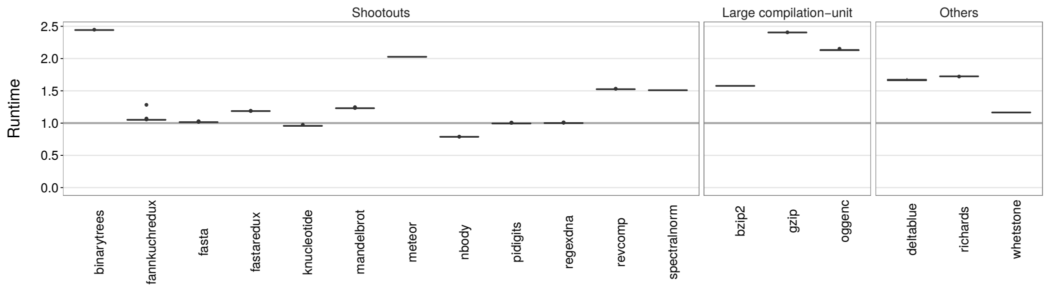

Figure 9. C benchmarks; peak performance (lower is better, relative to *Clang O3*).

6.3 Result

On the C benchmarks (see Figure 9), Sulong's peak performance ranges from being 0.79× faster than Clang (nbody), and being 2.45× slower (binarytrees). On average (geometric mean (Fleming and Wallace 1986)), Sulong is 1.39× slower than Clang. On nbody, Sulong is faster since it can use the SSE sqrt instruction instead of a call to the standard library, and since it can unroll a loop whose number of loop iterations depends on an input parameter to the function. On many benchmarks, Sulong achieves similar performance as Clang O3 (fannkuchredux, fasta, fastaredux, knucleotide, pidigits, regexdna, and whetstone). For most of these benchmarks, Sulong produces similarly efficient code as Clang. However, pidigits and regexdna spend most work in calls to (and in) third-party libraries. Having no overhead on these benchmarks demonstrates that Sulong can efficiently interface with native code. On the remaining C benchmarks (binarytrees, bzip2, deltablue, gzip, meteor, oggenc, revcomp, richards, and spectralnorm), Sulongs performance is between 1.5× and 2.45× slower than Clang O3.

On the Fortran benchmarks (see Figure 10), Sulong's peak performance is between 1.43× (nf) and 4.96× (doduc) slower than the performance of GCC O3 executables. On average, Sulong is 2.63× slower compared to GCC O3. So far, we mainly optimized Sulong for executing C programs, and have not yet looked into optimizing Fortran programs, which explains the larger gap between Sulong and GCC.

Besides missing various micro optimizations, there are three main reasons for the overheads on the C and Fortran benchmarks:

Needless interpreter-level object allocations: Graal implements a partial escape analysis with scalar replacement to optimize or remove object allocations where possible (Stadler et al. 2014). It is critical for performance, that all Java allocations that the LLVM IR interpreter uses in its runtime (i.e., interpreter-level allocations as opposed to user-level allocations) are optimized or removed in compiled code. Unfortunately, we still have situations where this is not the case, and where we either have to adapt data structures in the interpreter or fix problems in Graal's escape analysis.

Truffle's calling convention: Truffle passes function arguments in an Object array and returns the function return value as an Object, so parameters and return values have to be boxed and unboxed. Function inlining usually removes this overhead. However, in benchmarks that stress recursive calls (which can only be inlined up to a certain level) such as binarytrees and richards, the overhead is still significant.

Missing vectorization: Graal cannot produce vectorized code for Sulong, since it does not provide sufficient analyses for accesses to unmanaged memory.

7. Limitations

Sulong can currently execute most small and middle-sized single-threaded C and Fortran programs. We did not concentrate on other languages so far and thus did not implement, for example, LLVM IR exception handling, which is needed to execute C++ programs that use exceptions. Although we did not find any essential problems when executing LLVM IR on the JVM, our current implementations has several limitations:

Unsupported library functions: To achieve better performance and faster startup times, we still use the native (i.e., machine code) standard libraries instead of their bitcode versions. When Sulong is complete and fast enough, we will execute the LLVM IR of the standard libraries with Sulong for which we will only have to substitute system calls. Currently, Sulong does not support creating new processes with `fork`, since a call to `fork` would create a copy of the JVM. Similarly, we currently also do not support `setjmp`/`longjmp`, signal handling, and POSIX `pthreads` for multithreading.

Callbacks from native functions: In terms of native interoperability, our foreign function interface does not support native callbacks yet (Grimmer et al. 2013). For example, we cannot call a native function to which we pass a Truffle AST (e.g., `qsort`) that could be called from the native side. To prevent this case for the standard libraries, we substitute these functions with Java or bitcode equivalents (see above). For third-party libraries we compile such functions to a shared library which we then link.

Manipulation of function return addresses: In Sulong, the memory layout matches that of executables produced by static compilers. One exception is the function return address that executables store in the same stack as data passed to other functions. The Sulong interpreter implicitly uses the Java execution stack when executing functions. This execution stack is different from our data stack that uses unmanaged allocated memory. Thus, we cannot provide support for reading and manipulating function return addresses. However, this also restricts return oriented programming (a security exploit technique, Shacham 2007) since buffer overflows cannot overwrite the return address.

80 bit floats: Most primitive data types in LLVM IR directly map to Java data types. An exception is LLVM IR's 80 bit float type that Clang uses for C's long double data type on the AMD64 architecture. We do not completely support this data type so far due to the implementation effort required to correctly and efficiently implement it using Java primitives.

Inline assembler: Sulong only partially supports inline assembler by constructing a Truffle AST from it and representing the machine registers as Java objects. Still, Sulong cannot execute generated code (such as produced by JITs), for which Sulong would need to interpret the generated machine instructions.

8. Related Work

8.1 Java's Foreign Function Interfaces

Java's standard NFI is JNI (Liang 1999). JNI is a platform independent interface that not only allows calling native functions, but also enables programmers to interact with Java objects and the JVM. However, JNI requires the declaration of *native* Java methods and the implementation of native functions that match a generated header file, which makes JNI complicated to use, especially when a programmer only wants to call native functions. Due to the abstraction overheads, JNI is also slow (Kurzyniec and Sunderam 2001). Previous work showed that the overheads can greatly be reduced by inlining native function calls and by using the same intermediate language for Java and the target low-level language (Stepanian et al. 2005).

An alternative to JNI is Java Native Access[10] (JNA) which is built on top of JNI and provides access to shared native libraries that it dynamically links. Dynamic linking frees the programmer from the burden of writing boilerplate code, but makes calls slower. Efforts to reduce this overhead by generating call stubs using LLVM as a JIT compiler (but still using JNI) can improve performance by 7.84% (Tsai et al. 2013). Besides JNA, also the Java Native Runtime (JNR) is built on top of JNI and provides a user-oriented API to call native functions[11]. Based on the experiences with JNR, a JDK Enhancement Proposal (JEP 191) was drafted that tackles JNI's drawbacks and aims at providing better usability and optimizing calls to native functions (Nutter and Rose 2014). Project Panama, an OpenJDK subproject, works on improving interoperability between the JVM and native functions based on this JEP with the eventual goal to include the changes in the JDK[12].

In our previous work, we introduced the Graal NFI (Grimmer et al. 2013) to call native functions that are dynamically linked. The Graal NFI is fast, since it compiles a call stub to the native function before invoking it the first time, and inlines the call stub when the surrounding Java code is compiled. However, in contrast to JNA and JNR the programmer is responsible for data alignment and handling of unsafe memory, which makes it error-prone and difficult to use (it was designed for native language implementations on top of Truffle). Also, it is only available in the Graal compiler. Jeannie (Hirzel and Grimm 2007) is a language design that allows nesting Java and C code in the same file, which is then compiled down to JNI. Through static checks on syntax and semantics of both languages, it is easy to use and also eliminates writing boilerplate code.

Sulong is an alternative to traditional native function interfaces since it can execute low-level languages directly on the JVM. Sulong does not require writing boilerplate code, and programmers can use Sulong as a Java library to execute native functions. Additionally, Sulong is fast and supports execution of all LLVM languages. However, Sulong requires that the source code of the native function to be called is available. Also, it requires the Graal compiler in order to reach peak performance that is near to the performance of statically compiled code, and to call native functions.

8.2 PyPy

PyPy (Rigo and Pedroni 2006) and its virtual machine construction approach is an alternative to Truffle/Graal's metacompilation approach (Marr and Ducasse 2015). Both approaches strive to provide a reusable base for dynamic language implementations and also provide language interoperability mechanisms (Barrett et al. 2013, 2015; Grimmer et al. 2015b). In both cases, a language implementer can use high-level languages with automatic memory management for implementing a language. While PyPy uses RPython (a semantic subset of Python, Ancona et al. 2007) for the implementation of its interpreters, Truffle uses Java. PyPy language implementations can be any kind of interpreters, while Truffle implementations are implemented as self-optimizing AST interpreters. With Sulong, we showed how a hybrid bytecode/AST interpreter can be implemented in Truffle.

[10] `https://github.com/java-native-access/jna`

[11] `https://github.com/jnr/jnr-ffi`

[12] `http://openjdk.java.net/projects/panama/`

For an efficient implementation, PyPy uses a translation process to transform the RPython interpreter to low-level code for a target environment (Rigo and Pedroni 2006). This translation process first analyzes the interpreter, annotates it with types, and then consecutively transforms it to lower-level operations. For optimal performance, the translation target is a C interpreter that contains a tracing JIT compiler (Bolz et al. 2009). The tracing JIT is not applied to the user program, but to the interpreter running the user program. Similarly, Truffle compiles ASTs (and not traces) that represent the user program to machine code by using Graal as a dynamic compiler. With Sulong's approach, Graal also supports the compilation of bytecode interpreters and hybrid AST/bytecode interpreters.

8.3 Hybrid Compilation Approaches

Dynamo (Bala et al. 2000) is a dynamic optimization system that re-optimizes an already compiled native instruction stream to exploit dynamic optimizations. Like Sulong, Dynamo profits from static optimizations at compile time and profiling information at run time. In contrast to Sulong, Dynamo supports any kind of native instruction stream and not only those languages supported by LLVM. However, due to the low-level information on the machine code level, Dynamo's approach is limited in the optimizations that it can apply. Finally, Dynamo re-compiles traces while Sulong uses Truffle and Graal to compile function ASTs to machine code.

Previous work also includes a fat binary approach (Nuzman et al. 2013), where a program is distributed as an executable that comprises both the native code and the IR of that program. The program starts execution with the native code, which incurs only low start-up and warm-up costs. A runtime manager samples the execution count of the functions and when exceeding a certain threshold, it adds instrumentation to it. Finally, a repurposed Java compiler compiles the IR of that function to optimized machine code, for which it also uses the profiling feedback of the instrumented function. While Sulong has higher start-up and warm-up costs, it does not require a modified toolchain that is needed to produce fat binaries. Sulong can execute unmodified LLVM IR that is produced by language front ends for many languages.

8.4 Other Truffle Implementations

We previously worked on Truffle/C (Grimmer et al. 2014) and ManagedC (Grimmer et al. 2015a) which are Truffle interpreters for C. Similarly to Sulong, Truffle/C uses unmanaged memory for its allocations. ManagedC uses Java allocations instead of unmanaged memory. The C interpreters provide the same dynamic optimizations that Sulong does. In contrast to the C interpreters, Sulong also uses static optimizations by LLVM to optimize the program before executing it with its LLVM IR interpreter. Unlike the C interpreters, Sulong is not restricted to C but can execute a range of different languages by targeting LLVM IR. Also, the C interpreters do not have to efficiently support unstructured control flow since it is only used in exceptional situations, e.g., in exception handling using `goto`. To efficiently execute LLVM IR (which contains no high-level loop constructs), we use a hybrid bytecode/AST interpreter approach.

9. Conclusion and Future Work

In this paper we presented Sulong, a system to execute low-level languages such as C and Fortran on the JVM. By providing a Truffle LLVM IR interpreter, Sulong can execute all languages that can be translated to LLVM IR. By combining static optimizations with dynamic compilation Sulong can achieve peak performance that is near to the performance of code that is produced by industrial-strength compilers such as GCC and Clang. We demonstrated that Sulong currently runs C code with a peak performance that is in average $1.39\times$ slower than code compiled by Clang O3 and Fortran code $2.63\times$ slower compared to code compiled by GCC O3.

Other Truffle implementations can profit by using Sulong to implement their native function interfaces. JRuby+Truffle (a Truffle implementation of Ruby) already uses Sulong for its C extension support, and FastR (a Truffle implementation of R) provides an option to use Sulong instead of JNI for calling native routines. Due to Sulong's low overhead and Truffle's language interoperability mechanism that supports inlining across language boundaries, we expect that we can improve the performance of these languages when calling native code. In future work, we want to demonstrate this on case studies, and also provide a version of Sulong that only uses managed Java memory to guarantee memory safety for the programs it executes (Rigger et al. 2016).

Acknowledgments

We thank the Virtual Machine Research Group at Oracle Labs and the members of the Institute for System Software at the Johannes Kepler University for their support and contributions. We especially thank Roland Schatz for his assistance on performance improvements, Chris Seaton for the implementation of JRuby's C extensions using Sulong, and Mick Jordan for his work on implementing calls to native routines using Sulong. We also thank Edd Barrett, Benoit Daloze, Stefan Marr, and Chris Seaton for their comments which greatly improved the paper. The authors from Johannes Kepler University are funded in part by a research grant from Oracle. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. Rpython: a step towards reconciling dynamically and statically typed oo languages. In *Proceedings of DLS 2007*, pages 53–64, 2007.

V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of PLDI '00*, pages 1–12, 2000.

E. Barrett, C. F. Bolz, and L. Tratt. Unipycation: A case study in cross-language tracing. In *Proceedings of VMIL 2013*, pages 31–40, 2013.

E. Barrett, C. F. Bolz, and L. Tratt. Approaches to interpreter composition. *Computer Languages, Systems & Structures*, 44: 199–217, 2015.

E. Barrett, C. F. Bolz, R. Killick, V. Knight, S. Mount, and L. Tratt. Virtual machine warmup blows hot and cold. *ICOOOLPS*, 2016, 2016.

C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of ICOOOLPS 2009*, pages 18–25, 2009.

J. A. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. *ACM SIGOPS OSR*, 36(5):199–210, 2002.

B. Calder, P. Feller, and A. Eustace. Value profiling. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 259–269, 1997.

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings VMIL '13*, pages 1–10, 2013.

A. M. Erosa and L. J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of Computer Languages*, pages 229–240, 1994.

P. J. Fleming and J. J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, Mar. 1986.

Y. Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

M. Grimmer, M. Rigger, L. Stadler, R. Schatz, and H. Mössenböck. An efficient native function interface for java. In *Proceedings of PPPJ '13*, pages 35–44, 2013.

M. Grimmer, M. Rigger, R. Schatz, L. Stadler, and H. Mössenböck. Trufflec: Dynamic execution of c on a java virtual machine. In *Proceedings of PPPJ '14*, pages 17–26, 2014.

M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, and H. Mössenböck. Memory-safe execution of c on a java vm. In *Proceedings of PLAS'15*, PLAS'15, pages 16–27, 2015a.

M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, and H. Mössenböck. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of DLS 2015*, pages 78–90, 2015b.

M. Hirzel and R. Grimm. Jeannie: Granting java native interface developers their wishes. In *Proceedings of OOPSLA '07*, pages 19–38, 2007.

U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91*, pages 21–38, 1991.

U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *ACM Sigplan Notices*, volume 27, pages 32–43, 1992.

D. Kurzyniec and V. Sunderam. Efficient cooperation between java and native codes–jni performance benchmark. In *PDPTA'01*, 2001.

C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *CGO 2004*, pages 75–86, March 2004.

S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. 1999.

S. Marr and S. Ducasse. Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters. *ACM SIGPLAN Notices*, 50(10):821–839, 2015.

S. Marr, B. Daloze, and H. Mössenböck. Cross-language compiler benchmarking. In *DLS 2016 (to appear)*, 2016.

K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. Watson, and P. Sewell. Into the depths of c: elaborating the de facto standards. In *PLDI 2016*, pages 1–15, 2016.

C. O. Nutter and J. Rose. Jep 191: Foreign function interface, 2014. URL `openjdk.java.net/jeps/191`.

D. Nuzman, R. Eres, S. Dyshel, M. Zalmanovici, and J. Castanos. Jit technology with c/c++: feedback-directed dynamic recompilation for statically compiled languages. *TACO*, 10(4):59, 2013.

M. Rigger, M. Grimmer, and H. Mössenböck. Sulong - execution of llvm-based languages on the jvm. In *ICOOOLPS'16*, 2016.

A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *SPLASH 2006*, pages 944–953, 2006.

J. R. Rose. Bytecodes meet combinators: Invokedynamic on the jvm. In *Proceedings of VMIL '09*, pages 2:1–2:11, 2009.

C. Seaton. *Specialising Dynamic Techniques for Implementing The Ruby Programming Language*. PhD thesis, University of Manchester, 2015.

H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007*, pages 552–561, 2007.

L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of CGO '14*, pages 165–174, 2014.

L. Stadler, A. Welc, C. Humer, and M. Jordan. Optimizing r language execution via aggressive speculation. In *DLS 2016 (to appear)*, 2016.

L. Stepanian, A. D. Brown, A. Kielstra, G. Koblents, and K. Stoodley. Inlining java native calls at runtime. In *Proceedings of VEE '05*, pages 121–131, 2005.

Y.-H. Tsai, I.-W. Wu, I.-C. Liu, and J. J.-J. Shann. Improving performance of jna by using llvm jit compiler. In *ICIS 2013*, pages 483–488, 2013.

T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing ast interpreters. In *Proceedings of DLS '12*, pages 73–82, 2012.

T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. In *Proceedings of Onward! 2013*, pages 187–204, 2013.

Chapter 5

Safe Sulong

This chapter includes the paper that describes Safe Sulong and evaluates its bug-finding capabilities.

Paper: Manuel Rigger, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. Sulong, and Thanks for All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2018, pages 377–391, New York, NY, USA, 2018. ACM

Sulong, and Thanks For All the Bugs

Finding Errors in C Programs by Abstracting from the Native Execution Model

Manuel Rigger
Johannes Kepler University Linz
Austria
manuel.rigger@jku.at

Roland Schatz
Oracle Labs
Austria
roland.schatz@oracle.com

René Mayrhofer
Johannes Kepler University Linz
Austria
rene.mayrhofer@jku.at

Matthias Grimmer
Oracle Labs
Austria
matthias.grimmer@oracle.com

Hanspeter Mössenböck
Johannes Kepler University Linz
Austria
hanspeter.moessenboeck@jku.at

Abstract

In C, memory errors, such as buffer overflows, are among the most dangerous software errors; as we show, they are still on the rise. Current dynamic bug-finding tools that try to detect such errors are based on the low-level execution model of the underlying machine. They insert additional checks in an ad-hoc fashion, which makes them prone to omitting checks for corner cases. To address this, we devised a novel approach to finding bugs during the execution of a program. At the core of this approach is an interpreter written in a high-level language that performs automatic checks (such as bounds, NULL, and type checks). By mapping data structures in C to those of the high-level language, accesses are automatically checked and bugs discovered. We have implemented this approach and show that our tool (called *Safe Sulong*) can find bugs that state-of-the-art tools overlook, such as out-of-bounds accesses to the main function arguments.

CCS Concepts • **Software and its engineering → Dynamic compilers**; **Runtime environments**; *Interpreters*; *Software testing and debugging*; • **Security and privacy →** *Virtualization and security*;

Keywords Sulong; memory errors; bug detection; C

ACM Reference Format:
Manuel Rigger, Roland Schatz, René Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. 2018. Sulong, and Thanks For All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. In *ASPLOS '18: Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3173162.3173174

1 Introduction

C programs are plagued by bugs. In particular, memory errors such as buffer overflows, NULL pointer dereferences, and use-after-free errors cause critical bugs. Unlike higher-level languages, the C standard does not define any checks that could detect such erroneous accesses and then abort the program. If not prevented in the application logic, errors induce *undefined behavior*; in practice, they can corrupt memory, leak sensitive data, change the control flow, or crash the program. In some cases, errors remain undetected because they can cause delayed failures or do not exhibit any visible symptoms. Memory errors in C therefore often result in hard-to-find bugs or enable exploitation by attackers.

To tackle this issue, industry and academia have come up with a plethora of static and dynamic tools for finding bugs in C programs [62, 63, 73]. Static tools perform analyses of source code to detect errors of specific types; they typically rely on necessarily incomplete heuristics and give rise to both false positives and false negatives [11, 17, 27]. In contrast, dynamic tools insert additional checks either as part of the compilation process or at run time, and find errors during program execution. Although they only find errors that occur during a specific run of the program, they are expected to find all errors and not to produce false positives. Both static and dynamic bug-finding tools have been widely successful and have detected numerous bugs in commonly used libraries.

In this paper, we concentrate on dynamic bug-finding tools and demonstrate that state-of-the-art approaches such as LLVM's AddressSanitizer (ASan) [55] and Valgrind [42] miss real-world errors that programmers would expect to be found. We argue that this is due to current approaches not abstracting from the underlying machine's low-level execution model; the lack of source information makes it difficult to find all bugs, and a check can easily be forgotten. Furthermore, the checks are implemented using inexact techniques, which inherently causes these tools to miss errors. Dynamic bug-finding tools are either based on static compilers or employed after compilation. It is known that compiler optimizations at higher optimization levels interfere with bug-finding tools [64]; we show that compilers can also optimize away memory errors even when explicitly compiling without optimizations (i.e., with the `-O0` flag). Finally, bug-finding tools that support interoperability with native code usually provide restricted bug-finding coverage, which gives users a false sense of security. For example, both ASan and Valgrind cannot detect out-of-bounds accesses to the `main()` function's arguments.

In this paper, we present a novel approach to finding bugs at run time and to addressing these issues. We implemented this approach in a tool called *Safe Sulong*, which can detect out-of-bounds accesses, use-after-free errors, invalid free errors, double free errors, `NULL` dereferences, and accesses to non-existent variadic arguments. Our approach abstracts from the underlying machine's execution model, using an execution environment for C that is written in a high-level language. By abstracting pointers and other C data structures and representing them in the high-level language, we can rely on well-defined automatic checks of the high-level language to detect bugs in a C program. While we used Java for our implementation, the approach also works for other languages that check and disallow buffer overflows and `NULL` pointer dereferences. Our approach is exact (i.e., non-heuristic) and can find all errors of a specific category. To reach native speeds, it uses a dynamic compiler that compiles frequently executed functions to machine code. This compiler does not optimize away bugs, since it optimizes code based on safe semantics in the sense of Felleisen & Krishnamurthi [18], where run-time errors in the program must cause run-time exceptions. We do not provide interoperability with pre-compiled native code, because it would undermine our bug-finding capabilities. We assume that all C code (including libraries) is executed with our tool, which makes our approach impractical for programs that use libraries for which no source code is available.

In our evaluation we tried to find bugs in small open-source projects. We detected and fixed 68 errors, 8 of which were not found by ASan and Valgrind. We argue that these bugs are due to the lack of abstraction of current-bug finding tools. Additionally, we conducted a preliminary performance evaluation; our prototype lacks functionality to execute large benchmarks such as the ones of SPEC [23] and browsers. The evaluation demonstrates that Safe Sulong has a higher warm-up cost than current approaches, but a peak performance that is better than of other bug-finding tools.

Overall, this paper provides the following contributions:

- We present an alternative approach to bug-finding that abstracts from the underlying machine.
- We implemented our approach and evaluated its start-up costs, warm-up costs, and peak performance.

2 Background

2.1 Errors in C

To determine which memory errors are relevant in practice and should therefore be detected by bug-finding tools, we performed keyword searches of the Common Vulnerabilities and Exposures (CVE)[1] and the ExploitDB[2] databases. Unlike a previous study of memory errors (up to 2012) [63], we grouped the errors into different bug categories. Note that we concentrated only on memory errors (i.e., dereferencing invalid pointers) and thus did not consider memory leaks, reading from uninitialized memory, and other C errors.[3] Figures 1 and 2 show the results for the period from 2012 to 2017. Note that bug categories with a high number of vulnerabilities were also exploited more often.

Out-of-bounds accesses. The most common and dangerous bug category (as previously shown [9, 54, 63]) consists of out-of-bounds accesses to objects, which are also known as spatial memory safety errors. Not only do such bugs continue to be relevant, they are currently on an all-time high. We define an out-of-bounds access as a buffer overflow when it attempts to access memory past the end of an object, and as a buffer underflow when it accesses memory before the beginning of an object.[4] Bug-finding tools typically differ in whether they can detect out-of-bounds accesses to the stack, heap, or global (static) data and whether they detect read and/or write accesses. For example, Valgrind can only find heap buffer out-of-bounds accesses.

Use-after-free errors. The second-most common bug category comprises use-after-free errors (known as temporal memory errors), where an object allocated by `malloc()`, `calloc()`, or `realloc()` is freed, but then accessed again. Such an access is also known as an invalid access to a *stale* or *dangling pointer*.

NULL dereferences. The third-most important bug category is a `NULL` dereference. Note that this error can be detected during normal execution of a program, where dereferencing a `NULL` pointer results in a trap on most architectures.

Other errors. Since the remaining memory errors are less common, we classified invalid free errors, double free errors, and accesses to non-existent variadic arguments as "other errors". An invalid free error is caused when a pointer to a stack object or to a global object is passed to `free()`, or when the pointer passed points into the middle of an object. Double free errors occur when a heap object is freed twice. Accesses to non-existent variadic arguments happen when the number of passed variadic arguments is smaller than that expected

[1] https://cve.mitre.org/

[2] https://www.exploit-db.com/

[3] We are currently adding support for finding such bugs in Safe Sulong (see Section 6) and will describe them in a future paper.

[4] We do not consider out-of-bound accesses in sub objects (e.g., from one array field member to another), as they are deliberately used in `memcpy`-like patterns.

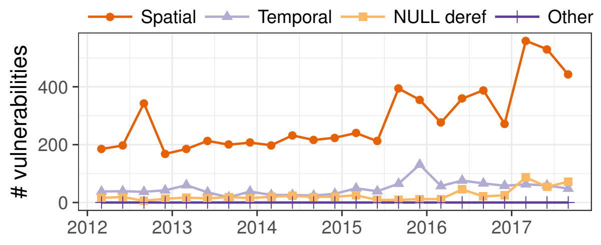

Figure 1. Number of reported vulnerabilities in the CVE database (2012-03 to 2017-09).

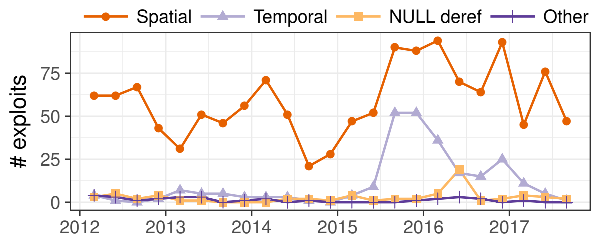

Figure 2. Number of available exploits in the ExploitDB (2012-03 to 2017-09).

by the function. One subclass of this error type are format-string vulnerabilities, where the format string specifies how many arguments on the stack should be accessed.

2.2 State of the Art

Shadow memory. Most practical bug-finding tools such as ASan [55], Mudflap [16], Valgrind [42], Dr. Memory [4], SoftBound+CETS [39, 40], and Purify [22] base their bug-finding capabilities on the concept of shadow memory: They maintain metadata about application memory in a separate memory area referred to as shadow memory, which is used to verify specific actions; for example, read accesses validate that a memory cell is accessible (i.e., allocated memory). Shadow-memory tools are typically combined with red-zone approaches: when a program allocates memory, the runtime of the tool marks the shadow-memory area associated with the program memory as accessible and a region around it as inaccessible (called a *redzone*). Most shadow-memory-based bug-finding tools use this technique to detect out-of-bounds accesses, use-after-free errors, double free errors, invalid free errors and NULL dereferences. Some tools also detect reads of uninitialized memory, use-after-scope errors, and memory leaks. We further discriminate between shadow-memory tools based on whether the instrumentation is added at compile or run time.

Compile-time instrumentation. Compile-time instrumentation involves inserting code for tracking allocations and inserting additional checks when (or before) the program is compiled. The most widely used compile-time instrumentation approach is LLVM's AddressSanitizer, which initially detected out-of-bounds accesses, use-after-free errors, and NULL dereferences [55] and has been extended to detect invalid free, double free, and use-after-scope (including use-after-return as a special case) errors as well as memory leaks. Another state-of-the-art tool that is less used in practice is SoftBound+CETS [39, 40], a bounds checker with a temporal memory safety tool. Mudflap [16] was used by the GCC project until GCC 4.9, when it was superseded by AddressSanitizer. It was known to have several shortcomings, such as reporting false positives and not detecting buffer overflows for neighboring objects in the memory [67]. Commercial tools include Purify [22], which is not strictly a compile-time approach, since it inserts code into object files, and Insure++ [44].

Dynamic instrumentation. Dynamic instrumentation involves inserting checks at the binary level during program execution. The advantages of dynamic instrumentation are that it works for any language that is compiled to machine code, that all code is checked even if the source code and meta information (such as debug information) are not available, and that it does not require recompilation [56]. The most widely used run-time instrumentation approach is Valgrind. Other dynamic instrumentation approaches include Dr. Memory [4] and Intel Inspector [28]. Note that binary instrumentation approaches cannot detect out-of-bounds accesses to the stack (unless the top of the stack is exceeded).

Other approaches. A plethora of other approaches tackle memory errors [62, 63, 73]. For example, *Polymorphic C* [59], *Cyclone* [29], and *CCured* [41] are well-known approaches that provide guarantees against memory errors. However, they require modification of the source code, and are thus not widely used. Canary-based approaches [35] inserts special values (called canaries) next to allocated memory to detect overflows, and inside freed objects to detect writes to freed memory. However, canary approaches can detect only invalid writes, as long as the canary value is not reset again.

2.3 Limitations of Current Approaches

Our main focus is on finding all memory errors in a C program. Current approaches to this have several limitations. We will provide examples for these limitations in our evaluation (see Section 4.1).

Problem 1 (P1): Lack of abstraction from the machine. Current bug-finding tools do not abstract from the low-level execution model of the underlying machine: instead of defining errors at the source level, they define them on the machine level. They insert additional checks either as part of a separate phase in an existing compiler (e.g., ASan or SoftBound) or directly into existing native code (e.g., Valgrind). On this level, the loss of source information makes it conceptually challenging (or impossible) to find all bugs that existed on the source level. Additionally, some approaches need to instrument all read and write operations, all allocations and deallocations, and all system calls [42]. The additional complexity when compared to source-level approaches makes it easy to overlook bugs. A forgotten check cannot easily be found, since in many cases the program behaves as intended, with the only exception that specific errors have gone undetected.

Problem 2 (P2): Compiler optimizations. Current bug-finding tools are built on top of an optimizing compiler such as Clang or GCC. As previously noted [64], this is an issue for bug-finding tools, since they implement C semantics that differ from those of the compiler's optimizer. For example, while bug-finding tools report errors for invalid accesses and abort the program, compilers assume undefined semantics for errors and sometimes optimize them away. It has been shown that compilers are increasingly taking advantage of undefined semantics to optimize code, which leads to more vulnerabilities [14, 65].

Compiler optimizations can lead to *false positives*. For example, a false positive that was found in an ASan-instrumented Firefox build was caused by load-widening [55] where a series of loads is transformed into a single load of several memory values at once while potentially exceeding the bounds of an object. Due to platform-specific alignment requirements, such an optimization can be correct at the system level; however, ASan classified it as a bug because the access would be out of bounds in C. While this issue has been fixed by disabling load-widening [55], such compiler optimizations can still cause false positives in dynamic-instrumentation bug-finding tools (such as Valgrind [56]).

A more serious problem is that compiler optimization can lead to missed errors, that is, *false negatives*. It is widely known that at high optimization levels (e.g., with the `-O2` flag), compilers optimize the code based on the fact that error semantics are undefined [33, 45]. For example, consider the (contrived) function in Figure 3. The function initializes elements of an array without using it further. The array accesses have no visible side effects, so the compiler optimizes the function to immediately return 0. The compiler can exploit the fact that an out-of-bounds access (when $length \geq 10$) has undefined error semantics. Consequently, out-of-bounds accesses that would have occurred in the original code might stay undetected at the binary level, and current bug-finding approaches are unable to find them. It has also been shown that compilers can remove redundant null-pointer checks, even at `-O0` [65]. Further, we have found that Clang can optimize away memory safety errors at `-O0`.

Since the compiler can optimize away bugs (or cause false positives), many projects decide to disable optimizations altogether (with the `-O0` flag) when testing and accept performance degradations. However, as we will demonstrate, explicitly disabling optimizations does not stop compilers from optimizing away bugs.

```c
int test(size_t length) {
    int arr[10] = {0};
    for (size_t i = 0; i < length; i++) {
        arr[i] = i;
    }
    return 0;
}
```

Figure 3. A C program with a potential out-of-bounds access is reduced to `return 0` by optimizing compilers.

Problem 3 (P3): Inexact approaches. Tools based on shadow-memory red-zone approaches typically cannot detect all bugs of a particular category. First, they cannot detect all out-of-bounds accesses. An access to an object running out of bounds and landing inside a different object is not detected as a bug, because the check does not access the redzones next to the original object. Second, shadow-memory approaches cannot reliably detect use-after-free errors. When freeing an object, these approaches mark its shadow memory as unallocated. If the block is quickly reallocated, subsequent uses of the dangling pointer stay undetected, since the memory is again marked as valid. ASan [55] and Purify [22] rely on heuristics to avoid rapid reallocation of freed memory. Note that SoftBound+CETS [39, 40] is not susceptible to such false negatives.

Problem 4 (P4): Finding invalid accesses in libc. Supporting external libraries is a challenge for bug-finding tools. Run-time instrumentation approaches inherently support existing machine code. In contrast, compile-time instrumentation approaches that support native interoperability require heuristics or special treatment of native functions in order to maintain a correct state of the shadow memory.

To achieve higher coverage, compile-time instrumentation approaches recommend creating special instrumented builds for external libraries [55]. This is a challenge in relation to libc, where most production-quality implementations contain non-standard C code (or hand-written assembly) that causes most bug-finding tools (both run-time and compile-time instrumentation approaches) to report errors. Examples are optimized versions of `strlen()` that compute the length of a string by word-wise comparison [66], which can—like the load-widening optimization—lead to out-of-bounds accesses. Current compile-time instrumentation tools disable instrumentation or checks for such functions, or replace them altogether.

The pragmatic alternative that compile-time instrumentation approaches such as ASan and Mudflap provide are so-called *interceptors* that wrap the system library functions and call them only after performing validity checks on the arguments. This approach is dangerous when users expect these interceptors to be comprehensive. As we show in Section 4.1, we found bugs in real-world programs that were not detected

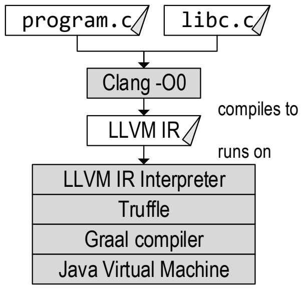

Figure 4. Overview of Safe Sulong.

by ASan due to a missing interceptor. Valgrind and Dr. Memory also provide replacements for these functions, which, however, do not work when these calls have already been inlined at compile time. Thus, Valgrind detects magic constants that point towards a `strlen()` implementation and disables checks for that code block [56].

3 Implementation

We developed Safe Sulong to address the four problems mentioned in Section 2.3. We designed our tool with a focus on bug-finding capabilities. Unlike state-of-the-art approaches that plug into compilers or into native code (see **P1**), Safe Sulong abstracts from the underlying machine and implements a simple execution model; it executes C programs using an interpreter written in Java that relies on automatic checks of the language. The interpreter uses an exact approach (to address **P3**), so no errors are missed. We do not provide interoperability with native code, since this could undermine the bug-finding capabilities (see **P3**). However, we provide our own libc that is written in standard C and is optimized for safety instead of performance (see **P4**). Unlike state-of-the-art approaches that rely on compilers that exploit undefined behavior for compiler optimizations (see **P2**), we use a dynamic compiler that optimizes the code based on safe semantics and cannot optimize away invalid accesses. In summary, our approach allows us to find errors in C programs reliably while still reaching a good peak performance.

3.1 System Overview

Figure 4 shows the architecture of Safe Sulong. It comprises the following components:

Libc. We argued that current libc implementations (which are optimized primarily for performance) are detrimental to bug-finding tools. To address this issue, we implemented a libc that is written in standard C and does not rely on any GNU extensions. It performs additional checks based on run-time information [52]. To implement libc, Safe Sulong exposes functions that are implemented in Java and serve the same purpose as system calls. For example, when printing a pointer value using `printf("%p")`, the `printf()` implementation calls a function implemented in Java to retrieve a textual representation of the pointer. Currently, we support 126 common libc functions, which is sufficient to execute a large body of programs. However, we still lack support for threads and synchronization, interprocess communication, many low-level operations (`mmap()`, `mprotect()`, `setjmp()` and `longjmp()`), and less commonly used functions. As part of future work, we intend to support running an existing libc that chooses standards-conformance and safety over performance (e.g., the `musl libc`) on Safe Sulong. This will require us, for example, to add support for the safe execution of inline assembly that libcs use to implement functionality such as system calls, atomics, and busy-waiting processor hints.

Clang and LLVM IR. Safe Sulong executes LLVM Intermediate Representation (IR), which represents C functions in a simpler, lower-level format. LLVM is a flexible compilation infrastructure [34], and we use LLVM's front end Clang to compile the source code (our libc and the user application) to the IR. Note that we do not enable any of Clang's optimizations to lower the risk that bugs are optimized away. As part of future work, we will replace Clang with a non-optimizing front end, to eliminate this risk completely (see Section 6). Since LLVM IR retains all C characteristics that are important in our context, for simplicity we hereafter refer to LLVM IR objects as C objects. By executing LLVM IR, Safe Sulong could execute languages other than C that can be compiled to this IR, including C++ and Fortran.

Truffle. We used Truffle [68] to implement our LLVM IR interpreter. Truffle is a language implementation framework written in Java. To implement a language, a programmer writes an Abstract Syntax Tree (AST) interpreter in which each operation is implemented as an executable node. Nodes can have children that parent nodes can execute to compute their results.

Graal. Truffle uses Graal [72], a dynamic compiler, to compile frequently executed Truffle ASTs to machine code. Graal applies optimistic optimizations based on assumptions that are later checked in the machine code [15, 60, 61]. If an assumption no longer holds, the compiled code *deoptimizes* [25], that is, control is transferred back to the interpreter, and the machine code of the AST is discarded. Graal optimizes based on safe semantics and cannot introduce false positives or false negatives with respect to the bugs that Safe Sulong finds.

LLVM IR Interpreter. The LLVM IR interpreter (approx. 60k lines of Java code) is at the core of Safe Sulong; it executes both the user application and the enhanced libc. It performs checks while executing the LLVM IR and aborts execution with an error when it detects a bug. First, a front end parses the LLVM IR and constructs a Truffle AST for each LLVM IR function. The interpreter then starts executing the main() function's AST, which can invoke other ASTs. During execution, Graal compiles frequently executed functions to machine code.

JVM. Safe Sulong's interpreter can run on any JVM, since it is written in Java. However, to reach native speeds, it requires a JVM that implements the Java-based JVM compiler interface (JVMCI [53]). JVMCI will be included in OpenJDK 9 by default and enables Graal as a compiler. Note that our tool is platform-independent and provides the same bug-finding capabilities on all platforms. Additionally, Safe Sulong running on a Windows JVM can execute code that was written for libc under Linux.

3.2 Managed Objects and Type Safety

We base the execution model of Safe Sulong on an abstraction of the underlying machine. Our basic idea is to implement our interpreter in Java (i.e., in a high-level language) and represent C data structures as Java data structures. Since Java provides well-specified automatic bounds and type checks, the interpreter automatically checks and detects invalid accesses, such as out-of-bounds accesses, use-after-free errors, and NULL pointer dereferences. As C programs sometimes deliberately contain patterns that violate the C standard [6, 38], we relaxed our type rules (see below). Note that the interpreter could also have been implemented in another high-level language that provides these capabilities.

Figure 5 shows a simplified version of our class hierarchy, which is based on a previous Truffle implementation of C [21]. The base class for all objects is ManagedObject, from which subclasses for all primitives, pointers, functions, arrays, and structs inherit. To represent primitive types, we implemented classes that wrap a Java primitive. For example, to represent an LLVM IR I32 object (which corresponds to a C int on AMD64), we use a Java int, since both have the same bit width. For some data types, no equivalent Java primitive exists; for example, Clang produces LLVM IR code that can contain integers with uncommon bit widths such as I48. We implemented such types using a Java byte array. To represent function pointers, we use a function ID to look up the AST for a function at a function call site. Note that we use inline caches to make function pointer calls efficient [24], and even enable speculative inlining [51]. For arrays, we use Java arrays. For structs, we employ an array-based map-like data structure that is provided by the Truffle framework [21, 69], and contains ManagedObjects. To represent pointers, we implemented an Address class that contains a reference to its pointee and an integer field offset used for pointer arithmetic.

Figure 6 shows an example where malloc() allocates an int array with three elements. Our interpreter maps this allocation to an Address that points to an I32HeapArray which holds a reference to a primitive Java int array (Section 3.3 explains on how we determine the allocation type). The offset in Address is initially 0; when pointer arithmetics compute an address in the middle of an object, the offset is updated. For example, execution of the expression arr[2] first sets the offset to 8, which is computed by multiplying the size of arr's type by 2. When the interpreter executes the load, it takes the offset from Address, divides it by 4 (since the dereferenced object is an int array), and uses the value 2 obtained to index the Java array.

In contrast to approaches that represent C objects as raw bytes that are stored in a large array [36] (e.g., LLJVM[5]), the presented type hierarchy guarantees type safety and restricts invalid pointer casts [10] when the cast pointer is used to read or write from the object. For example, in our architecture an integer array can only hold integer values and no Addresses; storing an Address would require converting it to an int that could be stored in the array. While strict type safety is beneficial to improve program quality and finding bugs, it can prevent real-world programs from executing [6, 31, 38]; for example, we found that many programs rely on invalid type casts to deliberately violate C's type rules. To provide a pragmatic solution, we relaxed the type safety rules to accommodate common patterns that we observed in real-world programs. For example, when the program stores a double in a long array, we simply take the bit representation of the double, convert it to a long, and store it in the array. As part of future work, we plan to further investigate the trade-offs between executing real-world programs and finding bugs.

3.3 Memory allocation

Every allocated object is either a stack object, a heap object, or a global object, that is, automatic, dynamic, or static memory, respectively. We know the type for stack allocations, and can thus directly allocate memory of the specified type in the function prologue. For heap objects (allocated by malloc(), calloc(), or realloc()) we do not know the type of object that will be stored in it. Thus, we allocate the corresponding Java object only on the first cast, read, or write access (i.e., when the type of the object becomes known) and propagate the type back to the allocation site (similar to allocation mementos in V8 [7]). The next time the allocation function is called, we directly allocate an object of the observed type. For global objects, the parser allocates objects at the start of the program.

We have subclasses of each data structure for each storage location. For example, an I32Array has the subclasses I32AutomaticArray, I32HeapArray, and I32StaticArray. Each heap object implements the HeapObject interface, which is used to free objects (see Figure 7). The free() method sets an object's data to null so that the garbage collector can reclaim the memory. Having different classes

[5]https://github.com/davidar/lljvm

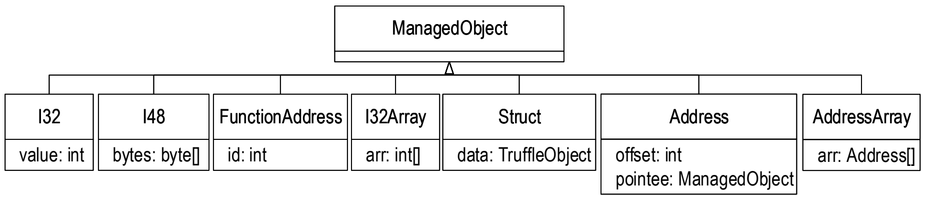

Figure 5. Simplified Class Hierachy of `ManagedObject`.

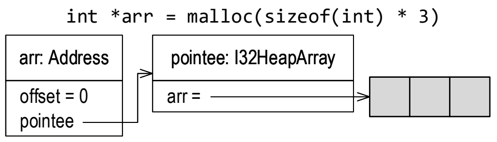

Figure 6. Example of pointer arithmetics and memory allocation (`I32HeapArray` is a subclass of `I32Array`).

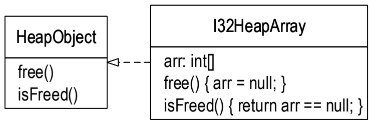

Figure 7. The `HeapObject` interface is used to free heap objects.

```
HeapObject obj =
  (HeapObject) pointer.pointee;
if (pointer.offset != 0) {
  throw new InvalidFreeException();
}
if (obj.isFreed()) {
  throw new DoubleFreeException();
}
obj.free();
```

Figure 8. Implementation of the free method.

for different storage locations also allows us to print meaningful error messages, since we can include the memory type of an object that is illegally accessed or freed.

3.4 Finding bugs

We implemented our bug-finding capabilities by relying on the JVM's automatic checks. In contrast to C, the Java language semantics require that illegal loads, stores, and casts result in an exception. Thus, a JVM cannot simply optimize invalid accesses away.

Out-of-bounds accesses. We translate load and store accesses to arrays in C to array accesses in Java. When the JVM executes the load, it first checks whether the index is in bounds; an out-of-bounds index access results in a Java `ArrayIndexOutOfBoundsException`. Note that such checks reduce the performance of Java programs. To address this, Java compilers such as Graal eliminate checks when they can prove that the index will always be in bounds [71]. As structs do not exist in Java, we represent them using a custom data structure [21, 69], for which we have to perform explicit bounds checks.

Use-after-free accesses. We map C objects that are allocated on the heap to Java objects that have references to the data objects via the `data` field. If an object is freed, the reference to its data is set to `null`. A subsequent access will result in a `NullPointerException`, since Java checks and prevents dereferences of `null`.

Double free errors. As shown in Figure 8, we explicitly check for double free errors in the AST node of the `free()` function using the `isFreed()` method specified by `HeapObject`. This method is implemented by checking whether the data field has already been set to `null`.

Invalid free errors. To detect invalid free errors, Safe Sulong first casts the object to be freed to the `HeapObject` interface. If the object was not allocated on the heap, a `ClassCastException` is thrown, since Java checks every type cast. Therefore, invalid free errors with a wrong pointee are detected. The code verifies next that the pointer offset is zero, that is, an exception is thrown if the pointer does not point to the start of the pointee. Only if the checks succeed is the pointee freed.

Variadic argument errors. Figure 9 shows how we implemented variadic arguments. A call to `va_start()` sets up the processing of variadic arguments by allocating space for a struct that holds a counter and an array of pointers to the variadic arguments. `va_start()` can also initialize this array, since the interpreter exposes the number of variadic arguments via the `count_varargs()` function, which can determine this number because the interpreter passes function arguments as a Java `Object` array that has a field for the array length. We do not require the user to specify the types of the variadic arguments, since we can obtain pointers to them via the `get_vararg()` function. When the user accesses a variadic argument via `va_arg()`, the current variadic argument index is used to access the pointer array. The result is then dereferenced using the user-specified type, which results in a type error for type violations. We can also detect an access to a non-existent variadic argument, as it

```
struct varargs {
  int counter;
  void **args;
};

#define va_list struct varargs *

#define va_start(ap, last)
ap = (va_list)malloc(sizeof(struct varargs));
ap->args = (void **)
    malloc(sizeof(void *) * count_varargs());

for (ap->counter = count_varargs() - 1;
     ap->counter != -1;
     ap->counter--) {
  ap->args[ap->counter] =
    get_vararg(ap->counter);
}

ap->counter = 0;

#define va_arg(ap, type) *((type *)
  (ap->args[ap->counter++]))
```

Figure 9. Implementation of variadic arguments.

would cause an out-of-bounds read of the malloced array. This allows our interpreter to detect the classic format-string problems [8].

4 Evaluation

In our evaluation, we primarily seek to demonstrate the effectiveness of Safe Sulong as a bug-finding tool (Section 4.1). We also show the resource costs of our implementation to argue that our approach is efficient enough to be used in practice. Safe Sulong's run-time performance varies during execution: at the beginning it is poorer than that of other tools (Section 4.2) but it improves when warmed up (Section 4.3). We performed all measurements on a quad-core Intel Core i7-6700HQ CPU at 2.60GHz on Ubuntu version 14.04 (with kernel 4.3.0-040300rc3-generic) with 16 GB of memory.

4.1 Effectiveness

We claimed that Safe Sulong is an effective bug-finding tool. To test this claim, we selected C projects from Github (see below) and executed them with Safe Sulong to find errors in them. We also sought to demonstrate that state-of-the-art approaches fail to detect common real-world bugs that Safe Sulong can detect. To this end, we executed each of the known faulty programs under the same conditions with ASan and Valgrind, that is, with the most popular compile-time and run-time instrumentation approaches, to check whether they could also find the error. Note that SoftBound+CETS [39, 40] was another candidate in the evaluation; however, it is not actively maintained and the version of SoftBound+CETS that is based on a recent LLVM version is still experimental.[6] Consequently, we restricted our evaluation to ASan and Valgrind. Although SoftBound+CETS was formally proven to find all memory-safety violations, we still expect false negatives because it detects memory errors that exist on the machine-level and not on the source-level as it applies LLVM's optimizations both before and after its passes [39, 40]; bugs that exist on the source-level could be removed by the compiler (**P2**).

We primarily selected small programs, ranging from 25 to 4792 (on average 289) lines of code (LOC)[7] because we observed that they were more likely to contain errors than larger projects, as they were often personal "hobby projects" that had not been tested with bug-finding tools. In some cases, this enabled us to find bugs simply by using Safe Sulong to execute the test suite of the project. When the project lacked a test suite, we executed the program, providing both expected input and corner cases. Finding bugs in larger programs would have required us to use automated testing strategies such as fuzzing [20]. Additionally, small programs were unlikely to use library functions that were not yet supported by Safe Sulong (see Section 6). Finally, many small projects relied only on the C standard library and were otherwise self-contained, so we did not have to compile additional dependencies.

In total, we found 68 errors in 63 projects and provided bug fixes for them, many of which were accepted by the project maintainers. Table 1 shows the distribution of the bugs, which roughly follows the distribution of the vulnerability and exploits databases (see Section 2.1). As expected, the majority of bugs were out-of-bounds accesses caused by strings not being `NUL`-terminated, not allocating enough space for a string to hold the `NUL` terminator, missing checks, integer overflows, incorrect hard-coded sizes, performing a check after an invalid access has already happened (see [65]), off-by-one errors in comparisons, and other errors. Table 2 shows that the out-of-bounds accesses included both reads and writes (with almost equal distribution) as well as buffer underflows and overflows. Most out-of-bounds accesses occurred to stack objects, but we also identified several to heap objects, global objects, and to the `main()` function's arguments. A smaller number of bugs were `NULL` dereferences that could also have been found without a bug-finding tool. We found only 1 use-after-free error and 1 variadic argument error (where arguments did not match the format string).

We compiled the programs with Clang using no optimizations (`-O0`), since we aimed to find as many errors as possible. In order to show that compiling with optimizations results

[6]See https://github.com/santoshn/SoftBoundCETS-3.9.

[7]To calculate the LOC, we used `cloc`, which omits comments and empty lines.

Buffer overflows	61
NULL dereferences	5
Use-after-free	1
Varargs	1

Table 1. Error distribution of the detected bugs.

Read	32
Write	29

Underflow	8
Overflow	53

Stack	32
Heap	17
Global	9
Main args	3

Table 2. Distribution of out-of-bounds accesses according to reads/writes, overflows/underflows, and memory kind.

in the failure to detect specific errors, we also compiled the programs at optimization level `-O3` for ASan and Valgrind. We used standard options to execute Valgrind, but after finding out that ASan does not check zero-initialized global data by default, we had to enable the `-fno-common` compiler flag for ASan.

Valgrind `-O0` and `-O3` found slightly more than half of the errors because Valgrind reliably detects only out-of-bounds accesses to the heap and misses many of the out-of-bounds accesses to the stack and to global variables. Note that Valgrind detects reads of uninitialized values, so it could arguably be used to indirectly identify out-of-bounds reads to the stack (14 out of 31 stack accesses). However, we found that this feature is not reliable, and that compiling with either `-O0` or `-O3` reveals different but overlapping sets of bugs.

ASan `-O0` detected 60 of the 68 errors that Safe Sulong found. Only 56 errors (a subset of those found with `-O0`) were also found with `-O3`, since in the other cases Clang optimized away bugs. From the 68 errors that Safe Sulong detected, 8 could neither be found by Valgrind nor by ASan at either optimization level (`-O0` and `-O3`).

1. Uninstrumented main arguments array (P4, P1). We argued that to tools that are based on low-level approaches it is not always obvious whether the programs analyzed contain uninstrumented native code or data. We found that neither ASan [49] nor Valgrind detects out-of-bounds accesses to the `main()` function's arguments—a bug that we discovered in three applications. Figure 10 shows an example: the buffer for `argv` is created before the program (and libc) is invoked and is therefore not instrumented. Note that the `main()` function can have an additional argument for a pointer to an array of environment variables (declared as `int main(int argc, char *argv[], char *envp[])`); this array is initialized irrespective of the `main()` function's signature [37]. A missing or incorrect check might allow an attacker to exploit an out-of-bounds access to leak secrets contained in an environment variable.

```
#include <stdio.h>

int main(int argc, char** argv) {
    printf("%d %s\n", argc, argv[5]);
}
```

Figure 10. ASan does not detect out-of-bounds accesses to the main function.

```
const char t[2] = " \n";
token = strtok(buf, t);
```

Figure 11. The delimiter passed to `strtok()` is not `NUL`-terminated.

```
int counter;
// ...
printf("counter: %ld\n", counter);
```

Figure 12. A wrong format specifier is used, which causes an out-of-bounds read.

2. Missing interceptors (P1). Two bugs could not be found by ASan due to missing or incomplete interceptors; Valgrind did not find them, because the out-of-bounds accesses did not occur in heap-allocated objects. The first bug was caused by an unterminated string that the program passed to the `strtok()` libc function (see Figure 11). ASan failed to detect this bug, as it lacked an interceptor for `strtok()`, which we consequently implemented [47, 48]. We also found one error in which the program passed an integer to `printf("%ld")`, where the format string specified a `long` (see Figure 12). Note that Clang detected the bug statically and printed a warning; however, ASan did not detect the error, because the interceptor for `printf()` checks only pointer arguments.

3. Backend compiler optimizations (P2). In one case, a bug was eliminated by the compiler when compiling with `-O0`, namely a global array out-of-bounds access similar to that shown in Figure 13. Clang statically detected the out-of-bounds access and printed a warning. However, Clang's front end had not yet optimized away the bug, so Safe Sulong was able to detect it while executing the LLVM IR; not until LLVM's back end was it optimized away. Thus, ASan was unable to detect the bug. Valgrind would not have detected the bug in either case, since the array was not allocated on the heap. Arguably, a user could have found the bug given the compiler warning. However, we found cases were Clang `-O0` optimized bugs away even without emitting a warning [50].

4. Overflowing the redzone (P3). As previously shown, shadow-memory red-zone approaches are inexact and cannot find all errors of a particular category. Safe Sulong found such a case in a program that reads a number and converts

```
int count[7] = {0, 0, 0, 0, 0, 0, 0};

int main(int argc, char** args) {
    return count[7];
}
```

Figure 13. The out-of-bounds error in this program is optimized away, even with optimizations disabled (-O0 flag).

```
const char * strings[] = {"zero","one","two","↩
    three","four","five","six" /* ... */ };

void convert(FILE *input, FILE *output) {
    int number;
    fscanf(input, "%d", &number);
    // ...
    fprintf(output, "%s\n", strings[number]);
}
```

Figure 14. A large number as user input causes a buffer overflow that can exceed ASan's redzone.

it to a string; Figure 14 shows a simplified version of the program. In this example, the user input is used to index a global array; if the input number is too large, it causes a buffer overflow. ASan can only find the buffer overflow if the index is close to the object, that is, if it does not exceed the redzone; for our random inputs the access exceeded the redzone and the program either printed (null) or crashed. Valgrind could not find the error, since strings is a global buffer.

5. Missing variadic arguments (P1). In the projects we evaluated, we found only a few implementations of variadic functions. However, we identified a missing argument to the variadic printf() libc function. As in Figure 10, Clang detected the bug statically, since printf() is a well-known libc function. However, the bug could also have occurred in an application-specific function, where Clang would not have been able to detect it; similar format string vulnerabilities have recently been identified in libxml2 (CVE-2016-4448), in Dropbear SSH (CVE-2016-7406), and PHP (CVE-2016-4071). ASan and Valgrind cannot detect such errors at run time.

Discussion. Safe Sulong detected bugs that Valgrind and ASan missed. On the one hand, they missed bugs due to fundamental limitations of their approaches that cannot be addressed without significant enhancements. These stem from the limitations of shadow memory and redzones (4) as well as from their reliance on compilers that optimize based on unsafe semantics (3). On the other hand, they missed bugs due to implementation issues that could be addressed by implementation enhancements or fixes. The uninstrumented main() arguments (1) could be addressed by adding the missing instrumentation. Variadic arguments (5) could

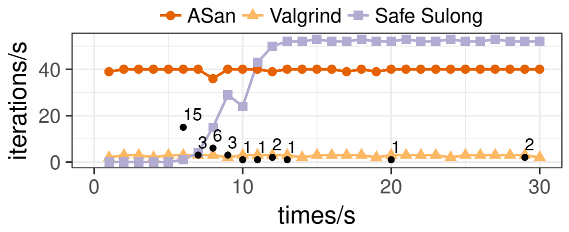

Figure 15. Warm-up time on the meteor benchmark. The x-axis shows the time in seconds and the y-axis the number of iterations an approach could run in the last second. The dots indicate the numbers of ASTs that Graal compiled up to that point.

also be instrumented, passing the argument types supplied at the call site and verifying them in the callee [3]. Missing or incomprehensive interceptors (2) could be addressed by comprehensive implementations or by instrumenting a libc implementation.

We argue that also implementation issues highlight the conceptual advantage of Safe Sulong's abstraction from the native execution model; in Safe Sulong, all accesses are checked automatically, so instrumentation for corner cases cannot be forgotten. Note that state-of-the-art static checkers can detect some of the bugs that were missed by Valgrind and ASan but detected by Safe Sulong; however, they suffer from false positives and other issues [30]. More precise tools such as SoftBound+CETS do not suffer from the limitations of shadow memory and redzones (4); however, SoftBound+CETS applies an unsound optimization where instrumentation for copied memory is omitted [39], which could result in missed bugs.

4.2 Start-up and Warm-up Costs

Safe Sulong uses a dynamic compilation approach and has therefore some additional run-time performance costs.

First, the LLVM IR interpreter has a noticeable start-up cost, which is the time between the user starting the program and the program beginning to run. We measured the start-up time by executing a "Hello, World!" program 100 times for each tool and measuring the execution time using /usr/bin/time. Safe Sulong needs slightly more than 600 ms to start up, during which the JVM initializes and starts Safe Sulong, which must then parse libc before calling the main function. Note that we could improve the start-up performance by lazily parsing libc and by improving the performance of our parser, but this was not the focus of our research. The start-up time of Safe Sulong for this program is longer than that of Valgrind, which needs around 500 ms to instrument and execute the program. With less than 10 ms, ASan starts up the fastest.

Second, the LLVM IR interpreter has a high warm-up cost, which is the time after start-up until the application reaches peak performance. Figure 15 illustrates the warm-up times of ASan, Valgrind, and Safe Sulong on the meteor benchmark. To approximate how Safe Sulong would behave for larger programs (which Safe Sulong currently fails to execute), we continuously executed the benchmark and plotted how many iterations per second the respective approach could execute over time. Safe Sulong's warm-up costs can be attributed mostly to the time the program spends in the interpreter; not until the interpreter has identified hot functions does Graal compile them to machine code. The curve shows a typical VM warm-up [2]. Not until second 6 did Safe Sulong complete its first execution of the benchmark. During this time, Graal had compiled the 15 most important functions to machine code. Afterwards, it soon sped up and executed more iterations per second than Valgrind (in second 7), and ASan (in second 11).

Note that the benchmark contains a call to `printf()` and to other libc functions, which Safe Sulong also interprets and compiles to machine code during execution. Even after compilation, the program fails to immediately reach peak performance, since we currently lack on-stack replacement [1, 19], which is used by production VMs to reduce the warm-up costs by switching from an interpreted method to a compiled method while executing in a loop [19, 26, 32]. However, our peak performance is higher than that of other tools as we demonstrate in Section 4.3. For ASan, we see that compile-time instrumentation approaches incur almost no warm-up costs, because checks are inserted during compilation and the runtime is initialized during start-up. Run-time approaches can insert checks either during start-up or on demand while executing the program. Valgrind inserts them while executing the program; nonetheless, the warm-up costs are not visible and likely overshadowed by the execution time needed for one iteration.

To address start-up and warm-up costs, the Graal project currently explores ahead-of-time compilation for the interpreter and the compiler [70, 72]. Applying this approach to Safe Sulong would allow us to create a standalone tool which would no longer require a JVM, and would have a smaller memory footprint and lower warm-up costs, since the parser and other components would have been compiled before starting the program.

4.3 Peak Performance

In this section we evaluate the peak performance that Safe Sulong can reach on long-running programs. Safe Sulong is a prototype and currently cannot execute large programs such as the SPEC benchmarks. Thus, we decided to evaluate benchmarks from the Computer Language Benchmark game [58], which contains smaller benchmarks for comparing the performance of different programming languages. When we executed this suite's fastaredux benchmark with Safe Sulong, we discovered that a loop ran out of bounds because, due to a rounding error, probabilities did not add up to the value `1.00`. We reported and fixed the bug [46], and used the fixed version in our evaluation. Additionally, we included the whetstone benchmark [43].

We measured the performance of executables compiled by Clang with disabled optimizations (`-O0`) and enabled optimizations (`-O3`) as baselines. Since our aim was to find as many errors as possible, we compiled the benchmarks using `Clang -O0` for all bug-finding tools, although Safe Sulong would also profit from compiler optimizations. In addition to assessing the performance of Safe Sulong, we also measured the performance of executables compiled by Clang 3.9 using ASan based on LLVM version 3.9 and Valgrind version 3.12. A direct comparison of run-time performance between different tools is not fair, since they provide different features. Our measurements are therefore intended to demonstrate that the peak performance of programs under Safe Sulong is sufficient to make our approach viable in practice. To approximate the performance of larger programs, we had to account for the adaptive compilation techniques of Truffle and Graal by setting up a harness that warmed up the benchmarks. By executing 50 in-process warm-up iterations, we ensured that every benchmark reached a steady state. We executed each benchmark 10 times and used the last iteration of each run as a sample for computing the peak performance. We also used the same benchmark harness for the other tools, even though their warm-up costs are minimal.

Figure 16 shows box plots for the peak performance relative to that of `Clang -O0`. We excluded Valgrind from the plots because it was `10`× to `58`× slower than `Clang -O0` on 5 benchmarks. Its slowdown was lowest on `spectralnorm`, `fasta`, and `fannkuchredux` (`2.3`, `3.6` and `5.1`, respectively). We did not plot the results for the `binarytrees` benchmark, since ASan was 14× slower and Valgrind 58× slower than `Clang -O0`. This slowdown was due to `binarytrees` being allocation-intensive, which suggests that current bug-finding approaches cannot deal well with allocation-intensive benchmarks. On this benchmark, Safe Sulong was only 1.7× slower than `Clang -O0`. In almost all benchmarks, Safe Sulong was faster than `ASan -O0`; they were on a par only on `fastaredux`. Safe Sulong was faster than `Clang -O0`, except on the `fastaredux` and `nbody` benchmarks. On `fannkuchredux` and `mandelbrot`, Safe Sulong was even on a par with `Clang -O3`. Safe Sulong exhibited the poorest performance on `fastaredux`, where it was `2.5`× slower than `Clang -O0`. As part of future work, we plan to further reduce Safe Sulong's overhead.

5 Limitations

Native interoperability. Interoperability with pre-compiled binaries is a double-edged sword. It is necessary to execute closed-source libraries [62] and convenient for users, but it results in overlooked bugs, as our findings have demonstrated. What sets Safe Sulong apart from

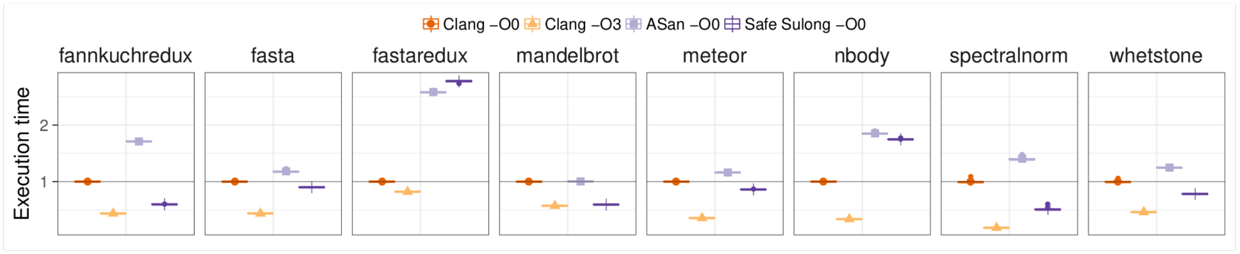

Figure 16. Execution times relative to `Clang -O0` (peak performance, lower is better).

state-of-the-art shadow-memory red-zone approaches (which are inherently inexact) is that our approach is exact and aims to find all errors of a category. To maintain this property, Safe Sulong does not provide a native function interface. This is also the source of Safe Sulong's most significant drawback, as this renders it unusable with programs that require this interoperability. We want to address this issue in the future by using *binary translation* to convert binaries to LLVM IR, which can be executed by Safe Sulong. Translators that can convert binary code to LLVM IR already exist; MC-Semantics [13], REVAMB [12], QEMU [5] support x86, and LLBT [57] support the translation of ARM code.

Warmup time. As discussed in Section 4.2, Safe Sulong needs significantly more time to execute small programs due to warm-up time. Similar to state-of-the-art JVMs, we start by running the program in an interpreter and only compile frequently executed functions to machine code. For approaching the warm-up time of current JVMs, we still lack on-stack replacement, which would allow us to switch to a compiled version of a function while executing in its loop. The Graal project is experimenting with ahead-of-time compilation of the interpreter and the JIT compiler to provide a long-term solution that reduces warm-up time. Note that the JIT compilation approach allows Safe Sulong to reach better peak performance than other bug-finding tools, which could make it applicable to long-running server applications in production.

Programs that rely on non-standard C. Safe Sulong cannot execute all programs that occur "in the wild". We assume that a programmer wants to eliminate undefined behavior from the execution. Consequently, we also require that a program does not violate the type rules of the C standard (e.g., the strict-aliasing rule [10, 31]). Since type violations continue to be relatively common, we relaxed some of the type rules to accommodate real-world code. Additionally, a previous survey discussed certain non-standard-compliant C patterns that are commonly assumed to work [38]. Currently, Safe Sulong lacks support for many such patterns, for example, for tagged pointers where pointers are converted to integers, values stored in spare bits, and converted back to an address. We could implement further relaxations to support such patterns; for instance, we could allow users to store integers in the `offset` field of `Address`.

6 Future Work

Completeness. Safe Sulong can execute most LLVM IR instructions, but is still a prototype that fails to execute larger programs such as the SPEC benchmarks. Such programs require system libraries (most importantly libc) which rely on system calls, inline assembly, compiler builtins, and linker features. As part of future work, we will extend Sulong to support these features, and replace our custom, incomplete libc by an existing, complete one such as musl libc.

Detection of memory leaks. Many bug-finding tools such as Dr. Memory, Valgrind, Purify, and ASan offer support for memory-leak detection. Our approach is based on an exact garbage collector, which reclaims memory when it is no longer needed, irrespective of whether it has been freed or not. We plan to add support for detecting objects that have not been freed by having a background thread that is notified when the garbage collector collects an object (using Java's `PhantomReferences`). When this thread receives a notification, we can check whether the object has been freed manually to print an error in case it has not.

Replace Clang as a front end. As we have demonstrated, Clang (and other C compilers) can optimize away code with undefined behavior even with optimizations disabled. We cannot exclude the possibility that Clang optimized away other bugs that could then no longer be found by ASan, Valgrind, and Safe Sulong. To address this issue, we intend to implement a C front end that does not perform any optimizations.

7 Conclusion

In this paper, we have presented a novel bug-finding tool for C programs that is based on abstraction of the underlying machine. We implemented our approach in a tool called Safe Sulong, which discovered several errors in open-source projects that current bug-finding tools could not find. By using dynamic compilation, Safe Sulong reaches a peak performance that is comparable to that of Clang `-O0`, and even that of Clang `-O3` in some cases.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. We thank Stephen Kell for his insightful comments, and Ingrid Abfalter for proofreading and editorial assistance. We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute of System Software at the Johannes Kepler University Linz for their support and contributions. The authors from Johannes Kepler University are funded in part by a research grant from Oracle.

A Found Bugs

We uploaded a list of found bugs to http://ssw.jku.at/General/Staff/ManuelRigger/ASPLOS18-SafeSulong-Bugs.csv.

References

[1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2000. Adaptive Optimization in the JalapeñO JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM, New York, NY, USA, 47–65. https://doi.org/10.1145/353171.353175

[2] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133876

[3] Priyam Biswas, Alessandro Di Federico, Scott A. Carr, Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, Michael Franz, and Mathias Payer. 2017. Venerable Variadic Vulnerabilities Vanquished. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 186–198.

[4] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 213–223.

[5] Vitaly Chipounov and George Candea. 2010. *Dynamically Translating x86 to LLVM using QEMU*. Technical Report. École polytechnique fédérale de Lausanne.

[6] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. 2015. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 117–130. https://doi.org/10.1145/2694344.2694367

[7] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM '15)*. ACM, New York, NY, USA, 105–117. https://doi.org/10.1145/2754169.2754181

[8] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. 2001. FormatGuard: Automatic Protection from Printf Format String Vulnerabilities. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10 (SSYM'01)*. USENIX Association, Berkeley, CA, USA, 15–15.

[9] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. 2000. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, Vol. 2. IEEE, 119–129.

[10] Pascal Cuoq, Loïc Runarvot, and Alexander Cherepanov. 2017. Detecting Strict Aliasing Violations in the Wild. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 14–33.

[11] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. ACM, New York, NY, USA, 144–157.

[12] Alessandro Di Federico and Giovanni Agosta. 2016. A jump-target identification method for multi-architecture static binary translation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM, 17.

[13] Artem Dinaburg and Andrew Ruef. 2014. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada.*

[14] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The correctness-security gap in compiler optimization. In *Security and Privacy Workshops (SPW), 2015 IEEE*. IEEE, 73–87.

[15] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings VMIL '13*. 1–10.

[16] Frank Ch Eigler. 2003. Mudflap: Pointer Use Checking for C/C+. *Proceedings of the First Annual GCC Developers' Summit* (2003), 57–70.

[17] David Evans and David Larochelle. 2002. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Softw.* 19, 1 (Jan. 2002), 42–51. https://doi.org/10.1109/52.976940

[18] Matthias Felleisen and Shriram Krishnamurthi. 1999. *Safety in Programming Languages*. Technical Report. Rice University.

[19] Stephen J. Fink and Feng Qian. 2003. Design, Implementation and Evaluation of Adaptive Recompilation with On-stack Replacement. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '03)*. 241–252.

[20] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated Whitebox Fuzz Testing.. In *NDSS*, Vol. 8. 151–166.

[21] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Memory-safe Execution of C on a Java VM. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security (PLAS'15)*. ACM, New York, NY, USA, 16–27. https://doi.org/10.1145/2786558.2786565

[22] Reed Hastings and Bob Joyce. 1991. Purify: Fast detection of memory leaks and access errors. In *In proc. of the winter 1992 usenix conference*. Citeseer.

[23] John L Henning. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer* 33, 7 (2000), 28–35.

[24] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)*. Springer-Verlag, London, UK, UK, 21–38.

[25] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. ACM, New York, NY, USA, 32–43. https://doi.org/10.1145/143095.143114

[26] Urs Hölzle and David Ungar. 1994. Optimizing Dynamically-dispatched Calls with Run-time Type Feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 326–336. https://doi.org/10.1145/178243.178478

[27] Gerard J Holzmann. 2002. Static source code checking for user-defined properties. In *Proc. IDPT*, Vol. 2.

[28] Intel. [n. d.]. Intel Inspector 2017. ([n. d.]). https://software.intel.com/en-us/intel-inspector-xe

[29] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In

Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02). USENIX Association, Berkeley, CA, USA, 275–288.

[30] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 672–681.

[31] Stephen Kell. 2016. Dynamically Diagnosing Type Errors in Unsafe Code. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 800–819. https://doi.org/10.1145/2983990.2983998

[32] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot&Trade; Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.* 5, 1, Article 7 (May 2008), 32 pages. https://doi.org/10.1145/1369396.1370017

[33] Chris Lattner. 2011. What Every C Programmer Should Know About Undefined Behavior. (2011). http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html.

[34] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. 75–88. https://doi.org/10.1109/CGO.2004.1281665

[35] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: Fast and Precise Error Detection via Evidence-based Dynamic Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 911–922. https://doi.org/10.1145/2884781.2884784

[36] J. Martin and H. A. Muller. 2001. Strategies for migration from C to Java. In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*. 200–209. https://doi.org/10.1109/.2001.914988

[37] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2013. System V Application Binary Interface. *AMD64 Architecture Processor Supplement, Draft v0* 99 (2013).

[38] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 1–15. https://doi.org/10.1145/2908080.2908081

[39] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

[40] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*. 31–40. https://doi.org/10.1145/1806651.1806657

[41] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.* 27, 3 (2005), 477–526. https://doi.org/10.1145/1065887.1065892

[42] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[43] Painter Engineering. [n. d.]. Whetstone benchmark. ([n. d.]). http://www.netlib.org/benchmark/whetstone.c.

[44] Parasoft. [n. d.]. Insure++. https://www.parasoft.com/product/insure/.

[45] John Regehr. 2010. A Guide to Undefined Behavior in C and C++. (2010). https://blog.regehr.org/archives/213.

[46] Manuel Rigger. 2016. Fix for fasta-redux C gcc #2 program. (2016). https://alioth.debian.org/tracker/?func=detail&atid=413122&aid=315503&group_id=100815.

[47] Manuel Rigger. 2017. Add an interceptor for strtok/rL298650. (2017). https://reviews.llvm.org/rL298650.

[48] Manuel Rigger. 2017. Asan does not detect ouf-of-bounds read in strtok. (2017). https://github.com/google/sanitizers/issues/766.

[49] Manuel Rigger. 2017. Asan does not detect out-of-bounds accesses to argv #762. (2017). https://github.com/google/sanitizers/issues/762.

[50] Manuel Rigger. 2017. Clang -O0 performs optimizations that undermine dynamic bug-finding tools (LLVM Developers Mailing List). (2017). http://lists.llvm.org/pipermail/llvm-dev/2017-March/111371.html.

[51] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL 2016)*. ACM, New York, NY, USA, 6–15. https://doi.org/10.1145/2998415.2998416

[52] Manuel Rigger, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. 2018. Introspection for C and its Applications to Library Robustness. *The Art, Science, and Engineering of Programming* 2 (2018).

[53] John Rose. 2014. JEP 243: Java-Level JVM Compiler Interface. (2014). http://openjdk.java.net/jeps/243.

[54] SANS. 2011. CWE/SANS TOP 25 Most Dangerous Software Errors. (2011). https://www.sans.org/top25-software-errors/.

[55] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*. 309–318.

[56] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *USENIX Annual Technical Conference, General Track*. 17–30.

[57] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. 2012. LLBT: an LLVM-based static binary translator. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 51–60.

[58] shootouts. [n. d.]. The Computer Language Benchmarks Game. ([n. d.]). http://benchmarksgame.alioth.debian.org/.

[59] Geoffrey Smith and Dennis M. Volpano. 1998. A Sound Polymorphic Type System for a Dialect of C. *Sci. Comput. Program.* 32, 1-3 (1998), 49–72. https://doi.org/10.1016/S0167-6423(97)00030-0

[60] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. 2013. An experimental study of the influence of dynamic compiler optimizations on Scala performance. In *Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013*. 9:1–9:8. https://doi.org/10.1145/2489837.2489846

[61] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*. 165. https://doi.org/10.1145/2544137.2544157

[62] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. 48–62. https://doi.org/10.1109/SP.2013.13

[63] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses (RAID'12)*. 86–106. https:

//doi.org/10.1007/978-3-642-33338-5_5

[64] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined behavior: what happened to my code?. In *Asia-Pacific Workshop on Systems, APSys '12, Seoul, Republic of Korea, July 23-24, 2012*. 9. https://doi.org/10.1145/2349896.2349905

[65] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. 260–275. https://doi.org/10.1145/2517349.2522728

[66] Henry S Warren. 2013. *Hacker's delight*. Pearson Education.

[67] GCC Wiki. 2014. Mudflap Pointer Debugging. (2014). https://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging].

[68] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. 13–14. https://doi.org/10.1145/2384716.2384723

[69] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. An object storage model for the truffle language implementation framework. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*. 133–144. https://doi.org/10.1145/2647508.2647517

[70] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 662–676. https://doi.org/10.1145/3062341.3062381

[71] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. 2007. Array bounds check elimination for the Java HotSpot client compiler. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, PPPJ 2007, Lisboa, Portugal, September 5-7, 2007*. 125–133. https://doi.org/10.1145/1294325.1294343

[72] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[73] Yves Younan, Wouter Joosen, and Frank Piessens. 2012. Runtime Countermeasures for Code Injection Attacks Against C and C++ Programs. *ACM Comput. Surv.* 44, 3, Article 17 (June 2012), 28 pages. https://doi.org/10.1145/2187671.2187679

Chapter 6

Lenient C

This chapter includes the paper on Lenient C, which assigns semantics to otherwise undefined behavior, which is implemented as an execution mode in Safe Sulong.

Paper: Manuel Rigger, Roland Schatz, Matthias Grimmer, and Hanspeter Mössenböck. Lenient Execution of C on a Java Virtual Machine: Or: How I Learned to Stop Worrying and Run the Code. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, ManLang 2017, pages 35–47, New York, NY, USA, 2017. ACM

Lenient Execution of C on a Java Virtual Machine

or: How I Learned to Stop Worrying and Run the Code

Manuel Rigger
Johannes Kepler University Linz, Austria
manuel.rigger@jku.at

Roland Schatz
Oracle Labs, Austria
roland.schatz@oracle.com

Matthias Grimmer
Oracle Labs, Austria
matthias.grimmer@oracle.com

Hanspeter Mössenböck
Johannes Kepler University Linz, Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

Most C programs do not conform strictly to the C standard, and often show *undefined behaviors*, for instance, in the case of signed integer overflow. When compiled by non-optimizing compilers, such programs often behave as the programmer intended. However, optimizing compilers may exploit undefined semantics to achieve more aggressive optimizations, possibly breaking the code in the process. Analysis tools can help to find and fix such issues. Alternatively, a C dialect could be defined in which clear semantics are specified for frequently occurring program patterns with otherwise undefined behaviors. In this paper, we present *Lenient C*, a C dialect that specifies semantics for behaviors left open for interpretation in the standard. Specifying additional semantics enables programmers to make safe use of otherwise undefined patterns. We demonstrate how we implemented the dialect in *Safe Sulong*, a C interpreter with a dynamic compiler that runs on the JVM.

CCS CONCEPTS

• **Software and its engineering → Virtual machines**; **Imperative languages**; **Interpreters**; *Translator writing systems and compiler generators*;

KEYWORDS

C, Undefined Behavior, Sulong

ACM Reference format:
Manuel Rigger, Roland Schatz, Matthias Grimmer, and Hanspeter Mössenböck. 2017. Lenient Execution of C on a Java Virtual Machine. In *Proceedings of ManLang 2017, Prague, Czech Republic, September 27–29, 2017,* 13 pages.
https://doi.org/10.1145/3132190.3132204

C is a language that leaves many semantic details open. For example, it does not define what should happen in the case of an out-of-bounds access to an array, when a signed integer overflow occurs, or when a type rule is violated. In such cases, not only does the invalid operation yield an undefined result, but — according to the C standard — the whole program is rendered invalid. As compilers become more powerful, an increasing number of programs break because *undefined behavior* allows more aggressive optimization and may lead to machine code that does not behave as expected. Consequently, programs that rely on undefined behavior may introduce bugs that are hard to find, can result in security vulnerabilities, or remain as time bombs in the code that explode after compiler updates [34, 51, 52].

ManLang 2017, September 27–29, 2017, Prague, Czech Republic

While bug-finding tools help programmers to find and eliminate undefined behavior in C programs, the majority of C programs will still contain at least some non-portable code. This includes unspecified and implementation-defined patterns, which do not render the whole program invalid, but can cause unexpected results. Specifying a more lenient C dialect that better suits programmers' needs and addresses common programming mistakes has been suggested to remedy this [2, 8, 13]. Such a dialect would extend the C standard and assign semantics to otherwise non-portable behavior; it would make C safe in the sense of Felleisen & Krishnamurthi [15]. We devised *Lenient C*, a C dialect which, for example,

- assumes allocated memory to be initialized,
- assumes automatic memory management,
- allows dereferencing pointers using an incorrect type,
- defines corner cases of arithmetic operators,
- and allows pointers to different objects to be compared.

Every C program is also a Lenient C program. However, although Lenient C programs are source-compatible with C programs, they are not guaranteed to work correctly when compiled by C compilers.

We implemented Lenient C in *Safe Sulong* [37], an interpreter with a dynamic compiler that executes C code on the JVM. Per default, Safe Sulong aborts execution when it detects undefined behavior. As part of this work, we added an option to assume the Lenient C dialect when executing a program to support execution of incompliant C programs. Implementing Lenient C in Safe Sulong allowed us to validate the approach without having to change a static compiler. Although a managed runtime is not a typical environment for running C, it is a good experimentation platform because such runtimes typically execute memory-safe high-level languages that provide many features that we also want for C, for example, automatic garbage collection and zero-initialized memory. In this context, Lenient C is a dialect that is suited to execution on the JVM, .NET, or a VM written in RPython [39]. If Lenient C turns out to be useful in managed runtimes, a subset of its rules might also be incorporated into static compilers.

We assume that implementations of Lenient C in managed runtimes represent C objects (primitives, structs, arrays, etc.) using

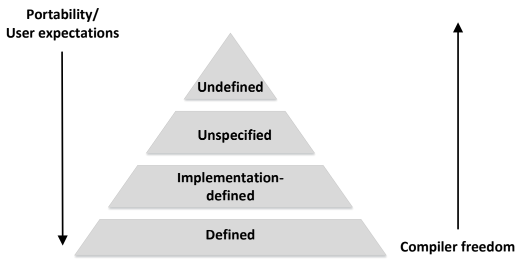

Figure 1: The Pyramid of "Undefinedness"

an object hierarchy, and that pointers to other objects are implemented using managed references. This approach enables use of a GC, which would not be possible if a large byte array were used to represent C allocations [25]. In terms of language semantics, we focused on implementing operations in a way most programmers would expect. Undefined corner cases in arithmetic operations behave similarly to Java's arithmetic operations, which also resemble the behavior of AMD64. This paper makes the following contributions:

- a relaxed C dialect called Lenient C that gives semantics to undefined behavior and is suitable for execution on a JVM and other managed runtimes;
- an implementation of this dialect in Safe Sulong — an interpreter written in Java;
- a comparison of Lenient C with the Friendly C proposal and the anti-patterns listed in the *SEI CERT C Coding Standard*.

1 BACKGROUND

1.1 Looseness in the C Standard

The main focus of C is on performance, so the C standard defines only the language's core functionality while leaving many corner cases undefined (to different degrees, see below). For example, unlike higher-level-languages, such as Java and C#, C does not require local variables to be initialized, and reading from uninitialized variables can yield undefined behavior [43].[1] Avoiding storage initialization results in speed-ups of a few percent [27]. As another example, 32-bit shifts are implemented differently across CPUs; the shift amount is truncated to 5 bits and 6 bits on X86 and PowerPC architectures, respectively [22]. In C, shifting an integer by a shift amount greater than the bit width of the integer type is undefined, which allows the CPU's shift instructions to be used directly on both platforms.

The C standard provides different degrees of looseness, as illustrated by the pyramid of "undefinedness" in Figure 1. Programmers usually want their programs to be *strictly conforming*; that is, they only rely on *defined* semantics. Strictly-conforming programs exhibit identical behavior across platforms and compilers (C11 4 §5). Layers above "defined" incrementally provide freedom to compilers, which limits program portability and results in compiled code that often does not behave as the user expected [51, 52]. *Implementation-defined* behavior allows free implementation of a specific behavior that needs to be documented. Examples of implementation-defined behavior are casts between pointers that underlie different alignment requirements across platforms. *Unspecified* behavior, unlike implementation-defined behavior, does not require the behavior to be documented. Since it is allowed to vary per instance, unspecified behavior typically includes cases in which compilers do not enforce a specific behavior. An example is using an unspecified value, which can, for example, be produced by reading padding bytes of a struct (C11 6.2.6.1 §6). Another example is the order of argument evaluation in function calls (C11 6.5.2.2 §10). *Undefined* behavior provides the weakest guarantees; the compiler is not bound to implement any specific behavior. A single occurrence of undefined behavior renders the whole program invalid. The tacit agreement between compiler writers seems to be that no meaningful code needs to be produced for undefined behavior, and that compiler optimizations can ignore it to produce efficient code [13]. Consequently, the current consensus among researchers and industry is that C programs should avoid undefined behavior in all instances, and a plethora of tools detect undefined behavior so that the programmer can eliminate it [e.g., 4, 5, 14, 17, 20, 31, 44, 46, 50]. Examples of undefined behavior are `NULL` dereferences, out-of-bounds accesses, integer overflows, and overflows in the shift amount.

1.2 Problems with Undefined Behavior

While implementing Safe Sulong, we found that most C programs exhibit undefined behavior and other portability issues. This is consistent with previous findings. For example, six out of nine SPEC CINT 2006 benchmarks induce undefined behavior in integer operations alone [11].

It is not surprising that the majority of C programs is not portable. On the surface, the limited number of language constructs makes it easy to approach the language; its proximity to the underlying machine allows examining and understanding how it is compiled. However, C's semantics are intricate; the informative Annex J on portability issues alone comprises more than twenty pages. As stated by Ertl, "[p]rogrammers are usually not language lawyers" [13] and rarely have a thorough understanding of the C standard. This is even true for experts, as confirmed by the *Cerberus* survey, which showed that C experts rely, for example, on being able to compare pointers to different objects using relational operators, which is clearly forbidden by the C standard [26].

Furthermore, much effort is required to write C code that cannot induce undefined behavior. For example, Figure 2 shows an addition that cannot overflow (which would induce undefined behavior). Such safe code is awkward to program and defeats C's original goal of defining its semantics such that efficient code can be produced across platforms.

In general, code that induces undefined behavior cannot always be detected at compile time. For example, adding two numbers is defined as long as no integer overflows happen. It is also seldom a problem when the program is compiled with optimizations turned off (e.g., with flag `-O0`). However, it is widely known that compilers perform optimizations at higher optimization levels that

[1]Note that reading an uninitialized variable produces an indeterminate value, which — depending on the type — can be a trap representation or an unspecified value.

```
signed int sum(signed int si_a, signed int si_b) {
  if (((si_b > 0) && (si_a > (INT_MAX - si_b))) ||
      ((si_b < 0) && (si_a < (INT_MIN - si_b)))) {
    /* Handle error */
  } else {
    return si_a + si_b;
  }
}
```

Figure 2: Avoiding overflows in addition [42]

cause programs to behave incorrectly if they induce undefined behavior [22, 34]. For instance, about 40% of the Debian packages contain unstable code that compilers optimize away at higher optimization levels, often changing the semantics because compilers exploit incorrect checks or undefined behavior in the proximity of checks [52]. This is worrisome, since optimizing away checks that the user deliberately inserted is likely to create vulnerabilities in the code [2, 12, 52]. Finally, code can be seen as a *time bomb* [34]. Increasingly powerful compiler optimization can cause programs to break with compiler updates; if code that induces undefined behavior does not break now, it might do so in the future [22].

1.3 Calls for a Lenient C

One strategy to tackle portability issues is to detect them and to fix the relevant code. To this end, a host of static and dynamic tools enable programmers to detect memory problems [4, 31, 44, 46], integer overflows [5, 50], type problems [20], and other portability issues in programs [14, 17]. Another approach is to educate programmers and inform them about common portability pitfalls in C. The most comprehensive guide to avoiding portability issues is the *SEI CERT C Coding Standard* [42], which documents best practices for C.

Due to the portability issues described, a more lenient dialect of C has been called for (see below). Rather than consider common patterns that are in conflict with the C standard as portability problems, it would explicitly support them by assigning semantics to them in a way that programmers would expect from the current C standard. Most code that would execute correctly at `-O0`, even if it induces undefined behavior, would also correctly execute with this dialect. For unrecoverable errors, it would require implementations to trap (i.e., abort the program). This dialect would be source-compatible with standard C: every program that compiles according to the C standard would also compile with this dialect. Consequently, such an effort would be different from safer, C-like languages such as *Polymorphic C* [45], *Cyclone* [18], and *CCured* [30], which require porting or annotating C programs.

Three notable proposals for such a safer C dialect can be found. Bernstein called for a "boring C compiler" [2] that would prioritize predictability over performance and could be used for cryptographic libraries. Such a compiler would commit to a specified behavior for undefined, implementation-defined, and unspecified semantics. The proposal did not contain concrete suggestions, except that uninitialized variables should be initialized to zero. A second proposal, a "Friendly Dialect of C", was formulated by Cuoq, Flatt, and Regehr [8]. The *Friendly C* dialect is similar to C, except that it replaces occurrences of undefined behavior in the standard either with defined behavior or with unspecified results (which do not render the whole program or execution invalid). Friendly C specifies 14 changes to the language, addressing some of the most important issues, but was meant to trigger discussion and not to cover all the deficiencies of the language comprehensively. Eventually, Regehr [35] abandoned the proposal and concluded that too many variations to a Friendly C standard would be possible to have experts reach a consensus. Instead, he proposed that reaching a consensus should be skipped in favor of developing a friendly C dialect, which could be adopted by more compilers if used by a broader community. A third proposal, for a "C*" dialect in which operations at the language level directly correspond to machine level operations, was outlined by Ertl [13]. Ertl observed that the C standard gave leeway to implementations to efficiently map C constructs to the hardware. However, he noted that compiler maintainers diverge from this philosophy and implement optimizations that go against the programmer's intent, by deriving facts from undefined behavior that enable more aggressive optimizations. Ertl believes that C programmers unknowingly write programs that target the C* dialect because they are not sufficiently familiar with the C rules. According to him, the effort needed to convert C* to C programs, however, would have a poor cost-benefit ratio, given that programmers could hand-tune the C code.

2 LENIENT C

We present *Lenient C*, a C dialect that assigns semantics to behavior in the C11 standard that is otherwise undefined, unspecified, or implementation-defined. Table 1 presents the rules that supersede those of the C11 standard and specify Lenient C.

A previous study categorized undefined behavior according to whether it involved the core language, preprocessing phases, or library functions [17]. We restrict Lenient C to the core language, and will consider extensions to it as part of future work, memory management functions being the only exception. We were primarily interested in undefined behavior that compilers cannot statically detect in all instances. Consequently, we disregarded problematic idioms such as writing-through consts [14], where an object with a const-qualified type is modified by casting it to a non-const-qualified type (C11 6.7.3 §6). We believe that increased research into compiler warnings and errors enables elimination of such bugs [47]. We also excluded undefined behavior caused by multiple modifications between sequence points [21] – which guarantee that all previous side effects have been performed – which includes expressions such as `i++ * i++` (C11 6.5 §2).

We created this dialect while working on the execution of C code on the JVM, using *Safe Sulong*, a C interpreter with a dynamic compiler. Per default, Safe Sulong aborts execution when it detects undefined behavior. However, we found that most programs induce undefined behavior or exhibit other portability issues. As an alternative to fixing such programs, we added an option to execute programs assuming the less strict Lenient C dialect.

Lenient C was inspired by Friendly C; additionally, we sought to support many anti-patterns that are described in the *SEI CERT C Coding Standard*, as they reflect non-portable idioms on which programmers rely. While the dialect can be implemented by static compilers, Lenient C programs are best suited to execution in a

ID	Lenient C	SEI CERT	Friendly C
General			
G1	Writes to global variables, traps, I/O, and program termination are considered to be side effects.		6, 13
G2	Externally visible side effects must not be reordered or removed.		6, 13
G3	Signed numbers must be represented in two's complement.	INT16-C	
G4	Variable-length arrays that are initialized to a length smaller or equal to zero must produce a trap.	ARR32-C	
Memory Management			
M1	Dead dynamic memory must eventually be reclaimed, even if it is not manually freed.	MEM31-C	
M2	Objects can be used as long as they are referenced by pointers.	MEM30-C	1
M3	Calling free() on invalid pointers must have no effect.	MEM34-C	
Accesses to Memory			
A1	Reading uninitialized memory must behave as if the memory were initialized with zeroes.	EXP33-C	8
A2	Reading struct padding must behave as if it were initialized with zeroes.	EXP42-C	
A3	Dereferences of `NULL` pointers must abort the program.	EXP34-C	4
A4	Out-of-bounds accesses must cause the program to abort.	MEM35-C	4
A5	A pointer must be dereferenceable using any type.	EXP39-C	10
Pointer Arithmetic			
P1	Computing pointers that do not point to an object must be permitted.	ARR30-C, ARR38-C, ARR39-C	9
P2	Overflows on pointers must have wraparound semantics.		9
P3	Comparisons of pointers to different objects must give consistent results based on an ordering of objects.	ARR36-C	
P4	Pointer arithmetic must work not only for pointers to arrays, but also for pointers to any type.	ARR37-C	
Conversions			
C1	Arbitrary pointer casts must be permitted while maintaining a valid pointer to the object.	MEM36-C, EXP36-C	10, 13
C2	Converting a pointer to an integer must produce an integer that, when compared with another pointer-derived integer, yields the same result as if the comparison operation were performed on the pointers.	INT36-C, ARR39-C	
C3	When casting a floating-point number to a floating point-number, or when casting between integers and floating-point numbers, a value not representable in the target type yields an unspecified value.	FLP34-C	
Functions			
F1	Non-void functions that do not return a result implicitly return zero.	MSC37-C	14
F3	A function call must trap when the actual number of arguments does not match the number of arguments in the function declaration.	EXP37-C, DCL40-C	
Integer Operations			
I1	Signed integer overflow must have wraparound semantics.	INT32-C	2
I2	The second argument of left- and right-shifts must be reduced to the value modulo the size of the type and must be treated as an unsigned value.	INT34-C	3
I3	Signed right-shift must maintain the signedness of the value; that is, it must implement an arithmetic shift.		
I4	If the second operand of a modulo or division operation is 0, the operation must trap.	INT33-C	5
I5	Like signed right-shifts, bit operations on signed integers must produce the same bit representation as if the value were cast to an unsigned integer.		7

Table 1: The rules of Lenient C compared to those of the *SEI CERT C Coding Standard* and Friendly C

managed environment. In other words, Lenient C makes some assumptions that hold for managed runtimes such as the JVM or .NET, but typically not for static compilers, such as LLVM and GCC, that compile C code to an executable. For example, Lenient C assumes automatic memory management. Although garbage collectors (GCs) exist that can be compiled into applications [3, 33], they are not commonly used. Nonetheless, we believe that many of Lenient C's rules might also inspire their implementation in static compilers.

In the following sections, we describe how we implemented the Lenient C dialect in Safe Sulong, and expand on its design decisions. Section 3 describes an object hierarchy suitable for implementing Lenient C in object-oriented languages. Section 4 addresses memory management and expands on Lenient C's memory management and memory error handling requirements. Section 5 examines operations on pointers, and Section 6 discusses how Lenient C envisages the implementation of arithmetic operations.

3 REPRESENTING C OBJECTS

All our C objects are represented using classes that inherit from a `ManagedObject` base class.[2] We found such an approach to be more idiomatic than using a single array of bytes to represent memory. Subclasses comprise integer, floating point, struct, union, array, pointer, and function pointer types. For example, we represent the C `float` type as a `Float` subclass. To denote values, we use Haskell-style type constructors. A `float` value `3.0` is thus expressed as `Float(3.0)`.

The `ManagedObject` class specifies methods for reading from and writing to objects that the subclasses need to implement. The read operation is expressed as a method `object.read(type, offset)` that reads a specific type from an object at a given offset. Note that the `offset` is always measured in bytes. For example, reading a float at byte offset 4 from an object is expressed as `object.read(Float, 4)`. The write method is expressed as a method `object.write(type, offset, value)`. The C standard requires that every object can

- be treated as a sequence of `unsigned chars`, so every subclass must implement at least one method that can read and write the `char` type (C11 6.2.6.1 §4),
- and can be read using the type of an object, so, for example, an `int` object must implement read and write methods for `int`.

Additionally, we allow objects to be treated using other types, by concatenating their byte representations (see Section 5.4).

3.1 Integer and Floating-Point Types

Safe Sulong represents primitive types as Java wrapper classes. In subsequent examples, we assume an LP64 model in which an `int` has 32 bits, a `long` 64 bits, and a pointer 64 bits. However, our architecture also works for other 64-bit and 32-bit models; we will point out differences that influence the implementation at the corresponding places in the text.

For the C types `bool`, `char`, `short`, `int`, and `long` we use wrapped Java primitive types. For example, an `int` in C corresponds to a 32-bit integer in LP64, and we map it to a Java class `I32` that holds a Java `int`. Note that we do not need separate types for signed and unsigned integers; for example, we represent both `int` and `unsigned int` using a Java `int`. However, we need to provide both signed and unsigned operations (see Section 6).

We also represent `float` and `double` types using wrapped Java equivalents. C has a `long double` data type that is represented as an 80-bit floating-point type on AMD64. Since this data type does not exist in Java, we provide a custom implementation that emulates the behavior of 80-bit floats.[3]

[2]Safe Sulong interprets LLVM IR [23], which is a RISC-like intermediate representation, and not C code. LLVM IR also contains other integer types (e.g., I33 and I48) that we map to a wrapped `byte` array.

[3]Emulating 80-bit floats is inefficient and error-prone. As part of future work, we plan to provide a more efficient implementation of this type. However, we found that only few C programs rely on `long doubles`.

```
int main() {
    int val, arr[3];
    int *ptr1 = &val;        // (val, 0)
    int *ptr2 = &arr[2];     // (arr, 8)
    int *ptr4 = 0;           // (NULL, 0)
}
```

Figure 3: Various pointer tuples

3.2 Pointers and Function Pointers

We implement pointers using a class `Address`. `Address` has two fields: a `ManagedObject` field `pointee` that refers to its pointee, and an integer `offset` that denotes the offset within the object. The offset must be large enough to hold an integer with the same bit width as a pointer; assuming LP64, it is 64 bits wide. We denote a pointer-tuple by `(pointee, offset)`. The idea of representing pointers as a tuple is not new; for example, formal C models [19, 24] and also previous implementations of C on the JVM used such a representation [10, 16]. Figure 3 shows tuples for three different pointers. `ptr1` points to the start of an `int`; the offset is `0`. `ptr2` points to the second element of an integer array; the offset is `8` (`2 * sizeof(int)`). `ptr3` is a `NULL` pointer, which is obtainable by an integer constant `0`. The C standard specifies that `NULL` is guaranteed to be unequal to any pointer that points to a function or object (C11 6.3.2.3). We implement the `NULL` constant by an `Address` that has a `null` pointee and an `offset` of 0. Note that Section 5.1 gives a detailed account of pointer arithmetic.

To represent function pointers, we use a class that comprises a wrapped `long` that represents a *function ID*. For every parsed function, a unique ID starting from 1 is assigned. An ID of 0 represents a `NULL` function pointer. For calls, this ID is used to locate the executable representation of the function. Note that forgotten return statements in non-void functions induce undefined behavior (C11 6.9.1 §12). To address this, Safe Sulong implicitly returns a zero value of the return type when control reaches the end of the function. Note that another error class is when a function call supplies a wrong number of arguments, for which Lenient C requires the function call to trap.

3.3 Arrays

We represent C arrays by means of Java classes that wrap Java arrays. Primitive C arrays are represented by primitive Java arrays. For example, the type `int[]` is represented as a Java `int` array. We represent other C arrays using Java arrays that have a `ManagedObject` subtype as their element type. For example, we represent C pointer arrays as Java `Address` arrays. In our type hierarchy, arrays and structs are nested objects, which the `read` and `write` operations must take into account. Consequently, a given `offset` value must be decomposed to select the array element and then the offset within that element. For example, to read a byte from an `I32Array`, the `I8` read operation computes the value as the right-most byte taken from `values[offset / 4] >> (8 * (offset % 4))`. The division selects the array member, and the modulo the byte inside the integer.

```
struct {
    int a;
    long b;
} t;

char* val = (char*) &t;
val[9] = 1;
```

Figure 4: Writing a char into a struct member

3.4 Structs and Unions

Java lacks a struct type. We represent structs using a map [55] that contains `ManagedObjects`. A struct member can be accessed using an operation `getfield(offset)` that returns a tuple `(object, offset′)`.[4] The `object` denotes the member stored at the byte position `offset`, and `offset′` denotes the offset relative to the start of this member. Figure 4 shows an example. Note that the struct has a size of 16 bytes, where the stored `int` takes up 4 bytes, the padding values after the `int` 4 bytes, and the `long` 8 bytes. To write a byte at offset 9, Safe Sulong first selects the member using `getfield(9)`; the tuple returned is `(I64(0), 1)`. It then writes the value to the selected member object using `object.write(I8, 1)`. The read operation looks similar.

Safe Sulong also takes into account padding bytes, whose values are not specified by the standard (C11 6.2.6.1 §6). It initializes such bytes with a sequence of `I8(0)`. This allows programmers, for example, to compare structs using byte-by-byte comparison. Note that an alternative way of representing structs would be to use classes that represent struct members by fields. In a source-to-source transformation approach, these classes could be generated when compiling the program [25]. In interpreters, this would be more complicated because they would have to generate Java bytecode at run time.

In this object hierarchy, unions are structs with only one field. Unions allow programmers to view a memory region under different types. When reading a value from a union using a type that is different from the type that was last used when storing to this union, the standard requires that the union be represented in the new type (C11 6.5.2.3 §3). To account for this, we allocate a union with a sub-type that reflects the most general member type: when aliasing primitive values and pointers, we select `Addresses` or arrays of `Addresses`, since integers and floating-point numbers can be stored in the `offset` field of an address. As an alternative to using a single type, a map operation `writefield(offset, object)` could be introduced that replaces an existing object at the given offset to store a member with a different type. Such an approach would resemble *tagged unions*, which are, for example, used by precise GCs for C [33].

4 MEMORY MANAGEMENT

Two of our main concerns are how to implement memory management for C and how to handle memory errors. Allocating stack objects and global objects is straightforward, since their type is known. We map such allocations to one of the types presented in Section 3. Variable-length array declarations that have a negative or zero size induce undefined behavior (C11 6.7.6.2 §5). We trap in such cases, which corresponds to Java's default behavior when the size is negative (we still have to explicitly check for zero). For heap objects (allocated by `malloc()`, `calloc()`, or `realloc()`) we do not know the type of object that will be stored in it. Thus, we allocate the corresponding Java object only on the first cast, read, or write operation (i.e., when the type of the object becomes known).[5] Another approach to addressing untyped heap allocations would be to determine a type using static analysis [20].

4.1 Uninitialized Memory

There is no clear guidance on how a program should behave when it reads from uninitialized storage, an action which can induce undefined behavior [43]. There are two contradictory use cases, one of which we must support in our lenient execution model.

The first use case is that some programs purposefully read from uninitialized memory to create entropy. The entropy originates from previously allocated memory; uninitialized stack reads can read previous activation frames, while uninitialized heap reads can read malloced and freed heap memory. This pattern is problematic, and commonly used bug-finding tools such as Valgrind [31] and MSan [46] report it as a program error. Another issue is that reads to uninitialized memory make applications prone to information-leak attacks [27]. While allowing a program to read stale values could be dangerous, initializing all data structures with random values (to create entropy) would be overkill.

The second scenario is that programmers read uninitialized storage by accident. When executing programs with Safe Sulong, we found a number of programs that forgot to initialize memory or assumed that it was zero-initialized. Those programs worked correctly when uninitialized reads returned zero, which was suggested by Bernstein [2]. Zero-initialization is also supported by SafeInit [27], a protection system for C/C++ programs. Like SafeInit, we decided to support the second scenario, as it does not obviously jeopardize system security. Our implementation initializes all values to zero (recursively for nested objects); primitives are initialized to zero values, while pointers are initialized to `NULL`. Note that this approach is close to Java's default behavior, which initializes fields with an `Object` type to zero values if they are not initialized explicitly.

4.2 Memory Leaks and Dangling Pointers

C requires programmers to manually manage heap memory: memory allocated by `malloc()` must be freed using `free()`. Forgetting to free an object causes a memory leak, which can impact performance and can lead to the application running out of memory. Since Safe Sulong runs on a JVM, the JVM's GC reclaims objects after they are no longer needed. Note that automatic memory management cannot easily be implemented for static compilers; hence, it is also not covered by Friendly C.

[4]Read operations typically always access the same struct field through its name in the source code. Consequently, we mitigate the map-lookup costs by caching the lookup result. Note that an index for a struct member is the same across struct objects, so reading the same struct field from different objects is also efficient.

[5]For efficiency, we propagate the type back to the allocation site, similarly to allocation mementos in V8 [7]. Subsequent calls to the allocation function directly allocate an object of the observed type.

A *dangling pointer* is a pointer whose pointee has exceeded its lifetime. Accessing such a pointer induces undefined behavior. There are two situations in which a dangling pointer can be created:

- A heap object is freed using `free()` (C11 7.22.3.3).
- A C object with automatic storage duration (i.e., a stack variable) exceeds its lifetime (C11 6.2.4 §2).

There is no use case for accessing dangling pointers; they are caused by errors in manual memory management. In our type hierarchy, we could detect such errors by setting automatic objects to `null` after leaving a function scope, and by letting `free()` calls set the data of a pointee to `null` [37]. However, since we strive for lenient execution, we do not set them to `null` and retain references to objects whose lifetimes are exceeded. Consequently, programs can access dangling pointers as if they were still alive. This is useful to execute programs which, for example, access a dangling pointer shortly after a `free()` call under the assumption that the memory has not yet been reallocated. Only when the program loses all references to a pointee does the GC reclaim the pointee's memory.

Not aborting execution on use-after-free errors is Lenient C's most controversial design decision, as many tools strive to find such errors [29, 49, 58]. Safe Sulong can be used to find such errors, when executing programs using its default mode (instead of assuming the Lenient C dialect).

4.3 Buffer Overflows and NULL Dereferences

Besides use-after-free errors and invalid free errors, also buffer overflows and `NULL` dereferences are of concern, as they induce undefined behavior. For buffer overflows, an out-of-bounds read could produce a predefined zero value. This would work well when a non-delimited string was passed to a function operating on it; when reading zero, the function would assume that it had reached the end of the string. However, we also found that some programs with out-of-bounds reads did not terminate when producing a zero value upon out-of-bounds reads. For example, the fasta-redux benchmark ran out of bounds while adding up floating-point values. Due to a rounding error, the number did not add up to 1.00, and the program only terminated when reading positive garbage values [36]. In general, this approach is known as failure-oblivious computing [40], which ignores out-of-bound writes and produces a sequence of predefined values to accommodate various scenarios. As there is no value sequence that works for all programs, we decided to trap on buffer overflows. This also corresponds to Java's default semantics. Since we represent C arrays and structs using Java arrays, Java automatically performs bounds checks on accesses. On most architectures, `NULL` dereferences produce traps and usually cause unrecoverable program errors. Consequently, Lenient C also traps on `NULL` dereferences.

5 POINTER OPERATIONS

Pointers and pointer arithmetic are the main difference between C and other higher-level languages such as Java and C#, which use managed references instead. Consequently, this section explains how Safe Sulong implements operations that involve pointers.

```
int main() {
    int arr[3] = {1, 2, 3};
    ptrdiff_t diff1 = &arr[3] - &arr[0];
    size_t diff2 = (size_t) &arr[3] - (size_t) &arr[0];
    printf("%td %ld\n", diff1, diff2); // prints 3 12
}
```

Figure 5: Computing the pointer difference

5.1 Pointer Arithmetic

Addition or subtraction of integers. The standard defines additions and subtractions where one operand is a pointer `P` and the other an integer `N` (C11 6.5.6). Such an operation yields a pointer with the same type as `P` which points `N` elements forward or backward, depending on whether the operation is an addition or subtraction. For example, `arr + 5` computes an address by taking the address of `arr` and incrementing it by five elements. In our hierarchy, such an address computation creates a new pointer based on the old pointee and an updated offset. We compute the pointer as a new tuple `(pointee, oldPointer.offset + sizeof(type) * N)`. For example, if `arr` was an `int`, we would compute the offset by `sizeof(int) * N`. Note that the standard defines these operators only for pointers to arrays (C11 6.5.6 §8), while Lenient C allows pointer arithmetic for pointers to any type.

Subtraction of two pointers. The standard defines that subtracting two pointers yields the difference of the subscripts of the two array elements (C11 6.5.6 §9). Figure 5 shows a code snippet that subtracts two pointers, where one points to the start and one to the end of an array; note that the standard requires a common pointee (or a pointer one beyond the last array element). We implement pointer subtraction by subtracting the two integer representations of the pointers (see Section 5.3). Note that it would be sufficient to subtract the two pointer offsets; however, this could lead to unexpected results for different `pointee`s (which is undefined behavior) with the same `offset` since the difference would suggest a common pointee.

Pointer overflow. The C standard allows pointers to point only to an object or to one element after it (C11 6.5.6 §8). The latter is useful when iterating over an array in a loop using a pointer. Lenient C abolishes these restrictions: in Safe Sulong a pointer is, through the `offset` field, handled like an integer and is, for example, allowed to overflow. However, we prohibit dereferencing out-of-bounds pointers (see Section 4.3).

Pointer comparisons. Two pointers `a` and `b` can be compared using the same comparison operators as integers and floating-point numbers.

Implementing the equality operators (== and !=) is straightforward. For example, to determine equality for two pointers, we check whether they refer to the same pointee and have the same pointer offset. In Java, we implement the pointee comparison using `a.pointee == b.pointee`, which checks for object equality. If the expression yields `true`, we also compare the offset using `a.offset == b.offset`.

Implementing the relation operators (<, >, <=, and >=) is more difficult. The C standard defines these operators only for pointers to the same object or its subobjects (for structs and arrays); comparing two different objects yields undefined behavior (C11 6.5.8 §5). To

```
void *memmove(void *dest, void const *src, size_t n) {
    char *dp = dest;
    char const *sp = src;
    if (dp < sp) {
        while (n-- > 0)  *dp++ = *sp++;
    } else {
        dp += n; sp += n;
        while (n-- > 0)  *--dp = *--sp;
    }
    return dest;
}
```

Figure 6: Non-portable implementation of `memmove` (adapted from [48])

implement standard-compliant behavior, comparing the pointer offset would be sufficient; for example, to implement <, we could compare `a.offset < b.offset`. However, we found that programs often compare pointers to different objects. For example, Figure 6 shows a naive implementation of `memmove` that potentially compares two pointers to different objects, which is undefined behavior. For such patterns, comparing only the pointer offsets would give unexpected results, since it does not order objects. Instead, we establish an ordering using the integer representations of the pointers (see Section 5.3).

5.2 Pointer-to-Pointer Casts

In general, casts between pointers are implementation-defined (C11 6.3.2.3 §7). At the platform level, they are undefined if the converted pointer is not correctly aligned for the referenced type. Safe Sulong's abstracted architecture does not require any pointer alignments, so we support casts between different pointer types, as required by Lenient C. Since in our architecture, pointer-to-pointer casts do not change the underlying object representation, we can simply achieve the desired behavior by not performing any action.

5.3 Conversions between Pointers and Integers

We found that many applications assume pointers to be regular integer types. Consequently, some programs arbitrarily convert pointers to integers, perform computations on the integers, convert them back and dereference them. Additionally, programmers sometimes craft pointers from integers that are not obviously related. For example, the Cerberus survey showed that programmers rely on being able to compute the difference between two pointers, and using the pointer difference to refer from one object to another [26]. Another example are compressed oops in the Hotspot VM, where on 64-bit architectures addresses are compacted to 32 bits [41]. Finally, some popular C applications store information in unused bits of an address [6].

Such patterns are implementation-defined and discouraged (C11 6.3.2.3 §5); for example, they often cause vulnerabilities when upgrading to a platform on which data types have a different bit width [56]. Approaches that represent C memory as an array can easily support them, but they cannot rely on the GC to reclaim dead C objects. When programmers can construct pointers arbitrarily, a GC cannot securely reclaim *any* objects. Consequently, GCs for C must compromise. For example, the *Boehm GC* assumes all values to be pointers that, if treated as pointers, refer to a valid memory region. The *Magpie* GC assumes only those values to be reachable that have a pointer type [33]. Given the tradeoffs, we present two strategies for converting integers to pointers: the first prohibits converting integers to pointers, and the second must — like the Boehm GC and Magpie — rely on heuristics for garbage collection.

The first strategy converts an address to a 64-bit integer value by concatening the 32-bit hash of the pointee with the `offset` (`(long) System.identityHashCode(pointee) << 32 | offset`). Once an address has been converted to an integer, it loses its reference to its `pointee`. When converted back to a pointer, we assign the integer value to `offset`, and `NULL` to the `pointee`. The pointer can no longer be dereferenced. This can be a problem if a pointer is copied byte-wise (e.g., in functions similar to `memmove` or `memcpy`), since only its integer representation is copied. If two pointers referring to the same object are converted to integers, the ordering is maintained if the `offset` does not exceed 32 bits. For pointers with a `NULL` `pointee` we use the 64-bit pointer offset as an integer representation to maintain the order relation between pointers that were converted back and forth to integers. Note that this approach is unsound for pointers to different `pointee`s, because it can yield identical or overlapping values for different pointees. This representation allows the relation operators to be total and transitive. However, it violates antisymmetry; that is, two pointers can have the same integer representation when they refer to different objects. That is, `(long) p == (long) q` for two pointers `p` and `q` can yield `true`, even if the pointers refer to different objects. Nevertheless, we have not yet found a program that relies on the antisymmetry property; programs typically use the equality operators on pointers to determine equality.

The second strategy is to assign a unique ID to every object when it is converted to an integer. The first strategy could also use unique IDs if an application requires antisymmetry. This ID is to be incremented by the size of the object. To support dereferencable pointers that were obtained by integers, we store "escaping" objects (i.e., objects whose pointers are converted to integers) in a tree data structure that associates the range of addressable bytes with an object. When an integer is converted to a pointer, the conversion operation looks up the object in this tree. Using the integer representation of the first strategy here would be dangerous, since an application could gain access to another object if they share the same hash code. Note that escaped objects stored in the tree would never be collected by the GC. To address this, the GC is allowed to collect such pointers when the application runs low on memory (by using a `SoftReference`). An alternative strategy would involve using a least-recently-used technique [32] to keep only those mappings alive that are used by the application. The drawback is that object graphs could be collected even though the application still wants to use them, specifically when the integer value is the only reference to the object graph, and when the application runs low on memory.

5.4 Reading from Memory

Two pointers can alias, which means that they can point to the same memory location. A frequent source of errors is that compilers assume that pointers cannot alias when programmers intend them to do [9]. The best known aliasing restriction is the strict-aliasing

```
int func(int *a, long *b) {
    *a = 5;
    *b = 8;
    return *a;
}
```

Figure 7: Example demonstrating strict aliasing

rule: the C11 standard specifies that two pointers of different types (if neither is a char pointer) cannot alias (C11 6.5 §7). Figure 7 shows an example that can yield unexpected results for programmers that are not familiar with this rule. Note that, without optimization, passing two identical pointers will likely yield a value of 8, since a and b alias. However, C's type rules do not allow them to alias, and when compiler optimizations are enabled, the return value is typically optimized to always be 5. Consequently, large projects often disable strict aliasing through the `-fno-strict-aliasing` compiler flag in GCC and LLVM [9, 26]. In Lenient C, we explicitly allow two pointers of different types to alias. Moreover, we allow that an incompatible type can be used to read from — or write a value to — the pointee. In Safe Sulong, storing a value or reading a value maps to a call of the `read` or `write` operation on the pointee.

6 ARITHMETIC OPERATIONS

C programmers often do not anticipate the semantics of corner cases in arithmetic operations. Many approaches try to find program errors related to arithmetic operations, especially integer-based errors [5, 11]. Our goal is to define the semantics of integer operations as programmers would currently expect from the C standard. To this end, Lenient C largely follows the way how corner cases are handled in Java, which also corresponds closely to the AMD64 operations. Note that unsigned operations can be implemented with operations on signed types. For example, we implement unsigned division on integers using `Integer.divideUnsigned` provided by the Java Class Library. Below, we explain how we address the corner cases in the arithmetic operations.

Data Model. C does not commit to a specific data model (e.g., LP64) that assigns sizes to all data types, and neither does Lenient C. However, in contrast to the C standard, we assume that signed integers are represented in two's complement, as is the case in most programming languages and hardware architectures. Consequently, we can assign useful semantics to implementation-defined corner cases in arithmetic operations. We define that right-shifting a negative value (of a signed type), which is implementation-defined (C11 6.5.7 §5), behaves like an arithmetic shift; that is, the sign bit of the value is extended to preserve the signedness of the number.
Signed integer overflow. While integer overflow is defined for unsigned types, it is undefined for signed integers. Many signed operations can overflow (+, -, *, /, %, and $\ll$ (C11 6.5.7 §4)), specifically when the result of the operation cannot be represented in the data type of the operation. Programs commonly rely on both signed and unsigned integer overflow, for example, for hashing, overflow checking, bit manipulation, and random-number generation [11]. Since in two's complement the range of representable positive and negative numbers is asymmetric, overflows can also occur for division and modulo.

On architectures that support two's complement, integer overflow typically wraps around, as most programmers would expect. GCC and Clang provide the `-fwrapv` flag, which enforces this behavior. For example, the SPEC 2000 `197.parser` benchmark requires this flag, since today's compilers would otherwise optimize the code in a way that lets the benchmark go into an infinite loop [11]. Safe Sulong provides wraparound semantics per default, which we implemented using the standard Java arithmetic operators.
Division by zero. If the second operand of a division or modulo operation is zero, the result is undefined (C11 6.5.5 §5). In most languages and on most architectures, division by zero traps.[6] Since it is unclear what value a division by zero should return, Lenient C always traps in such cases, which also corresponds to Java's default behavior.
Invalid shift amount. If the shift amount of a left- or right-shift is negative or greater or equal to the width of the shifted operand, the result is undefined. As initially demonstrated, architectures handle negative shift amounts and excessively large shift amounts differently. We decided to implement the semantics of Java, which also correspond to those of AMD64, where the shift amount is truncated to 5 bits.
Integer and floating-point conversions. Converting between floating-point numbers and integers or converting between floating-point numbers with different types can yield undefined behavior if the value is not representable by the destination type. Examples include converting `NaN` to an integer, converting a large `double` value to a `float`, and converting a large `long` value to a `float`. To implement casts efficiently across platforms, execution yields an unspecified value in such cases.

7 EVALUATION

We evaluated our Lenient C dialect by comparing it with the Friendly C standard and the *SEI CERT C Coding Standard* (see Table 1). Additionally, we implemented the dialect in Safe Sulong.

Comparison with Friendly C. Of the 14 features that the Friendly C standard proposes, Lenient C explicitly addresses 12, for which it mostly requires stricter guarantees. Friendly C aims to be implemented by a static compiler and makes tradeoffs that enable its efficient implementation across platforms (see below). Lenient C prioritizes consistent behavior and safety over speed, and allows implementers less leeway. Lenient C requires freed objects to stay alive, which meets Friendly C's requirement that a pointer's value should not change when its lifetime is exceeded (1). It requires trapping upon out-of-bounds accesses and `NULL` pointer dereferences (4), whereas Friendly C also allows an unspecified value. Friendly C demands more lenient treatment for signed-integer overflows (2), invalid shift amounts (3), division-related overflows (5), and unsigned left-shifts (7). Lenient C addresses these demands and leaves less leeway for a compatible implementation; for example, Friendly C specifies an unspecified result for shift operations with an invalid shift amount, while Lenient C requires the shift value to be masked. Like Lenient C, Friendly C requires that externally visible side effects not be reordered (6), and that a compiler should not be granted additional optimization opportunities when inferring that a pointer is invalid (13). Additionally, both Lenient C and

[6] In MySQL, however, division by zero yields a `NULL` result.

Friendly C abolish the strict aliasing rule (10). While Friendly C specifies reads from uninitialized storage to yield an unspecified value (8), Lenient C requires such reads to return 0. Both Friendly C and Lenient C allow out-of-bounds pointers and arbitrary computations on pointers (9). With respect to functions, Friendly C requires that, when control reaches the end of a non-void function, an unspecified value be returned if the return statement is missing (14); Lenient C requires 0 to be returned.

Both Friendly C and Lenient C are not comprehensive. Of the two points that Lenient C does not address, one (11) is related to data races. We will consider extending the Lenient C standard to address multithreading issues as part of future work. The other point (12) proposes `memmove` semantics for `memcpy`. Using `memcpy` with overlapping arguments is a common error, so Safe Sulong implements `memcpy` using `memmove`. However, we deferred the discussion of lenient semantics for standard library functions. Lenient C has additional guarantees compared to Friendly C. It demands additional lenience on memory management errors (M1, M2, M3) and requires struct padding to be initialized to 0 (A2). It also establishes an ordering on objects that should hold when pointers are converted to integers (P3, C2). Lenient C allows arbitrary pointer casts (C1) and pointer arithmetic on pointers to non-array objects (P4). Additionally, it specifies semantics when types or number of arguments in a function call do not match the function declaration (F2, F3). Lenient C requires signed numbers to be represented in two's complement (G3), an invalid size in variable-length arrays to trap (G4), and otherwise undefined casts to produce an unspecified value (C3).

Comparison with the *SEI CERT C Coding Standard*. The *SEI CERT C Coding Standard* is a forward-looking set of best practices for the C11 language. It comprises 14 chapters with individual rules, each of which describes a best practice along with anti-patterns. Our goal in Lenient C is not to rely on programmers following these practices; instead, we assume that they have anti-patterns in their code which they assume to work correctly. Thus, for our evaluation we examined whether Lenient C addresses such anti-patterns. The *SEI CERT C Coding Standard* recommendations are comprehensive, and we excluded a number of chapters because they do not fall into the scope of our work. Specifically, we excluded the chapters on the preprocessor (PRE), library functions (FIO, ENV, SIG, ERR), and concurrency problems (CON).

The chapter regarding declarations and initialization (DCL) contains several rules of interest to Lenient C. It requires that variables be declared with appropriate storage durations (DCL30-C); Lenient C keeps referenced objects alive and thus accepts inappropriate storage durations. The chapter requires that no incompatible declarations of the same object or function be made (DCL40-C), which Lenient C partly addresses by trapping when a function is called with a wrong number of arguments.

The chapter regarding expressions (EXP) consists of rules with different concerns: it discusses invalid read operations, non-portable pointer casts, and errors in calling functions. Lenient C allows programs to read uninitialized memory (EXP33-C) and compare padding values (EXP42-C), which it requires to be initialized with zeroes. When `NULL` pointers are dereferenced, Lenient C specifies that the implementation must trap (EXP34-C). It enables arbitrary pointer casts (EXP36-C) and reading pointers using an incompatible type (EXP39-C). Lenient C requires trapping when a function is called with a wrong number of arguments (EXP37-C). The rule also addresses wrong types in arguments, for which a callee in Safe Sulong performs automatic conversions.

The chapter on integers (INT) warns against wrong integer conversions (INT31-C), using types with an incorrect precision (i.e., bit width, INT35-C) and unsigned integer wrapping (which is defined behavior, INT30-C); these rules are of little concern to Lenient C. Lenient C specifies wrapping semantics for signed overflow (INT32-C), traps on division or remainder operations with zero as second operand (INT33-C), and requires the shift amount to be masked (INT34-C). One rule is concerned with conversions between pointers and integers (INT36-C) and details anti-patterns using crafted pointers, which are implementation-defined. Lenient C does not specify the semantics of casts between pointers and integers. Safe Sulong provides two different standard-compliant strategies, of which only the second (that stores escaped objects in a map) addresses user expectations in this context. As part of future work, we plan to investigate both strategies using a case study of user programs.

The chapter on floating-point numbers (FLP) is concerned mainly with issues that are valid for floating-point numbers in general, so they are of little interest in our evaluation; however, we defined that otherwise undefined casts between integers and floating-point numbers yield unspecified values (C3).

We addressed all anti-patterns of the array chapter (ARR), which primarily discusses pointer arithmetic. Lenient C supports pointer arithmetic on non-array types (ARR37-C, ARR39-C), creating out-of-bounds pointers (ARR30-C, ARR38-C, ARR39-C), but traps when dereferencing an out-of-bounds pointer (ARR30-C). It requires trapping for non-positive variable-length array sizes (ARR32-C). Additionally, Lenient C supports subtracting and comparing pointers to different objects (ARR39-C).

The characters and strings chapter (STR) discusses issues which are statically detectable or which concern the usage of library functions.

All rules of the memory management chapter (MEM), except the `realloc` alignment requirement, are of interest to us, and Lenient C addresses each of them. Lenient C allows dangling pointers (MEM30-C) to be accessed as if they were still alive and ignores invalid frees (MEM34-C). It assumes a GC that reclaims memory that is no longer needed (MEM31-C). Upon out-of-bounds accesses, Lenient C requires implementations to trap (MEM33-C, MEM35-C).

The miscellaneous chapter (MISC) mostly discusses library functions; however, MSC37-C states that control should never reach the end of a non-void function, in which case Lenient C specifies a zero value to be returned.

Implementation in Safe Sulong. We implemented Lenient C in Safe Sulong, a system for executing LLVM-based languages on the JVM. It does not directly execute C code, but LLVM IR, which is the RISC-like intermediate format of the LLVM framework [23]. We implemented Safe Sulong on top of the Truffle language implementation framework [53], which uses the Graal compiler [57] to compile frequently-executed functions to machine code. Graal optimizes the code based on Java semantics and thus preserves side effects such as `NULL` dereferences, out-of-bounds accesses, and arithmetic errors. Safe Sulong is based on Native Sulong [38], but it represents C objects on the managed Java heap instead of allocating them in native memory. Its peak performance — reached after a warm-up phase where Graal compiles frequently executed functions to machine code — is currently around half that of executables compiled by Clang `-O3` on small benchmarks. As part of future work we intend to thoroughly evaluate Safe Sulong's performance and lower its overhead.

8 RELATED WORK

ManagedC. We previously worked on a Truffle implementation for C called *ManagedC* [16]. ManagedC aimed to detect out-of-bounds accesses and use-after-free errors, but otherwise assumed strictly conforming C programs. Note that the implementation of Lenient C in Safe Sulong is based on ManagedC, in particular in its representation of pointers. However, while ManagedC had a relaxed mode which allowed some illegal type casts, it left open which C dialect it supported and how other portability issues (e.g., subtracting pointers to different objects) were addressed. Further, Lenient C's main goal is not to find errors in C programs, but to tolerate them whenever possible. Unlike ManagedC, we also tolerate use-after-free errors.

Dialects of C. Several C-like languages have been proposed, for example, *Polymorphic C* [45], *Cyclone* [18], and *CCured* [30]. These dialects add type safety and/or detection of memory errors to C-like languages, but are not source-compatible with C. They do not touch on other aspects of non-portable behavior.

Pointer-to-Integer casts. Kang et. al presented an approach to using pointer-to-integer casts in formal memory models [19]. Most formalizations rely on logical memory models (e.g., CompCert [24]), in which pointers are represented as pairs of an allocation block and an offset within that block, similar to our pointer pairs. They extended this approach such that a pointer has two representations: one in the concrete and one in the logical model. Per default, all allocation blocks are allocated as logical blocks; only when a pointer is cast to an integer is the logical pointer realized as an integer. This approach is similar to ours, where we convert pointers that are cast to an integer to a concrete representation that takes into account the hash code and `offset`.

C to Java converters. Several systems exist for executing C on the JVM, by converting C programs either to Java or to Java bytecode [10, 25]. C-to-Java systems are typically used to migrate legacy code and thus focus on producing readable code at the cost of correctness (e.g., by not supporting unsigned types [25]). Most of them do not support non-portable patterns such as casting pointers to integers. Only Demaine's approach touched on lenient execution [10]; for example, he stated that pointer comparisons between different objects could be established by ordering of the heap. These approaches use an object hierarchy similar to ours, which makes them suitable for implementing Lenient C.

CHERI. CHERI [54] is a RISC-based instruction set architecture that provides hardware support for memory safety through unforgeable fat-pointers (called capabilities). As with Safe Sulong, the CHERI authors found that it was straightforward to support well-behaved C programs, but that it was difficult to compile and run those with non-portable behavior. They studied problematic patterns (portable, undefined, and implementation-defined idioms) including removing const qualifiers, pointer arithmetic idioms, storing bits in an address and storing pointers in integer variables [6]. They found many instances of these patterns, and adapted their execution model to better support such idioms.

9 CONCLUSIONS AND FUTURE WORK

We found that implementing the Lenient C dialect is helpful when executing C programs that can be found "in the wild", as it removes the need to "fix" them to use only standard-compliant C. This dialect is best suited for execution on a managed runtime. However, we hope that some of the rules will be incorporated into static compilers to alleviate the problem of compiler optimizations conflicting with user assumptions (thus breaking their code). The inspiration to create this dialect came while executing non-portable programs with Safe Sulong. However, Safe Sulong is a prototype and cannot execute large programs, mainly due to unimplemented standard library functions. Consequently, we have validated Lenient C informally on programs of up to 5000 lines of code. We are currently adding support for running a complete, well-behaved libc (such as the musl libc [28]) on top of Safe Sulong. This requires Safe Sulong to execute inline assembly and provide functionality that is typically provided by the operating system [37]. Once Safe Sulong has reached a degree of completeness that enables it to execute larger applications, we will conduct a case study to determine which features of Lenient C are most useful for large real-world programs and which features are still missing. In particular, we have yet to determine which of the two strategies of converting between pointers and integers is most suitable in practice. Lenient C still lacks stricter semantics for standard library functions, preprocessing and other issues (e.g., related to const and restrict qualifiers). Furthermore, C/C++ concurrency semantics remain (among others) unsatisfactory [1, 8], and Lenient C currently lacks stricter semantics for multithreading. We will consider these issues as part of our future work.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. We also thank Ingrid Abfalter, whose proofreading and editorial assistance greatly improved the manuscript. We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute of System Software at Johannes Kepler University Linz for their support and contributions. We thank Benoit Daloze and Mike Hearn for comments on an early draft. The authors from Johannes Kepler University Linz were funded in part by a research grant from Oracle.

REFERENCES

[1] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings.* 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

[2] Daniel Julius Berstein. 2015. boringcc. (2015). https://groups.google.com/forum/m/#!msg/boring-crypto/48qa1kWignU/o8GGp2K1DAAJ (Accessed August 2017).

[3] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Software: Practice and Experience* 18, 9 (Sept. 1988), 807–820. https://doi.org/10.1002/spe.4380180902

[4] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on*

Code Generation and Optimization (CGO '11). IEEE Computer Society, Washington, DC, USA, 213–223. https://doi.org/10.1109/CGO.2011.5764689

[5] David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. 2007. RICH: Automatically protecting against integer-based vulnerabilities. *Department of Electrical and Computing Engineering* (2007), 28.

[6] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. 2015. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 117–130. https://doi.org/10.1145/2694344.2694367

[7] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM '15)*. ACM, New York, NY, USA, 105–117. https://doi.org/10.1145/2754169.2754181

[8] Pascal Cuoq, Matthew Flatt, and John Regehr. 2014. Proposal for a Friendly Dialect of C. (2014). https://blog.regehr.org/archives/1180 (Accessed August 2017).

[9] Pascal Cuoq, Loïc Runarvot, and Alexander Cherepanov. 2017. Detecting Strict Aliasing Violations in the Wild. In *Verification, Model Checking, and Abstract Interpretation: 18th International Conference, VMCAI 2017, Paris, France, January 15–17, 2017, Proceedings*, Ahmed Bouajjani and David Monniaux (Eds.). Springer International Publishing, Cham, 14–33. https://doi.org/10.1007/978-3-319-52234-0_2

[10] Erik D. Demaine. 1998. C to Java: converting pointers into references. *Concurrency: Practice and Experience* 10, 11-13 (1998), 851–861. https://doi.org/10.1002/(SICI)1096-9128(199809/11)10:11/13<851::AID-CPE385>3.0.CO;2-K

[11] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2015. Understanding Integer Overflow in C/C++. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1, Article 2 (Dec. 2015), 29 pages. https://doi.org/10.1145/2743019

[12] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *Proceedings of the 2015 IEEE Security and Privacy Workshops (SPW '15)*. IEEE Computer Society, Washington, DC, USA, 73–87. https://doi.org/10.1109/SPW.2015.33

[13] M Anton Ertl. 2015. What every compiler writer should know about programmers or "Optimization" based on undefined behaviour hurts performance. In *Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*.

[14] Jon Eyolfson and Patrick Lam. 2016. C++ const and Immutability: An Empirical Study of Writes-Through-const. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. 8:1–8:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.8

[15] Matthias Felleisen and Shriram Krishnamurthi. 1999. *Safety in Programming Languages*. Technical Report. Rice University.

[16] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Memory-safe Execution of C on a Java VM. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security (PLAS'15)*. ACM, New York, NY, USA, 16–27. https://doi.org/10.1145/2786558.2786565

[17] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 336–345. https://doi.org/10.1145/2813885.2737979

[18] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02)*. USENIX Association, Berkeley, CA, USA, 275–288. http://dl.acm.org/citation.cfm?id=647057.713871

[19] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A Formal C Memory Model Supporting Integer-pointer Casts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 326–335. https://doi.org/10.1145/2737924.2738005

[20] Stephen Kell. 2016. Dynamically Diagnosing Type Errors in Unsafe Code. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 800–819. https://doi.org/10.1145/2983990.2983998

[21] Robbert Krebbers. 2014. An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 101–112. https://doi.org/10.1145/2535838.2535878

[22] Chris Lattner. 2011. What Every C Programmer Should Know About Undefined Behavior. (2011). http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html (Accessed August 2017).

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[24] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (Dec. 2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[25] Johannes Martin and Hausi A Muller. 2001. Strategies for migration from C to Java. In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*. 200–209. https://doi.org/10.1109/.2001.914988

[26] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the De Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/2908080.2908081

[27] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. 2017. SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities. (Feb. 2017). https://www.vusec.net/download/?t=papers/safeinit_ndss17.pdf

[28] musl libc. 2017. (2017). https://www.musl-libc.org/ (Accessed August 2017).

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. (2010), 31–40. https://doi.org/10.1145/1806651.1806657

[30] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.* 27, 3 (May 2005), 477–526. https://doi.org/10.1145/1065887.1065892

[31] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[32] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. (1993), 297–306. https://doi.org/10.1145/170035.170081

[33] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. 2009. Precise Garbage Collection for C. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM '09)*. ACM, New York, NY, USA, 39–48. https://doi.org/10.1145/1542431.1542438

[34] John Regehr. 2010. A Guide to Undefined Behavior in C and C++. (2010). https://blog.regehr.org/archives/213 (Accessed August 2017).

[35] John Regehr. 2015. The Problem with Friendly C. (2015). https://blog.regehr.org/archives/1287 (Accessed August 2017).

[36] Manuel Rigger. 2016. Fix for fasta-redux C gcc #2 program. (2016). https://alioth.debian.org/tracker/?func=detail&atid=413122&aid=315503&group_id=100815 (Accessed August 2017).

[37] Manuel Rigger. 2016. Sulong: Memory Safe and Efficient Execution of LLVM-Based Languages. In *ECOOP 2016 Doctoral Symposium*. http://ssw.jku.at/General/Staff/ManuelRigger/ECOOP16-DS.pdf

[38] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL 2016)*. ACM, New York, NY, USA, 6–15. https://doi.org/10.1145/2998415.2998416

[39] Armin Rigo and Samuele Pedroni. 2006. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 944–953. https://doi.org/10.1145/1176617.1176753

[40] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. 2004. Enhancing Server Availability and Security Through Failure-oblivious Computing. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=1251254.1251275

[41] John Rose. 2012. CompressedOops. (2012). https://wiki.openjdk.java.net/pages/diffpages.action?pageId=11829259&originalId=26312779 (Accessed August 2017).

[42] Robert C. Seacord. 2008. *The CERT C Secure Coding Standard* (1st ed.). Addison-Wesley Professional.

[43] Robert C. Seacord. 2016. Uninitialized Reads. *Queue* 14, 6, Article 50 (Dec. 2016), 17 pages. https://doi.org/10.1145/3028687.3041020

[44] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[45] Geoffrey Smith and Dennis Volpano. 1998. A Sound Polymorphic Type System for a Dialect of C. *Sci. Comput. Program.* 32, 1-3 (Sept. 1998), 49–72. https://doi.org/10.1016/S0167-6423(97)00030-0

[46] E. Stepanov and K. Serebryany. 2015. MemorySanitizer: Fast detector of uninitialized memory use in C. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 46–55. https://doi.org/10.1109/CGO.2015.7054186

[47] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and Analyzing Compiler Warning Defects. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 203–213. https://doi.org/10.1145/2884781.2884879

[48] What's the difference between memcpy and memmove? 2017. (2017). http://c-faq.com/ansi/memmove.html (Accessed August 2017).

[49] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 405–419. https://doi.org/10.1145/3064176.3064211

[50] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. 2009. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *NDSS*.

[51] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined Behavior: What Happened to My Code?. In *Proceedings of the Asia-Pacific Workshop on Systems (APSYS '12)*. ACM, New York, NY, USA, Article 9, 7 pages. https://doi.org/10.1145/2349896.2349905

[52] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 260–275. https://doi.org/10.1145/2517349.2522728

[53] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. ACM, New York, NY, USA, 13–14. https://doi.org/10.1145/2384716.2384723

[54] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 457–468. http://dl.acm.org/citation.cfm?id=2665671.2665740

[55] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. An Object Storage Model for the Truffle Language Implementation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2647508.2647517

[56] Christian Wressnegger, Fabian Yamaguchi, Alwin Maier, and Konrad Rieck. 2016. Twice the Bits, Twice the Trouble: Vulnerabilities Induced by Migrating to 64-Bit Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 541–552. https://doi.org/10.1145/2976749.2978403

[57] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[58] Yves Younan. 2015. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*. https://doi.org/10.14722/ndss.2015.23190

Chapter 7

Introspection

This chapter includes the paper that presents and evaluates our initial introspection ideas, where we proposed to provide metadata tracked by bug-finding tools to programmers.

Paper: Manuel Rigger, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. Introspection for C and its Applications to Library Robustness. *The Art, Science, and Engineering of Programming*, (2), 2018

Note: The last page, which contains a description of the paper's authors, was omitted for brevity.

Introspection for C and its Applications to Library Robustness

Manuel Rigger[a], René Mayrhofer[a], Roland Schatz[b], Matthias Grimmer[b], and Hanspeter Mössenböck[a]

a Johannes Kepler University Linz, Austria

b Oracle Labs, Austria

Abstract **Context:** In C, low-level errors, such as buffer overflow and use-after-free, are a major problem, as they cause security vulnerabilities and hard-to-find bugs. C lacks automatic checks, and programmers cannot apply defensive programming techniques because objects (e.g., arrays or structs) lack run-time information about bounds, lifetime, and types.

Inquiry: Current approaches to tackling low-level errors include dynamic tools, such as bounds or type checkers, that check for certain actions during program execution. If they detect an error, they typically abort execution. Although they track run-time information as part of their runtimes, they do not expose this information to programmers.

Approach: We devised an introspection interface that allows C programmers to access run-time information and to query object bounds, object lifetimes, object types, and information about variadic arguments. This enables library writers to check for invalid input or program states and thus, for example, to implement custom error handling that maintains system availability and does not terminate on benign errors. As we assume that introspection is used together with a dynamic tool that implements automatic checks, errors that are not handled in the application logic continue to cause the dynamic tool to abort execution.

Knowledge: Using the introspection interface, we implemented a more robust, source-compatible version of the C standard library that validates parameters to its functions. The library functions react to otherwise undefined behavior; for example, they can detect lurking flaws, handle unterminated strings, check format string arguments, and set *errno* when they detect benign usage errors.

Grounding: Existing dynamic tools maintain run-time information that can be used to implement the introspection interface, and we demonstrate its implementation in Safe Sulong, an interpreter and dynamic bug-finding tool for C that runs on a Java Virtual Machine and can thus easily expose relevant run-time information.

Importance: Using introspection in user code is a novel approach to tackling the long-standing problem of low-level errors in C. As new approaches are lowering the performance overhead of run-time information maintenance, the usage of dynamic runtimes for C could become more common, which could ultimately facilitate a more widespread implementation of such an introspection interface.

ACM CCS 2012

- **Computer systems organization** → **Dependable and fault-tolerant systems and networks**;
- **Software and its engineering** → **Language features**; **Error handling and recovery**; *Software reliability*;

Keywords reflection for C, library robustness, fault tolerance

The Art, Science, and Engineering of Programming

Submitted July 31, 2017

Published December 6, 2017

DOI 10.22152/programming-journal.org/2018/2/4

1 Introduction

Since the birth of C almost 50 years ago, programmers have written many applications in it. Even the advent of higher-level programming languages has not stopped C's popularity, and it remains widely used as the second-most popular programming language [47]. However, C provides few safety guarantees and suffers from unique security issues that have disappeared in modern programming languages. Buffer overflow errors, where a pointer that exceeds the bounds of an object is dereferenced, are the most serious issue in C [9]. Other security issues include use-after-free errors, invalid free errors, reading of uninitialized memory, and memory leaks. Numerous approaches exist that prevent such errors in C programs by detecting these illegal patterns statically or during run time, or by making it more difficult to exploit them [46, 48, 55]. When an error happens, run-time approaches abort the program, which is more desirable than risking incorrect execution, potentially leaking user data, executing injected code, or corrupting program state.

However, we believe that in many cases programmers could better respond to illegal actions in the application logic if they could use the metadata of run-time approaches (e.g., bounds information) to check invalid actions at run time and prevent them from happening. Library implementers in particular could use it to protect themselves from user input and to compensate for the lack of exception handling in C. For example, if they could check that an access would go out-of-bounds in a server library, they could log the error and ignore the invalid access to maintain availability of the system (as in failure-oblivious computing [35]). If the error happened in the C standard library instead, they could set the global integer variable `errno` to an error code, for example, to `EINVAL` for invalid arguments. Furthermore, a *special value* (such as `-1` or `NULL`) could be returned to indicate that something went wrong. Finally, explicit checks could prevent lurking flaws that would otherwise stay undetected. For example, in the case that a function does not actually access an invalid position in the buffer, bounds checkers cannot detect when an incorrect array size is passed to the function. Using bounds metadata, programmers could validate the passed against the actual array size.

In this paper, we present a novel approach that allows C programmers to query properties of an *object* (primitive value, struct, array, union, or pointer) so that they can perform explicit sanity checks and react accordingly to invalid arguments or states. These properties comprise the bounds of an object, the memory location, the number of arguments of a function with varargs, and whether an object can be used in a certain way (e.g., called as a function that expects and returns an `int`). The presented approach is *complementary* to dynamic tools, and does not aim to replace them. Programmers can insert custom input validations and error-handling logic where needed, but the dynamic tool that tracks the exposed metadata still aborts execution for errors that are not handled at the application level. Ultimately, this provides programmers with greater flexibility and increases the robustness of libraries and applications, defined as "[t]he degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [21].

As a proof of concept, we implemented the introspection interface for Safe Sulong [32], a bug-finding tool and interpreter with a dynamic compiler for C. Safe Sulong prevents buffer overflows, use-after-free, variadic argument errors, and type errors by checking accesses and aborting execution upon an invalid action. It already maintains relevant run-time information that it can expose to the programmer.

In a case study, we demonstrate how the introspection functions facilitate re-implementing the C standard library (`libc`) to validate input arguments. We use this `libc` in Safe Sulong as a source-compatible, more robust drop-in replacement for the GNU C Library. In contrast to the GNU C Library and other implementations, it can detect lurking flaws, handle unterminated strings, check format string arguments, and – instead of terminating execution – set *errno* when errors occur.

A plethora of other dynamic-bug finding tools and runtimes for C exist, and they could expose their run-time data via the introspection functions introduced in this paper. For example, bounds checkers [2, 11, 30, 38] could expose bounds information. Temporal memory safety tools [4, 19, 29, 31, 39, 44] could expose information about memory locations. Variadic argument checkers [3] and type checkers [18, 22] could expose information about variadic arguments and types. There are also combined tools that, for example, provide protection against both out-of-bounds accesses and use-after-free errors [17, 29, 30].

As the overhead of dynamic tools is decreasing [22, 29, 30, 38, 44], they could become standard in production, similar to stack canaries and address space layout randomization [46]. At this point in time, a wider adoption of the introspection functions (or a subset thereof) seems feasible. Additionally, we envisage that dynamic tools available now could distribute specialized libraries that benefit from introspection, as we will demonstrate using Safe Sulong's `libc` as an example.

In summary, this paper contributes in the following ways:

- We present introspection functions designed to allow programmers to prevent illegal actions that are specific to C (Section 3).
- We demonstrate how we implemented the introspection functions in Safe Sulong, a bug-finding tool and interpreter with a dynamic compiler for C (Section 4).
- In a case study, we show how using introspection increases the robustness of the C standard library (Section 5).

2 Background

In C, the lack of type and memory safety causes many problems, such as hard-to-find bugs and security issues. Moreover, manual memory management puts the burden of deallocating objects on the programmer. Consequently, C programs are plagued by vulnerabilities that are unique to the language. Faults can invoke undefined behavior, so compiled code can crash, compute unexpected results, and corrupt or read neighboring objects [50, 51]. It is often impossible to design C functions such that they are secure against usage errors, since they cannot validate passed arguments or global data. Below we provide a list of errors and vulnerabilities in C programs that we target in this work.

Out-of-bounds errors. Out-of-bounds accesses in C are among the most dangerous software errors [9, 37], since – unlike higher-level languages – C does not specify automatic bounds checks. Further, objects have no run-time information attached to them, so functions that operate on arrays require array-size arguments. Alternatively, they need conventions such as terminating an array by a special value.
Listing 1 shows a typical buffer overflow. The read_number() function reads digits entered by the user into the passed buffer arr and validates that it does not write beyond its bounds. However, its callee passes -1 as the length parameter, which is (through the size_t type) treated as the unsigned number SIZE_MAX. Thus, the bounds check is rendered useless, and if the user enters more than nine digits, the read_number() function overflows the passed buffer.
A recent similar real-world vulnerability is CVE-2016-3186, where a function in libtiff cast a negative value to size_t. As another example, in CVE-2016-6823 a function in ImageMagick caused an arithmetic overflow that resulted in an incorrect image size. Both faults resulted in buffer overflows.

Memory management errors. Objects that are allocated in different ways (e.g., on the stack or by malloc()) have different lifetimes, which influences how they can be used. For example, it is forbidden to access memory after it has been freed (otherwise known as an access to a *dangling pointer*). Other such errors include freeing memory twice, freeing stack memory or static memory, and calling free() on a pointer that points somewhere into the middle of an object [29]. Listing 2 shows examples of a use-after-free and a double-free error. Firstly, when err is non-zero, the allocated pointer ptr is freed and later accessed again as a dangling pointer in logError(). Secondly, the code fragment attempts to free the pointer again after logging the error, which results in a double-free vulnerability. C does not provide mechanisms to retrieve the lifetime of an object, which would allow checking and preventing such conditions. Consequently, use-after-free errors frequently occur in real-world code. For example, in CVE-2016-4473 the PHP Zend Engine attempted to free an object that was not allocated by one

Listing 1 Passing -1 to the size_t parameter renders the range check useless and could cause an out-of-bounds error while writing read characters to arr

```c
void read_number(char* arr, size_t length) {
  int i = 0;
  if (length == 0) return;
  int c = getchar();
  while (isdigit(c) && (i + 1) < length) {
    arr[i++] = c; c = getchar();
  }
  arr[i] = '\0';
}
// ...
char buf[10];
read_number(buf, -1);
printf("%s\n", buf);
```

Listing 2 Use-after-free error which is based on an example from the CWE wiki

```c
char* ptr = (char*) malloc(SIZE * sizeof(char));
if (err) {
  abrt = 1; free(ptr);
}
// ...
if (abrt) {
  logError("operation aborted", ptr); free(ptr);
}
// ...
void logError(const char* message, void* ptr) {
    logf("error while processing %p", ptr);
}
```

of libc's allocation functions. Other recent examples include a dangling pointer access and a double free error in OpenSSL (CVE-2016-6309 and CVE-2016-0705).

Variadic function errors. Variadic functions in C rely on the programmer to pass a count of variadic arguments or a format string. Furthermore, a programmer must pass the matching number of objects of the expected type. Listing 3 shows an example that uses variadic arguments to print formatted output, similar to C's sprintf() function. It is based on a function taken from the PHP Zend Engine. As arguments, the function expects a format string fmt, the variadic arguments ap, and a buffer xbuf to which the formatted output should be written. To use the function, a C programmer has to invoke a macro to set up and tear down the variadic arguments (respectively va_start() and va_end()). Using the va_arg() macro, xbuf_format_converter() can then directly access the variadic arguments. The example shows how a string can be accessed (format specifier "%s") that is then inserted into the buffer xbuf.
The function uses the format string to determine how many variadic arguments should be accessed. For example, for a format string "%s %s" the function attempts to access two variadic arguments that are assumed to have a string type. Accessing a variadic argument via va_arg() usually manipulates a pointer to the stack and pops the number of bytes that correspond to the specified data type (char * in our example). In a so-called *format string attack*, in which the function reads or writes beyond the stack due to nonexistent arguments, an attacker can exploit the inability of the function to verify the number and the types of the variadic arguments passed [8, 40].
In CVE-2015-8617, this function was the sink of a vulnerability that existed in PHP-7.0.0. The zend_throw_error() function called xbuf_format_converter() with a message string that was under user control. Consequently, an attacker could use format specifiers without matching arguments to read from and write to memory, and thus execute arbitrary code. As another example, in CVE-2016-4448 a vulnerability in libxml2 existed because format specifiers from untrusted input were not escaped.

Lack of type safety. Due to the lack of type safety, a programmer cannot verify whether an object referenced by a pointer corresponds to its expected type [22]. Listing 4 demonstrates this for function pointers. The apply() function expects a function pointer that accepts and returns an int. It uses the function to transform all elements of an array. However, its callee might pass a function that returns a double; a call on it would result in undefined behavior. Such "type confusion" cannot be avoided when

Listing 3 Example usage of variadic functions, taken from the PHP Zend Engine

```c
static void xbuf_format_converter(void *xbuf, const char *fmt, va_list ap) {
  char *s = NULL;
  size_t s_len;
  while (*fmt) {
    if (*fmt != '%') {
      INS_CHAR(xbuf, *fmt);
    } else {
      fmt++;
      switch (*fmt) {
        // ...
        case 's':
        s = va_arg(ap, char *);
        s_len = strlen(s);
        break;
        // ...
      }
      INS_STRING(xbuf, s, s_len);
    }
  }
}
```

Listing 4 Example of type confusion

```c
int apply(int* arr, size_t n, int f(int arg1)) {
  if (f == NULL) return -1;
  for (size_t i = 0; i < n; i++)
    arr[i] = f(arr[i]);
  return 0;
}

double square(int a) { return a * a; }

apply(arr, 5, square);
```

calling a function pointer, since objects have no types attached that could be used for validation.

Unterminated strings. Unterminated strings are a problem, since the string functions of libc (and sometimes also application code) rely on strings ending with a '\0' (null terminator) character. However, C standard library functions that operate on strings lack a common convention on whether to add a null terminator [28]. Additionally, it is not possible to verify whether a string is properly terminated without potentially causing buffer overreads. Listing 5 shows an example of an unterminated string vulnerability. The read function reads a file's contents into a string inputbuf. After the call, inputbuf is unterminated if the file was unterminated or if MAXLEN was exceeded. This is likely to cause an out-of-bounds write in strcpy(), since it copies characters to buf until a null terminator occurs. Recent similar real-world vulnerabilities include CVE-2016-7449, where strlcpy() was used to copy untrusted (potentially unterminated) input in GraphicsMagick. Further examples are CVE-2016-5093 and CVE-2016-0055, where strings were not properly terminated in the PHP Zend Engine as well as in Internet Explorer and Microsoft Office Excel [27].

Unsafe functions. Some functions in common libraries such as libc have been designed such that they "can never be guaranteed to work safely" [2, 12]. The most prominent

Listing 5 Example fragment that may produce and copy an unterminated string

```c
read(cfgfile , inputbuf, MAXLEN);
char buf[MAXLEN];
strcpy(buf, inputbuf);
puts(buf);
```

example is the gets() function, which reads user input from stdin into a buffer passed as an argument. Since gets() lacks a parameter for the size of the supplied buffer, it cannot perform any bounds checking and overflows the buffer if the user input is too large. Although C11 replaced gets() with the more robust gets_s() function, legacy code might still require the unsafe gets() function. In general, functions that lack size arguments – which prevents safe access to arrays – cannot be made safe without breaking source and binary compatibility.

3 Introspection Functions

To enable C programmers to validate arguments and global data, we devised introspection functions to query properties of C objects and the current function (see Appendix B). These functions allow programmers only to inspect objects and not to manipulate them; therefore, the presented functions are not a full reflection interface.

We designed these functions specifically to provide users with the ability to prevent buffer overflow, use-after-free, and other common errors specific to C. Through introspection, programmers can validate certain properties (memory location, bounds, and types) before performing an operation on an object. Additionally, introspection allows the number of variadic arguments passed to be queried and their types to be validated.

We built introspection based on several *introspection primitives*. These primitives are a minimal set of C functions that require run-time support. We also designed *introspection composites*, which are implemented as normal C functions and are based on the introspection primitives or on other composites. The introspection functions that we expose to the programmer contain both selected primitives and composites. We hereafter denote internal functions that are private to an implementation with an underscore prefix.

3.1 Object Bounds

Most importantly, we provide functions that enable the programmer to perform bounds checks before accessing an object. Simply providing a function that returns the size of an object is insufficient, since a pointer can point to the middle of an object. Instead, we require the runtime to provide two functions to return the space (in bytes) to the left and to the right of a pointer target: _size_left() and _size_right(). Their result is only defined for *legal* pointers, which we define as pointers that point to valid objects (not INVALID, see Section 3.2).

Listing 6 Example of how to query the space to the left and to the right of a pointee

```c
int *arr = malloc(sizeof(int) * 10);
int *ptr = &(arr[4]);
printf("%ld\n", size_left(ptr)); // prints 16
printf("%ld\n", size_right(ptr)); // prints 24
```

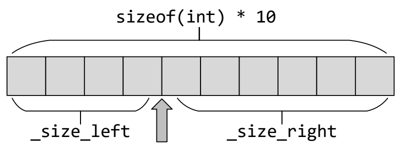

Figure 1 Memory Layout of the Example in Listing 6

Listing 7 Implementation of size_left() using the functions location(), _size_left(), and _size_right()

```c
long size_left(const void *ptr) {
  if (location(ptr) == INVALID) return -1;
  bool inBounds = _size_right(ptr) >= 0 && _size_left(ptr) >= 0;
  if (!inBounds) return -1;
  return _size_left(ptr);
}
```

Listing 6 illustrates the function return values when passing a pointer to the middle of an integer array to these functions. For the pointer to the fourth element of the ten-element integer array, _size_left() returns sizeof(int) * 4, and _size_right() returns sizeof(int) * 6. Figure 1 shows the corresponding memory layout. On an architecture where an int is four bytes in size the functions return 16 and 24, respectively.

We do not expose these two functions to the programmer, but base the composite functions size_left() and size_right() on them, which return -1 if the passed argument is not a legal pointer or out of bounds. Listing 7 shows the implementation of size_left(). Using location(), the function first checks that the pointer is legal (see Section 3.2). It then checks that the spaces to the left and to the right of the pointer are not negative, that is, the pointer is in bounds. If both checks are passed, the function returns the space to the left of the pointer using _size_left(); otherwise, it returns -1.

Listing 8 shows how using size_right() improves read_number()'s robustness (see Listing 1): If arr is a valid pointer but points to memory that cannot hold length chars, we can prevent the out-of-bounds access by aborting the program. Note that the check also detects lurking bugs, since it aborts even if fewer than length characters are read. If arr is not a valid pointer, the return value of size_right() is -1.

3.2 Memory Location

Querying the memory location of an object (e.g., stack, heap, global data) allows a programmer to obtain information about the lifetime of an object. For example, it enables programmers to prevent use-after-free errors by detecting whether an object has already been freed. Another use case is validating that no stack memory is

Listing 8 By using the size_right() function we can avoid out-of-bounds accesses in read_number()

```c
void read_number(char* arr, size_t length) {
  int i = 0;
  if (length == 0) return;
  if (size_right(arr) < length) abort();
  // ...
}
```

Listing 9 Example of how the location() enum constants relate to objects in a program

```c
int a;                        // location(&a) returns STATIC for global objects
void func() {
  static int b;               // location(&b) returns STATIC for static local objects
  int c;                      // location(&c) returns AUTOMATIC for stack objects
  int* d = malloc(sizeof(int) * 10);
                              // location(&d) returns DYNAMIC for heap objects
  free(d);                    // location(&d) returns INVALID for freed objects
}
```

Listing 10 By using location() and _size_left() we can check whether an object can be freed

```c
bool freeable(const void *ptr) {
  return location(ptr) == DYNAMIC && _size_left(ptr) == 0;
}
```

returned by a function. A programmer can also check whether a location refers to dynamically allocated memory to ensure that free() can be safely called on it. For this purpose, we provide the function location(), which determines where an object lies in memory.

The function returns one of the following enum constants:

- INVALID locations denote NULL pointers or deallocated memory (freed heap memory or dead stack variables). Programs must not access such objects.
- AUTOMATIC locations denote non-static stack allocations. Functions must not return allocated stack variables that were declared in their scope, since they become INVALID when the function returns. Further, stack variables must not be freed.
- DYNAMIC locations denote dynamically allocated heap memory created by malloc(), realloc(), or calloc(). Only memory allocated by these functions can be freed.
- STATIC locations denote statically allocated memory such as global variables, string constants, and static local variables. Static compilers usually place such memory in the text or data section of an executable. Programs must not free statically allocated memory.

Listing 9 shows how differently allocated memory relates to the enum constants used by location().

We provide the function freeable(), which is based on location(), to conveniently check whether an allocation can be freed. As Listing 10 demonstrates, a freeable object's location must be DYNAMIC, and its pointer must point to the beginning of an object. Listing 11 shows how we can use the freeable() function to improve the

Listing 11 By using the freeable() function we can avoid double-free errors

```c
char* ptr = (char*) malloc(SIZE * sizeof(char));
if (err) {
  abrt = 1;
  if (freeable(ptr)) free(ptr);
}
// ...
if (abrt) {
  logError("operation aborted", ptr);
  if (freeable(ptr)) free(ptr);
}
```

Listing 12 By using the location() function we can avoid use-after-free errors

```c
void logError(const char* message, void* ptr) {
  if (location(ptr) == INVALID)
    log("dangling pointer passed to logError!");
  else
    logf("error while processing %p", ptr);
}
```

robustness of the code fragment shown in Listing 2. It ensures that freeing the pointee is valid, and thus prevents invalid free errors, such as double freeing of memory. Nonetheless, the logError() function may receive a dangling pointer as an argument. To resolve this, we can check in logError() whether the pointer is valid (see Listing 12).

Note that some libraries, such as OpenSSL, use custom allocators to manage their memory. Custom allocators are outside the scope of this paper, but could be supported by providing source-code annotations for allocation and free functions; this information could then be used by the runtime to track the memory. The annotations for the allocation functions would need to specify how to compute the size of the allocated object, and the location of the allocated memory. Additionally, it might be desirable to add further enum constants, for example, for shared, file-backed, or protected memory. We omitted additional constants for simplicity.

3.3 Type

We provide a function that allows the programmer to validate whether an object is *compatible with* (can be treated as being of) a certain type. Such a function enables programmers to check whether a function pointer actually points to a function object (and not to a long, for example) and whether it has the expected function signature. As another example, programmers can use the function as an alternative to size_right() and size_left() to verify that a pointer of a certain type can be dereferenced.

C has only a weak notion of types, which makes it difficult to design expressive type introspection functions. For example, it is ambiguous whether a pointer of type int* that points to the middle of an integer array should be considered as a pointer to an integer or as a pointer to an integer array. Another example is heap memory, which lacks a dynamic type; although programmers usually coerce them to the desired type, objects of different types can be stored. Even worse, when writing to memory, objects

Listing 13 By using try_cast() we can ensure that we can perform an indirect call on the function pointer in apply()

```c
int apply(int* arr, size_t n, int f(int arg1)) {
  if (size_right(arr) < sizeof(int) * n || try_cast(&f, type(f)) == NULL)
    return -1;
  for (size_t i = 0; i < n; i++)
    arr[i] = f(arr[i]);
  return 0;
}
```

can be partially overwritten; for instance, half of a function pointer can be overwritten with an integer value, which makes it difficult to decide whether the pointer is still a valid function pointer.

Instead of assuming that a memory region has a specific type, we designed a function that allows the programmer to check whether the memory region is compatible with a certain type (similar to [22]). The try_cast() function expects a pointer to an object as the first argument and tries to cast it to the Type specified by the second argument. If the runtime determines that the cast is possible, it returns the passed pointer, and NULL otherwise. The cast is only possible if the object can be read, written to, or called as the specified type.

The Type object is a recursive struct which makes it possible to describe nested types (known as type expressions [1]). For example, a function pointer with an int parameter and double as the return type can be represented by a tree of three Type structs. The root struct specifies a function type and references a struct with an int type as the argument type as well as a struct with a double type as the return type. Since manually constructing Type structs is tedious, we specified the *optional* operator type(). As an argument, it requires the expression *example value*, whose declared type is returned as a Type run-time data structure. Since the declared type is a compile-time property, we want to resolve the type() operator during compile time; consequently, the programmer cannot take type()'s address and call it indirectly. The operator is similar to the GNU C extension typeof, which yields a type that can be used directly in variable declarations or casts.

Listing 13 shows how the type introspection functions make the function apply() (see Listing 4) more robust: apply() uses try_cast() to check whether the runtime can treat its first argument as the specified function pointer. Its second argument is the Type object that the type operator constructs from the declared function pointer type. The try_cast() function returns the first argument if it is compatible with the specified function pointer type; otherwise, it returns NULL. In addition to preventing the calling of invalid function pointers, apply() prevents out-of-bounds accesses by validating the array size.

The try_cast() function is similar to C++'s dynamic_cast(). However, we want to point out that C++'s dynamic_cast() works only for class checks (which are well-defined), while our approach works for all C objects. We believe that the exact semantics of try_cast() should be implementation-defined, since run-time information could differ between implementations. For example, depending on the runtime's knowledge of data execution prevention, it might either allow or reject the cast of a non-executable

Listing 14 By using count_varargs() and get_varargs() we can use variadics in a robust way

```c
double avg(int count, ...) {
  if (count == 0 || count != count_varargs())
    return 0;
  int sum = 0;
  for (int i = 0; i < count; i++) {
    int *arg = get_vararg(i, type(&sum));
    if (arg == NULL) return 0;
    else             sum += *arg;
  }
  return (double) sum / count;
}
```

char array filled with machine instructions to a function pointer. Further, different use cases exist, and a security-focused runtime might have more sources of run-time information and be more restrictive than a performance-focused runtime. For example, a traditional runtime would (for compatibility) allow dereferencing a hand-crafted pointer as long as it corresponds to the address of an object, while a security-focused runtime could disallow it. Thus, depending on the underlying runtime, compiler, and ABI, try_cast() can return different results.

3.4 Variadic Arguments

Our introspection interface provides macros to query the number of variadic arguments and enables programmers to access them in a type-safe way. They are implemented as macros and not as functions, since they need to access the current function's variadic arguments. The introspection macros make using variadic functions more robust and are, for example, effective in preventing format string attacks [8].

Querying the number of variadic arguments can be achieved by calling count_varargs(). The standard va_arg() macro reads values from the stack while assuming that they correspond to the user-specified type. As a robust alternative, introspection composites can use _get_vararg() to access the passed variadic arguments directly by an argument index. To access the variadic arguments in a type-safe way, we introduced the get_vararg() macro, which is exposed to the programmer and expects a type that it uses to call try_cast(). Listing 14 shows an example of a function that computes the average of int arguments. It uses count_varargs() to verify the number of variadic arguments and ensures that the i^{th} argument is in fact an int by calling get_vararg() with type(&sum). If an unexpected number of parameters or an object with an unexpected type is passed, the function returns 0.

For backwards compatibility, we used the introspection intrinsics to make the standard vararg macros (va_start(), va_arg(), and va_end()) more robust. Firstly, va_start() initializes the retrieval of variadic arguments. We modified it such that it allocates a struct (using the alloca() stack allocation function) and populates it using _get_vararg() and count_varargs(). The struct comprises the number of variadic arguments, an array of addresses to the variadic arguments, and a counter to index them. Secondly, va_arg() retrieves the next variadic argument. We modified it such that it checks that the counter does not exceed the number of arguments, increments the counter, in-

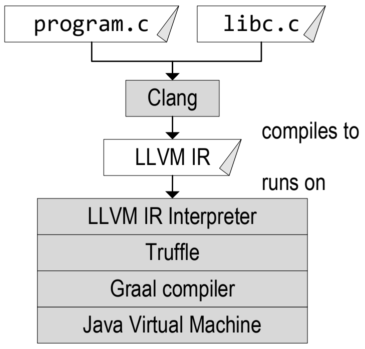

Figure 2 Overview of Safe Sulong

dexes the array, and casts the variadic argument to the specified type using try_cast(). If the cast succeeds, the argument is returned; otherwise a call to abort() exits the program. Finally, va_end() performs a cleanup of the data initialized by va_start(). We modified it such that it resets the variadic arguments counter.

Using the enhanced vararg macros improves the robustness of the xbuf_format_converter() function (see Listing 3), since the number of format specifiers must match the number of arguments, thus making it impossible to exploit the function through format string attacks. Note that the modified standard macros abort when they process invalid types or an invalid number of arguments, whereas the intrinsic functions allow programmers to react to invalid arguments in other ways.

4 Implementation

We implemented the introspection primitives in Safe Sulong [32], which is an execution system and bug-finding tool for low-level languages such as C. At its core is an interpreter written in Java that runs on top of the JVM. Although this setup is not typical for running C, it is a good experimentation platform because the JVM (and thus also Safe Sulong) already maintains all the run-time metadata that we want to expose. If exposing introspection primitives turns out to be useful for Safe Sulong, similar mechanisms could also be implemented for other runtimes (e.g., those of static compilation approaches). Unlike its counterpart Native Sulong [33], Safe Sulong uses Java objects to represent C objects. By relying on Java's bounds and type checks, Safe Sulong efficiently and reliably detects out-of-bounds accesses, use-after-free, and invalid free. When detecting such an invalid action, it aborts execution of the program. Section 4.1 gives an overview of the system, and Section 4.3 describes how we implemented the introspection primitives.

4.1 System Overview

Figure 2 shows the architecture of Safe Sulong, which comprises the following components:

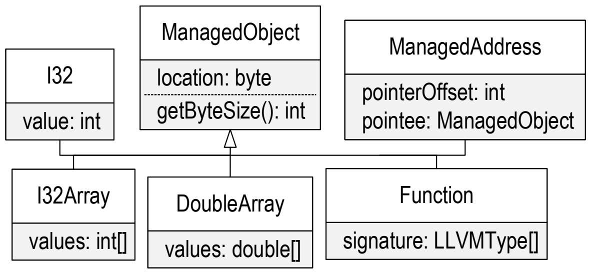

Figure 3 Diagram of the ManagedObject Hierarchy

Clang. Safe Sulong executes LLVM Intermediate Representation (IR), which represents C functions in a simpler, but lower-level format. LLVM is a flexible compilation infrastructure [25], and we use LLVM's front end Clang to compile the source code (libraries and the user application) to the IR.

LLVM IR. LLVM IR retains all C characteristics that are important for the content of this paper. It can, for instance, contain external function definitions and function calls. By executing LLVM IR, Safe Sulong can execute all languages that can be compiled to this IR, including C++ and Fortran. Using binary translators that convert binary code to LLVM IR even allows programs to be executed without access to their source code. For example, MC-Semantics [10] and QEMU [6] support x86, and LLBT [41] supports the translation of ARM code. Binary libraries that are converted to LLVM IR can then profit from the enhanced libraries that Safe Sulong can execute, such as our enhanced libc.

Truffle. The LLVM IR interpreter is based on Truffle [53]. Truffle is a language implementation framework written in Java. To implement a language, a programmer writes an Abstract Syntax Tree (AST) interpreter in which each operation is implemented as an executable node. Nodes can have children that parent nodes can execute to compute their results.

Graal. Truffle uses Graal [54], a dynamic compiler, to compile frequently executed Truffle ASTs to machine code. Graal applies aggressive optimistic optimizations based on assumptions that are later checked in the machine code. If an assumption no longer holds, the compiled code *deoptimizes* [20], that is, control is transferred back to the interpreter and the machine code of the AST is discarded.

LLVM IR Interpreter. The LLVM IR interpreter forms the core of Safe Sulong; it executes both the user application and the enhanced libc. First, a front end parses the LLVM IR and constructs a Truffle AST for each LLVM IR function. Then, the interpreter starts executing the main function AST, which can invoke other ASTs. During execution, Graal compiles frequently executed functions to machine code.

JVM. The system can run efficiently on any JVM that implements the Java-based JVM compiler interface (JVMCI [36]). JVMCI supports Graal and other compilers written in Java.

4.2 Introspection Primitives and Other Functions

While the majority of Safe Sulong's libc is implemented in C, the introspection primitives (and a core API, similar to system calls) are implemented directly in Java. Both are ultimately represented using executable ASTs, which are stored in a symbol table created prior to program execution. For functions contained in the LLVM IR file, the parser constructs the AST nodes from the instructions denoted in the LLVM IR function. For introspection primitives, we implemented special nodes that have no equivalent bitcode instruction (see Section 4.3). During execution, Safe Sulong looks up the AST in the symbol table using the function name. From the runtime's perspective, the implementation of that function is transparent.

4.3 Objects and Introspection

The LLVM IR interpreter uses Java objects instead of native memory to represent LLVM IR objects (and thus C objects). Figure 3 illustrates its type hierarchy. Every LLVM IR object is a ManagedObject which has subclasses for the different types. For example, an int is represented by an I32 object, which stores the int's value in the value field. Similarly, there are subclasses for arrays, functions, pointers, structs, and other types. Note that we have previously described a similar object hierarchy for the implementation of a *Lenient C* dialect and how certain corner cases are supported (e.g., deriving pointers from integers) [34]. In the introspection implementation, we needed to expose properties of these Java objects to the programmer:

Bounds. The ManagedObject class provides the method getByteSize(), which returns the size of an object. Safe Sulong represents pointers as objects of a ManagedAddress class that holds a reference to the pointee and a pointer offset that is updated through pointer arithmetics (pointee and pointerOffset). For example, for the pointer to the 4th element of an integer array in Listing 6, the pointerOffset is 16, and pointee references an I32Array that holds a Java int array (see Figure 4). If a program were to dereference the pointer, the interpreter would compute pointerOffset / sizeof(int) to index the array. We implemented the size_right() function by ptr.pointee.getByteSize() - ptr.pointerOffset.

Memory location. Although ManagedObjects live on the Java heap, the location() function needs to return their *logical* memory location. This location is stored in a field of the ManagedObject class. Depending on whether an object is allocated through malloc(), as a global variable, as a static local variable, or as a constant, we assign a different flag to its location field; calls to free() and deallocation of automatic variables assign INVALID. For instance, for an integer array that lives on the stack, the interpreter allocates an I32Array and assigns AUTOMATIC to its location. After leaving the function scope, its location is updated to INVALID. When the location() function is called with a pointer to the integer array, it returns the location field's value.

Type. For implementing the try_cast() function, we check if the type of the passed object (given by its Java class) is compatible with the type specified by the Type struct. For example, to check whether we can call a pointer as a function with a certain signature, we first compare the passed pointer with a Type that describes this signature.

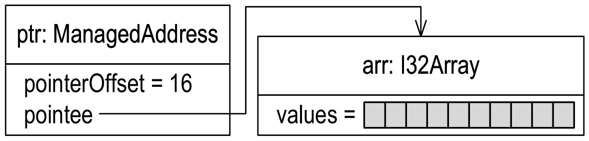

Figure 4 Representation of a pointer to the 4th element of an int array

If the pointer references a Safe Sulong object of type Function, the argument and return types are compared. This is possible because Function objects retain run-time information about their arguments and return types, which can be retrieved via the method getSignature().

Variadic arguments. In Safe Sulong, a caller explicitly passes its arguments as an object array (i.e., a Java array of known length) to its callee. Based on the function signature and the object array, the callee can count the variadic arguments to implement count_varargs() and extract them to implement _get_vararg().

5 Case Study: Safe Sulong's Standard Library

We implemented an enhanced libc for Safe Sulong. This libc uses introspection for checks that make it more robust against usage errors and attacks. For instance, its functions identify invalid parameters that would otherwise cause out-of-bounds accesses or use-after-frees. In such a case, the functions return special values to indicate that something went wrong, and then set errno to an error code. However, for functions for which no special value can be returned (e.g., because the return type is void), setting errno would be meaningless, since functions are allowed to change errno arbitrarily even if no error occurred. In these cases, the functions still attempt to compute a meaningful result. Such behavior is compliant with the C standards, since we prevent illegal actions with undefined behavior that could crash the program or corrupt memory.

For applications and libraries that run on Safe Sulong, the distribution format is LLVM IR and not executable code. Our standard library improvements are binary-compatible at the IR level, which means that users do not have to recompile their applications when using our enhanced libc. In addition, this standard library is source-compatible, so a user is not required to change the program when using it. Below, we give an overview of our enhanced library functions:

String functions. We made all functions that operate on strings (strlen(), atoi(), strcmp(), printf(), etc.) more robust by computing meaningful results even when a string lacks a null terminator. They do not read or write outside the boundaries of unterminated strings, which makes them robust against common string vulnerabilities. The functions increase availability of the system, since unterminated strings passed to libc do not cause crashes. Note that when a function outside libc relies on a terminated string, it will still trigger an out-of-bounds access and cause Safe Sulong to abort execution.

Listing 15 Robust implementation of strlen() that also works for unterminated strings

```
size_t strlen(const char *str) {
  size_t len = 0;
  while (size_right(str) > 0 && *str != '\0') {
    len++; str++;
  }
  return len;
}
```

Thus, increased availability does not harm confidentiality (e.g., by leaking data of other objects) and integrity (e.g., by overwriting other objects).
For instance, Listing 15 shows how we improved strlen() by preventing buffer overflows when iterating over a string, and by improving the handling of non-legal pointers (where size_right() returns -1). For terminated strings, strlen() iterates until the first '\0' character to return the length of the string. For unterminated strings, the function cannot return -1 to indicate an error, since size_t is unsigned, so we also do not set errno. Instead, it iterates until the end of the buffer and returns the size of the string until the end of the buffer.
The enhanced string functions also allow execution of the code fragment in Listing 5. Even though the source string may be unterminated, strcpy() will not produce an out-of-bounds read, since it stops copying when reaching the end of the source or destination buffer. The call to puts() also works as expected, and prints the unterminated string.

Functions that free memory. We made functions that free memory (realloc() and free()) more robust by checking whether their argument can safely be freed using freeable(). In Safe Sulong, malloc() is written in Java and allocates a Java object. By using the introspection functions we were able to conveniently and robustly implement realloc() in C without having to maintain a list of allocated and freed objects.

Format string functions. We made input and output functions that expect format strings more robust. Examples are the printf() functions (printf(), fprintf(), sprintf(), vfprintf(), vprintf(), vsnprintf(), vsprintf()) and the scanf() functions (scanf(), fscanf(), etc.). These functions expect format strings that contain format specifiers, and matching arguments that are used to produce the formatted output. Since the functions are variadic, we used count_varargs() to add checks that verify that the number of format specifiers is equal to the actual number of arguments. Further, the functions use get_vararg() to verify the argument types. This prevents format-string vulnerabilities and out-of-bounds reads in the format string, as demonstrated in the implementation of strlen().

Higher-order functions. We enhanced functions that receive function pointers such as qsort() and bsearch(). Listing 16 shows how qsort() can use try_cast() to verify that f is a function pointer that is compatible with the specified signature. Furthermore, the functions verify that no memory errors, such as buffer overflows, can occur.

gets() and gets_s(). While C11 replaced the gets() function with gets_s(), Safe Sulong can still provide a robust implementation for gets() (see Listing 17). Since size_right() can determine the size of the buffer to the right of the pointer, we can call it and use the returned size as an argument to the more robust gets_s() function. If the pointer

Listing 16 Robust qsort() implementation that checks whether it can call the supplied function pointer

```c
void qsort(void *base, size_t nitems, size_t size, int (*f)(const void *, const void*)) ↩
    ↪ {
  int (*verifiedPointer)(const void *, const void*) = try_cast(&f, type(f));
  if (size_right(base) < nitems * size || verifiedPointer == NULL) errno = EINVAL;
  else {
    // qsort implementation
  }
}
```

Listing 17 Robust implementation of gets() that uses the more robust gets_s() in its implementation

```c
char *gets(char *str) {
  int size = size_right(str);
  return gets_s(str, size == -1 ? 0 : size);
}
```

Listing 18 Robust implementation of gets_s() that verifies the passed size argument

```c
char *gets_s(char *str, rsize_t n) {
  if (size_right(str) < (long) n) {
    errno = EINVAL; return NULL;
  } else {
    // original code
  }
}
```

is not legal, we pass 0, which gets_s() handles as an error. We also made gets_s() more robust against erroneous parameters (see Listing 18). By using size_right() we can validate that the size parameter n is at least as large as the remaining space right of the pointer. The check prevents buffer overflows for gets() and gets_s(), and also passing of dead stack memory or freed heap memory.

6 Related Work

C Memory safety approaches. For decades, academia and industry have been coming up with approaches to tackling memory errors in C. Thus, there is a vast number of approaches that deal with these issues, both static and run-time approaches, both hardware- and software-based. We consider our approach as a run-time approach, since the checks (specified by programmers in their programs) are executed during run time. The literature provides a historical overview of memory errors and defense mechanisms [48], an investigation of the weaknesses of current memory defense mechanisms including a general model for memory attacks [46], and a survey of vulnerabilities and run-time countermeasures [55]. Using introspection to prevent memory errors is a novel approach that is complementary to existing approaches because the programmer can check for and prevent an invalid action; if the check is omitted and an invalid access occurs, an existing memory safety solution could still prevent the access.

Run-time types for C. libcrunch [22] is a system that detects type-cast errors at run time. It is based on liballocs [23], a run-time system that augments Unix processes with allocation-based types. libcrunch provides an __is_a() introspection function that exposes the type of an object. It uses this function to validate type casts and issues a warning on unsound casts. In contrast to our approach, libcrunch checks for invalid casts automatically, so the __is_a() function is not exposed to the programmer, nor are there other introspection functions. However, we believe that the system could be extended to provide additional run-time information that could be used to implement the introspection primitives. Typical overheads of collecting and using the type information are between 5-35%, which demonstrates that introspection functions are feasible in static compilation approaches.

Failure-oblivious computing. Failure-oblivious computing [35] is a technique that enables servers to continue their normal execution path in the presence of memory errors. Instead of aborting the program, invalid writes are discarded, and for invalid reads values are manufactured. Note that this approach is automatic, since the compiler inserts checks and continuation code where memory errors can occur. Failure-oblivious computing would, for example, work well for strlen by manufacturing the value zero when the NULL terminator is missing and the read runs over the buffer end. However, returning zero for out-of-bounds accesses does not work in general; for example, when the loop's exit condition checks if the array element is -1, failure-oblivious computing approaches could run into an endless loop. In contrast, using our introspection technique, programmers can take into account the semantics of a function to prevent such situations. Additionally, introspection can also be used for bug-finding (not only to increase availability), for example, by checking if the actual buffer length corresponds to the expected buffer length in functions like gets_s.

Static vulnerability scanners. Static vulnerability scanners identify calls to unsafe functions such as gets() depending on a policy specified in a vulnerability database [49]. Such approaches must decide conservatively whether a call is allowed, unlike our approach, which validates parameters at run-time through introspection. Nowadays, most compilers issue a warning when they identify a call to an unsafe function such as gets(), but not necessarily for other, slightly safer functions, such as strcpy().

Fault injection to increase library robustness. Fault injection approaches generate a series of test cases that exercise library functions in an attempt to trigger a crash in them. HEALERS [13, 14] is an approach that, after identifying a non-robust function, automatically generates a wrapper that sits between the application and its shared libraries to handle or prevent illegal parameters. To check the bounds of heap objects passed to the functions, the approach instruments malloc() and stores bounds information. In contrast to our solution, the approaches above support pre-compiled libraries. However, they can generate wrapper checks only where run-time information is explicitly available in the program. Additionally, they prevent the programmer from specifying the action in case of an error, and always set *errno* and return an error code.

Detecting API misusages. APISan [56] is a tool for finding API usage errors, such as cryptographic protocol API misues, but also integer overflows, NULL dereferences, memory leaks, incorrect return values, format string vulnerabilities, and wrong arguments. It is based on the idea that the dominant usage pattern of an API across several projects indicates its correct use. APISan is implemented by gathering execution traces using symbolic execution, from which it infers correct API usages; deviating patterns are potential API misuses. While this approach aims to identify incorrect use of libraries, our approach aims to make library functions more robust.

Replacing (parts of) libc. SFIO [24] is a libc replacement and addresses several of its problems. It mainly improved completeness and efficiency, but it also introduced safer routines for functions that operate on format strings. Additionally, the SFIO standard library functions are more consistent in their arguments and argument order, and thus less error-prone than some of the libc functions. In [28], the less error-prone strlcpy() and strlcat() functions were presented as replacements for the strcpy() and strncat() functions. Unlike our improved C standard library, these approaches lack source compatibility.

Safer implementation of library functions. To prevent format string vulnerabilities in the printf family of functions, FormatGuard [8] uses the preprocessor to count the arguments to variadic functions during compile time and checks that the number complies with the actual number at run time. FormatGuard replaces the printf functions in the C standard library with more secure versions while retaining compatibility with most programs. From a user perspective, FormatGuard is similar to Safe Sulong's standard library, in that both provide more robust C standard library functions. While our approach works only for runtimes that implement the introspection primitives, StackGuard works for arbitrary compilers and runtimes. However, our approach can also verify bounds, memory location, and types of objects.

Restricting buffer overflows in library functions. Libsafe [2] replaces calls to unsafe library functions (such as strcpy() and gets()) with wrappers that ensure that potential buffer overflows are contained within the current stack frame. It can prevent only stack buffer overflows, since it checks that write accesses do not extend beyond the end of the buffer's stack frame. In contrast, approaches exist that protect only against heap buffer overflows caused by C standard library functions [15]. By intercepting C standard library calls, the approach keeps track of heap memory allocations and performs bounds checking before calling the C standard library functions that operate on buffers. Both approaches work with any existing pre-compiled library, but do not protect against all kinds of buffer overflows. With our approach, a programmer can implement checks that prevent both heap and stack overflows, and use the introspection interface to also prevent use-after-free and other errors.

Reflection for C. Higher-level languages such as Java or C# throw exceptions when encountering out-of-bounds accesses and other errors. Exception handling is a more expressive approach than explicitly checking for invalid accesses in advance, since it separates the two concerns in the program. Some approaches introduced mechanisms to raise and catch exceptions in C [16, 26]. However, these approaches do not describe how invalid memory errors could be caught and exposed to the programmer as an exception.

7 Discussion

Advantages over existing tools. We assume that introspection is exposed by a runtime that automatically aborts when detecting an error (e.g., an out-of-bounds access). In this scenario, using introspection allows programmers to override the default behavior of aborting the program by checking for invalid states and by reacting to them before the failure occurs. Even if checks are omitted, the runtime aborts execution in case of an error. Additionally, introspection can be used to check for faults that might not result in errors during run time. While adding these checks does not come for free (i.e., they require programming effort), we believe that they can be useful at boundaries of shared libraries, and at the boundaries of subcomponents within a project.

Adoption of introspection. Two of the C/C++ tenets are that "you don't pay for what you don't use" [45] and to "trust the programmer" [5]. Hence, programmers often eschew checks even if they are possible without introspection functions [14]. An open question is thus whether C programmers would use introspection if they had access to it. We believe that there is a need for the safe execution of legacy C code (at the expense of performance) as an alternative to porting programs to safer languages.
It has yet to be determined which of the introspection functions are useful in practice (e.g., by conducting a case study on real-world programs). We believe that functions such as size_right() are easy to understand and use, and could prevent common errors in practice. In contrast, grasping the semantics of try_cast() is more difficult because C does not have a strong notion of typing, and use cases for it are also rare; consequently, it would probably be used less often.

Safer languages. Since using introspection requires changes to the source code, a question is whether a library should not simply be rewritten in some other systems programming language, such as Rust or Go, that approach the performance of C while being safe. First, preventing out-of-bounds accesses or use-after-free errors can already be prevented by using special runtimes without rewriting the project in a safer language (e.g., using AddressSanitizer [38] or SoftBound+CETS [29, 30]). However, our approach goes beyond these guarantees by allowing the programmer to handle errors in customized ways. Second, the effort required to rewrite an application would simply be too high for many real-world applications. In contrast, incrementally adding checks to an existing code base is less work.

Legacy code. Our approach also brings benefits for legacy applications, namely when a commonly used shared library is modified to employ introspection for additional checks: For example, there are legacy applications that use the insecure gets() libc function. Using our approach, a safe implementation of gets() can be provided if the runtime implements the introspection interface and libc uses it to query the length of the buffer. Thus, availability or security of legacy code can be improved simply by employing a libc that inserts additional checks enabled by introspection. In contrast, when reimplementing libc in a safer language, the function gets() cannot be made safe, as a buffer allocated by C code has no bounds information attached to it.

Static compilation. Introspection requires information about run-time properties of objects in the program. While interpreters and virtual machines often maintain this information, runtimes that execute native programs compiled by static compilers such as Clang or GCC do not. We want to point out that debug metadata (obtained by compiling with the -g flag) cannot provide per-object type information needed for introspection. However, it has been shown that per-object information (such as types) can be added add low cost to static compilation approaches [22] and hence make implementing the introspection functions in their runtimes feasible. As part of future work, we intend to implement introspection primitives using tools based on a static compilation model.

Partial metadata availability. While designing the interface, we assumed that a tool that implements introspection maintains all relevant metadata. However, some runtimes maintain only a subset of it; for example, bounds checkers track bounds information and can implement only _size_left() and _size_right(). Custom memory allocators that track heap allocations can implement only a subset of the function location(). It has yet to be investigated how code can benefit from runtimes that implement only parts of the interface. A compile-time approach would involve checking introspection features using preprocessor directives. Another approach would involve structuring the checks such that they do not fail when an introspection function returns a default value that indicates that the corresponding feature is unsupported.

Performance measurement. The focus of this work was on evaluating the usefulness of exposing introspection functions to library writers. We did not invest much time in optimizing the peak performance of our approach in Safe Sulong. Thus, we show its performance only on a small set of microbenchmarks for which we used our enhanced libc (see Appendix A). As part of future work, we want to extend Safe Sulong's completeness to execute larger benchmarks, such as SPEC INT [7].

8 Conclusion

We have presented an introspection interface for C that programmers can use to make libraries more robust. The introspection functions expose properties of objects (bounds, memory location, and type) as well as properties of variadic functions (number of variadic arguments and their types). We have described an implementation of the introspection primitives in Safe Sulong, a system that provides memory-safe execution of C code. However, our approach is not restricted to Safe Sulong; many dynamic bug-finding tools and runtimes exist that could implement (a subset of) the introspection interface. The approach is complementary to existing memory safety approaches, as programmers can use it to react to and prevent errors in the application logic. Finally, we have shown how we used the introspection interface to implement an enhanced, source-compatible C standard library.

Acknowledgements We thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. We also thank Ingrid Abfalter, whose proofreading and editorial assistance greatly improved the manuscript. We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute of System Software at Johannes Kepler University Linz for their support and contributions.

References

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[2] Arash Baratloo, Navjot Singh, and Timothy Tsai. "Libsafe: Protecting critical elements of stacks". In: *White Paper* (1999). http://www.research.avayalabs.com/project/libsafe.

[3] Priyam Biswas, Alessandro Di Federico, Scott A. Carr, Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, Michael Franz, and Mathias Payer. "Venerable Variadic Vulnerabilities Vanquished". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/biswas.

[4] Derek Bruening and Qin Zhao. "Practical Memory Checking with Dr. Memory". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pages 213–223. ISBN: 978-1-61284-356-8.

[5] C99 Committee. *Rationale for International Standard–Programming Languages–C. Revision 5.10*. 2003.

[6] Vitaly Chipounov and George Candea. *Dynamically Translating x86 to LLVM using QEMU*. Technical report. École polytechnique fédérale de Lausanne, 2010.

[7] Standard Performance Evaluation Corporation. *CINT2006 (Integer Component of SPEC CPU2006)*. Accessed October 2017. URL: https://www.spec.org/cpu2006/CINT2006/.

[8] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Michael Frantzen, and Jamie Lokier. "FormatGuard: Automatic Protection From printf Format String Vulnerabilities." In: *USENIX Security Symposium*. Volume 91. Washington, DC. 2001. URL: https://www.usenix.org/legacy/events/sec01/cowanbarringer.html.

[9] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. "Buffer overflows: attacks and defenses for the vulnerability of the decade". In: *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*. Volume 2. 2000, 119–129 vol.2. DOI: 10.1109/DISCEX.2000.821514.

[10] Artem Dinaburg and Andrew Ruef. "Mcsema: Static translation of x86 instructions to llvm". In: *ReCon 2014 Conference, Montreal, Canada*. 2014.

[11] Frank Ch Eigler. “Mudflap: Pointer Use Checking for C/C+”. In: *Proceedings of the First Annual GCC Developers’ Summit* (2003), pages 57–70.

[12] Common Weakness Enumeration. *CWE-242: Use of Inherently Dangerous Function*. Accessed July 2017. 2017. URL: https://cwe.mitre.org/data/definitions/242.html.

[13] Christof Fetzer and Zhen Xiao. “A flexible generator architecture for improving software dependability”. In: *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.* 2002, pages 102–113. DOI: 10.1109/ISSRE.2002.1173221.

[14] Christof Fetzer and Zhen Xiao. “An Automated Approach to Increasing the Robustness of C Libraries”. In: *Proceedings of the 2002 International Conference on Dependable Systems and Networks*. DSN ’02. Washington, DC, USA: IEEE Computer Society, 2002, pages 155–166. ISBN: 0-7695-1597-5.

[15] Cristof Fetzer and Zhen Xiao. “Detecting heap smashing attacks through fault containment wrappers”. In: *Proceedings 20th IEEE Symposium on Reliable Distributed Systems*. 2001, pages 80–89. DOI: 10.1109/RELDIS.2001.969756.

[16] Narain Gehani. “Exceptional C or C with exceptions”. In: *Software: Practice and Experience* 22.10 (1992), pages 827–848. ISSN: 1097-024X. DOI: 10.1002/spe.4380221003.

[17] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. “Memory-safe Execution of C on a Java VM”. In: *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*. PLAS’15. Prague, Czech Republic: ACM, 2015, pages 16–27. ISBN: 978-1-4503-3661-1. DOI: 10.1145/2786558.2786565.

[18] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. “TypeSan: Practical Type Confusion Detection”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: ACM, 2016, pages 517–528. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978405.

[19] Reed Hastings and Bob Joyce. “Purify: Fast detection of memory leaks and access errors”. In: *In proc. of the winter 1992 usenix conference*. Citeseer. 1991.

[20] Urs Hölzle, Craig Chambers, and David Ungar. “Debugging Optimized Code with Dynamic Deoptimization”. In: volume 27. 7. New York, NY, USA: ACM, July 1992, pages 32–43. DOI: 10.1145/143103.143114.

[21] IEEE. “Systems and software engineering – Vocabulary”. In: *ISO/IEC/IEEE 24765:2010(E)* (Dec. 2010), pages 1–418. DOI: 10.1109/IEEESTD.2010.5733835.

[22] Stephen Kell. “Dynamically Diagnosing Type Errors in Unsafe Code”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. Amsterdam, Netherlands: ACM, 2016, pages 800–819. ISBN: 978-1-4503-4444-9. DOI: 10.1145/2983990.2983998.

[23] Stephen Kell. “Towards a Dynamic Object Model Within Unix Processes”. In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* Onward! 2015. Pittsburgh, PA, USA: ACM, 2015, pages 224–239. ISBN: 978-1-4503-3688-8. DOI: 10.1145/2814228.2814238.

[24] David G. Korn and Kiem-Phong Vo. “SFIO: Safe/Fast String/File IO”. In: *Proceedings of the Summer 1991 USENIX Conference, Nashville, TE, USA, June 1991*. 1991, pages 235–256.

[25] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO ’04. Palo Alto, California: IEEE Computer Society, 2004, pages 75–. ISBN: 0-7695-2102-9.

[26] Peter A. Lee. “Exception handling in C programs”. In: *Software: Practice and Experience* 13.5 (1983), pages 389–405. ISSN: 1097-024X. DOI: 10.1002/spe.4380130502.

[27] Kai Lu. *Analysis of CVE-2016-0059 - Microsoft IE Information Disclosure Vulnerability Discovered by Fortinet*. Accessed July 2017. 2016. URL: https://blog.fortinet.com/2016/02/19/analysis-of-cve-2016-0059-microsoft-ie-information-disclosure-vulnerability-discovered-by-fortinet.

[28] Todd C. Miller and Theo de Raadt. “Strlcpy and Strlcat: Consistent, Safe, String Copy and Concatenation”. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’99. Monterey, California: USENIX Association, 1999, pages 41–41. URL: https://www.usenix.org/legacy/event/usenix99/millert.html.

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. “CETS: Compiler Enforced Temporal Safety for C”. In: *SIGPLAN Not.* 45.8 (June 2010), pages 31–40. ISSN: 0362-1340. DOI: 10.1145/1837855.1806657.

[30] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. “SoftBound: Highly Compatible and Complete Spatial Memory Safety for C”. In: *SIGPLAN Not.* 44.6 (June 2009), pages 245–258. ISSN: 0362-1340. DOI: 10.1145/1543135.1542504.

[31] Nicholas Nethercote and Julian Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: ACM, 2007, pages 89–100. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250746.

[32] Manuel Rigger. “Sulong: Memory Safe and Efficient Execution of LLVM-Based Languages”. In: *ECOOP 2016 Doctoral Symposium*. Rome, Italy, 2016.

[33] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. "Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle". In: *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*. VMIL 2016. Amsterdam, Netherlands: ACM, 2016, pages 6–15. ISBN: 978-1-4503-4645-0. DOI: 10.1145/2998415.2998416.

[34] Manuel Rigger, Roland Schatz, Matthias Grimmer, and Hanspeter Mössenböck. "Lenient Execution of C on a Java Virtual Machine or: How I Learned to Stop Worrying and Run the Code". In: *Proceedings of the 14th International Conference on Managed Languages and Runtimes*. ManLang 2017. Prague, Czech Republic: ACM, 2017, pages 35–47. ISBN: 978-1-4503-5340-3. DOI: 10.1145/3132190.3132204.

[35] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee Jr. "Enhancing Server Availability and Security Through Failure-oblivious Computing". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pages 21–21. URL: https://www.usenix.org/legacy/events/osdi04/tech/rinard.html.

[36] John Rose. *JEP 243: Java-Level JVM Compiler Interface*. Accessed July 2017. 2014. URL: http://openjdk.java.net/jeps/243.

[37] SANS. *CWE/SANS TOP 25 Most Dangerous Software Errors*. Accessed July 2017. 2011. URL: https://www.sans.org/top25-software-errors/.

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, pages 28–28. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[39] Julian Seward and Nicholas Nethercote. "Using Valgrind to Detect Undefined Value Errors with Bit-precision". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, pages 2–2. URL: https://www.usenix.org/legacy/events/usenix05/tech/general/seward.html.

[40] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. "Detecting Format String Vulnerabilities with Type Qualifiers". In: *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*. SSYM'01. Washington, D.C.: USENIX Association, 2001. URL: https://www.usenix.org/legacy/events/sec01/shankar.html.

[41] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. "LLBT: An LLVM-based Static Binary Translator". In: *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '12. Tampere, Finland: ACM, 2012, pages 51–60. ISBN: 978-1-4503-1424-4. DOI: 10.1145/2380403.2380419.

[42] shootouts. *The Computer Language Benchmarks Game*. Accessed July 2017. URL: http://benchmarksgame.alioth.debian.org/.

[43] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. "An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance". In: *Proceedings of the 4th Workshop on Scala*. SCALA '13. Montpellier, France: ACM, 2013, 9:1–9:8. ISBN: 978-1-4503-2064-1. DOI: 10.1145/2489837.2489846.

[44] Evgeniy Stepanov and Konstantin Serebryany. "MemorySanitizer: fast detector of uninitialized memory use in C++". In: *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. San Francisco, CA, USA, 2015, pages 46–55. DOI: 10.1109/CGO.2015.7054186.

[45] Bjarne Stroustrup. *The Design and Evolution of C++*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1994. ISBN: 0-201-54330-3.

[46] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pages 48–62. ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.13.

[47] TIOBE. *TIOBE Index for July 2017*. Accessed July 2017. 2017. URL: http://www.tiobe.com/tiobe-index/.

[48] Victor van der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. "Memory Errors: The Past, the Present, and the Future". In: *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*. RAID'12. Amsterdam, The Netherlands, 2012, pages 86–106. ISBN: 978-3-642-33337-8. DOI: 10.1007/978-3-642-33338-5_5.

[49] John Viega, J. T. Bloch, Yoshi Kohno, and Gary McGraw. "ITS4: a static vulnerability scanner for C and C++ code". In: *Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference*. Dec. 2000, pages 257–267. DOI: 10.1109/ACSAC.2000.898880.

[50] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. "Undefined Behavior: What Happened to My Code?" In: *Proceedings of the Asia-Pacific Workshop on Systems*. APSYS '12. Seoul, Republic of Korea: ACM, 2012, 9:1–9:7. ISBN: 978-1-4503-1669-9. DOI: 10.1145/2349896.2349905.

[51] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. "Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pages 260–275. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522728.

[52] *Whetstone benchmark*. Accessed July 2017. URL: http://www.netlib.org/benchmark/whetstone.c.

[53] Christian Wimmer and Thomas Würthinger. "Truffle: A Self-optimizing Runtime System". In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. SPLASH '12. Tucson, Arizona, USA, 2012, pages 13–14. ISBN: 978-1-4503-1563-0. DOI: 10.1145/2384716.2384723.

[54] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. "One VM to Rule Them All". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: ACM, 2013, pages 187–204. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509581.

[55] Yves Younan, Wouter Joosen, and Frank Piessens. "Runtime Countermeasures for Code Injection Attacks Against C and C++ Programs". In: *ACM Comput. Surv.* 44.3 (June 2012), 17:1–17:28. ISSN: 0360-0300. DOI: 10.1145/2187671.2187679.

[56] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. "APISan: Sanitizing API Usages through Semantic Cross-Checking". In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pages 363–378. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun.

A Preliminary Performance Evaluation

The focus of this work was on evaluating the usefulness of exposing introspection functions to library writers. We have not yet invested much time in optimizing the peak performance of our approach in Safe Sulong. To demonstrate that Safe Sulong can run programs in a testing environment, we ran six benchmarks of the Computer Language Benchmark Game [42] (binarytrees, fannkuchredux, fasta, mandelbrot, nbody, and spectralnorm) and the whetstone benchmark [52], once with the enhanced `libc` and once without introspection checks. We determined the average peak performance of 10 runs by measuring the execution time after 50 in-process warm-up iterations. On these benchmarks, Safe Sulong's peak performance was 2.3× slower than executables compiled by Clang with all optimizations turned on (`-O3` flag). We were unable to find any observable performance differences between the two `libc` versions, which is in part due to some of the introspection checks redundantly duplicating automatic checks performed by the JVM (e.g., bounds checks); such redundant checks can be eliminated by using the Graal compiler (e.g., through conditional elimination [43]). As part of future work, we will evaluate Safe Sulong's performance in combination with the enhanced `libc` on larger benchmarks that stress the introspection functionality.

B Introspection Functions

Table 1 shows the functions and macros of the introspection interface. Internal functions that are private to the implementation are denoted with an underscore prefix.

Table 1 Functions and macros of the introspection interface

Object bounds functions		
long _size_right(void *)	Primitive internal	Returns the space in bytes from the pointer target to the end of the pointed object. This function is undefined for illegal pointers.
long _size_left(void *)	Primitive internal	Returns the space in bytes from the pointer target to the beginning of the pointed object. This function is undefined for illegal pointers.
long size_right(void *)	Composite	Returns the remaining space in bytes to the right of the pointer. Returns -1 if the pointer is not legal or out of bounds.
long size_left(void *)	Composite	Returns the remaining space in bytes to the left of the pointer. Returns -1 if the pointer is not legal or out of bounds.
Memory location functions		
Location location(void *)	Primitive	Returns the kind of the memory location of the referenced object. Returns -1 if the pointer is NULL.
bool freeable(void *)	Composite	Returns whether the pointer is freeable (i.e., DYNAMIC non-null memory; pointer referencing the beginning of an object).
Type functions		
void* try_cast(void *, struct Type *)	Primitive	Returns the first argument if the pointer is legal, within bounds, and the referenced object can be treated as of being of the specified type and NULL otherwise.
Variadic function macros		
int count_varargs()	Primitive	Returns the number of variadic arguments that are passed to the currently executing function.
void* _get_vararg(int i)	Primitive internal	Returns the ith variadic argument (starting from 0) and returns NULL if i is greater or equal to count_varargs().
void* get_vararg(int i, Type* type)	Composite	Returns the ith variadic argument (starting from 0) as the specified type. Returns NULL if the object cannot be treated as being of the specified type or if i is greater or equal to count_varargs().

Chapter 8

Context-aware Failure-oblivious Computing

This chapter includes a paper about a failure-oblivious computing mechanism that is based on our introspection ideas.

Paper: Manuel Rigger, Daniel Pekarek, and Hanspeter Mössenböck. Preventing Buffer Overflows by Context-aware Failure-oblivious Computing. In *Proceedings of the 12th International Conference on Network and System Security*, NSS 2018, 2018

Note: Daniel Pekarek contributed significant parts of the implementation described in this paper, based on the ideas of the thesis' author.

Context-aware Failure-oblivious Computing as a Means of Preventing Buffer Overflows[⋆]

Manuel Rigger[1], Daniel Pekarek[1], and Hanspeter Mössenböck[1]

Johannes Kepler University Linz, Austria
{manuel.rigger,daniel.pekarek,hanspeter.moessenboeck}@jku.at

Abstract. In languages like C, buffer overflows are widespread. A common mitigation technique is to use tools that detect them during execution and abort the program to prevent data leakage or the diversion of control flow. However, for server applications, it would be desirable to prevent such errors while maintaining availability of the system. To this end, we present an approach to handling buffer overflows without aborting the program. This approach involves implementing a recovery logic in library functions based on an introspection function that allows querying the size of a buffer. We demonstrate that introspection can be implemented in popular bug-finding and bug-mitigation tools such as LLVM's AddressSanitizer, SoftBound, and Intel-MPX-based bounds checking. We evaluated our approach in a case study of real-world bugs and show that for tools that explicitly track bounds data, introspection results in a low performance overhead.

Keywords: memory safety · reliability · dependability · availability · fault tolerance

1 Introduction

Buffer overflows in C, where an out-of-bounds pointer is dereferenced, belong to the most dangerous software errors [5,32]. Unlike higher-level languages, buffer overflows invoke *Undefined Behavior* and are not prevented during execution; programmers also cannot handle them using exception or similar mechanisms, since the language lacks them. Buffer overflows allow attackers to overflow function addresses stored on the stack or heap and thus to maliciously divert execution of the program [28] and to leak sensitive data [31]. A plethora of tools exist that make their exploitation more difficult or detect them and abort execution of the program [34,32,36,30]. However, when availability of an application is important (e.g. for production servers), it would be preferable to continue execution as long as security is not compromised [24]. This could, for example, make it harder to perform a denial-of-service attack where a buffer overflow is exploited to crash the program or inject code.

[⋆] We thank Oracle Labs for funding this research. We thank Gergö Barany, Roland Yap, and Fabio Niephaus for their useful feedback on an early draft of this paper. We thank Ingrid Abfalter for proofreading and editorial assistance.

To safely maintain execution in the presence of buffer overflows, we have come up with the concept of *context-aware failure-oblivious computing*. Our core idea is that library writers (e.g., the libc maintainers) can query run-time data such as bounds information in library functions by using an introspection interface. This information can then be used to implement a recovery logic that can mitigate incorrect execution states instead of aborting the program. Library writers can implement a custom recovery logic that depends on each function's semantics, which is why we refer to our technique as being context-aware. For example, a libc function that processes an unterminated string could prevent an out-of-bounds access by checking for the end of the buffer to handle the fault and continue execution. We expect that this recovery logic would be used mainly in a production context, as it would be preferable that execution is aborted if an error occurs during development and testing so that programmers can fix the error.

Our work is based on a combination of *failure-oblivious computing* [25] and our previous work on an introspection interface for C to increase the robustness of libraries [23]. We show how the introspection interface can be used to implement a failure-oblivious computing mechanism. We evaluated our approach by demonstrating that introspection for preventing buffer overflows can be implemented in popular bug-finding and bug-mitigation tools such as LLVM's AddressSanitizer [27], SoftBound [15], and GCC's Pointer Bounds Checker, which is based on the Intel Memory Protection Extensions (MPX) [19]. Furthermore, we show how our approach allows execution to continue in the presence of buffer overflows found in real-world programs as described by the Common Vulnerabilities and Exposures (CVE) database [33], and demonstrate that the performance overhead for introspection implemented in approaches such as MPX is negligible.

2 Background

Failure-oblivious computing. One technique for maintaining availability in the presence of buffer overflows is *failure-oblivious computing*, where invalid writes are discarded and values for invalid reads are manufactured [25,26]. By carefully selecting a sequence of return values for invalid reads, the program can successfully continue execution in most cases. However, a drawback of this approach is that it is "blind"; that is, it cannot guess the context (i.e., a function's semantics) to return a meaningful value for all reads. In this paper, we address this aspect by making failure-oblivious computing context-aware.

Introspection for C. As part of previous work, we demonstrated how use of introspection (i.e., exposing run-time data) benefits the robustness of libraries [23]. The core idea of our approach was that bug-finding tools and runtimes for C that track additional metadata such as object bounds or object types can expose this data to library writers via an introspection interface, which programmers can use to check the input of library functions. We showed that various introspection functions can be used to detect bugs or to maintain availability of the program. For example, to detect buffer overflows by means of introspection, the `_size_right()` function can be applied, which expects a pointer and returns the number of allocated bytes to the right of the pointee (or zero for invalid pointers) and can therefore be used for bounds checks. In this paper, we expand on how introspection can be used to increase availability, which we define as *context-aware failure-oblivious computing.*

Evaluation of introspection. We have previously evaluated an introspection libc using Safe Sulong [21,22], an LLVM IR interpreter on top of the Java Virtual Machine (JVM) [20] which automatically keeps track of array lengths, object sizes, and object types of C data [23]. Although the JVM tracks all relevant run-time information necessary to implement our introspection mechanism, it is not a typical environment in which to execute C code. In this paper, we address this by evaluating our approach in the context of popular bug-finding and bug-mitigation tools for buffer overflows and show that our refined introspection approach prevents real-world errors while maintaining availability.

3 Introspection Interceptors

This section explains the implementation of the introspection-based libc functions. These enhanced functions rely on the `_size_right()` introspection function to mitigate buffer overflows. Challenges to introducing them were that the original code not be cluttered by the introspection checks, that the effort for implementing these checks be low, and that the code behave in the same way as the original library during correct execution.

Libc interceptors. Based on our requirements, we implemented the introspection-based libc functions as *interceptors*, which are wrappers that intercept calls to libc functions and which are used by many bug-checking and bug-mitigation tools (including ASan, GCC's Pointer Bounds Checker and SoftBound)[1]. The introspection logic was kept separate from the normal code to avoid cluttering of the original source code. The cost of adding introspection-based recovery logic was low, as for each unsafe function that we considered (e.g., `strlen()`), libc provides safer functions that expect an additional size argument, which we used for our implementation (e.g., `strnlen()`). By reusing existing libc functions from the same library, we expect correct execution to behave in the same way as without the interceptors. For example, consider our `strlen()` interceptor, which is based on the safer `strnlen()` function:

```
size_t strlen(const char *s) {
  return ORIGINAL(strnlen)(s, _size_right(s));
}
```

[1] Note that in our previous work we instead reimplemented parts of a libc to use introspection, which made the libc less readable and required programs to be compatible with this libc.

The ORIGINAL macro yields a reference to the function passed as its argument that is part of the library and prevents recursively calling interceptors. We implemented the _size_right() function in various memory-safety-checking tools, as described in Section 4. Both the original strlen() implementation and this interceptor behave correctly for strings that are terminated with a '\0', which is needed to determine their length. However, if an unterminated string is passed to the original strlen() implementation, the function results in a buffer overflow that causes bug-finding and bug-mitigation tools to abort execution. Using the introspection-based interceptor instead prevents the buffer overflow, as the string length can be computed even for strings for which the '\0' is missing, because the interceptor assumes the underlying buffer size to be the maximum length of the string. Note that application-level functions can still cause bug-finding and bug-mitigation tools to abort execution if these functions run over string bounds. However, in many cases, application-level functions process strings up to the length computed by strlen(), which consequently prevents an out-of-bounds access.

As another example, an introspection interceptor can address the insecure interface of gets(), which reads user input and writes it to a buffer whose size is unknown to the function:

```
char *gets(char *s) {
  return ORIGINAL(fgets)(s, _size_right(s), stdin);
}
```

Using introspection, gets() reads only as much user input as the buffer can store.

Some introspection interceptors correct invalid parameters, for instance, in memcpy:

```
void *memcpy(void *dest, const void *src,
             size_t n) {
  ssize_t dstsz = _size_right(dest);
  size_t len = n;
  if (dstsz < len) {
    len = dstsz;
  }
  return ORIGINAL(memcpy)(dest, src, len);
}
```

If the size of the destination buffer is smaller than the number of bytes that the function is expected to copy, the function ignores the writes that go out of bounds. Note that another check for the size of the source buffer would be applicable.

In contrast to our previous work [23], we treat the return value of _size_right() as a conservative estimate of the object's right bounds. This estimate can be the real size of the object, in which case the introspection interceptors work most reliably. However, it can also be at least as large as the actual allocation, which could include additional space due to alignment requirements (e.g., to accommodate approaches that track only allocation sizes). Finally, if no bounds information is available for a given pointer, returning `MAX_LONG` effectively disables the introspection interceptors. This is useful, since it allows execution without recompilation of the code even when no tool is used that could determine the bounds of an object.

4 Introspection in Tools

We implemented `_size_right()` by exposing existing bounds information in three tools, namely LLVM's AddressSanitizer [27], SoftBound [15], and GCC's Intel MPX-based Pointer Bounds Checker instrumentation. SoftBound and LLVM's AddressSanitizer (ASan) are both software-based approaches. SoftBound provides access to bounds information in constant time, and is therefore a favorable candidate for implementing introspection. ASan's representation of metadata is suboptimal for implementing introspection, because it does not explicitly maintain bounds information and finding the end of an object takes linear time. By implementing introspection in ASan, we wanted to determine a worst-case overhead for implementing introspection in existing tools. Intel MPX instrumentation allowed us to additionally evaluate a hardware-based approach.

SoftBound. SoftBound is a bounds checker that has also been enhanced by a mechanism (called CETS) to find temporal memory errors [16]. It tracks base and bounds information for every pointer as separate metadata. To propagate this metadata across call sites, SoftBound adds additional base and bounds metadata to pointer arguments of functions. To implement `_size_right()`, we return the right bounds of a pointee by subtracting its base address from its bounds, which are associated with the pointer. For all SoftBound experiments, we used the latest stable version 3.8.0, which is distributed together with CETS.

LLVM's AddressSanitizer. ASan is one of the most widely used bug-finding tools for C/C++ programs; it allows memory errors such as buffer overflows and use-after-free errors to be found by instrumenting the program during compile time. Its implementation is based on *shadow memory* [17], where a memory cell allocated by the program has a corresponding shadow memory cell that stores meta-information about the original allocation. To detect buffer overflows, ASan allocates space between allocations and marks the corresponding shadow memory as *redzones*; if a dereferenced pointer points to such a redzone, ASan detects the overflow and aborts the program. Shadow memory is not a favorable representation of metadata for introspection, since bounds information cannot be accessed in constant time. We implemented `_size_right()` in linear time by iterating over the current buffer until its associated shadow memory indicates that a redzone has been reached. For all LLVM and ASan experiments, we used the development branch of LLVM version 6.0.0 based on commit 1d871d6 in compiler-rt.

Intel MPX. Intel MPX is an instruction set extension that adds instructions for creating, maintaining, and checking bounds information. Although its performance overhead is relatively high [19], providing buffer overflow protection at the hardware level is a promising research direction [35]. To use Intel MPX, we relied on GCC's Pointer Bounds Checker instrumentation, which employs Intel MPX to verify bounds. Similarly to SoftBound's implementation, we implemented `_size_right()` by querying the upper bounds (using a GCC builtin function) and subtracted the pointer address from it. For all experiments, we used GCC version 7.2.0.

Using libc. To use our introspection-based libc extensions, we redefined the names of the libc functions by means of preprocessor macros. While this required recompilation of the target application, it allowed the tools to also instrument our introspection-based libc functions and did not require us to maintain bounds information, as libc calls from our interceptors invoked the tools' interceptors. Note that our approach could be extended by using the dynamic loader to load the interceptors to retain binary compatibility (e.g., using the `LD_PRELOAD` mechanism); however, redefining the function names was less invasive.

5 CVE Case Study

To demonstrate the applicability of our approach in real-world projects, we considered recent (i.e., less than one year old) buffer overflows in widely-used software such as Dnsmasq, Libxml2, and GraphicsMagick. We selected the first libc-related bugs that we found in the CVE database for which an executable exploit existed. For each buffer overflow, we evaluated whether our introspection-based approach could mitigate the error and whether execution could successfully continue. Our approach prevented four out of five buffer overflows while successfully continuing execution; in one case, execution was aborted due to a subsequent buffer overflow in user-level code. Note that the unmodified tools also detected those buffer overflows; however, they aborted the program instead of mitigating the error and continuing execution. Since we performed this case study on complex real-world applications, and because SoftBound is a research prototype, we could not successfully execute any of these applications with it. The unmodified SoftBound version was also unable to execute them.[2] However, we extracted the functions in which the errors occurred, which SoftBound could execute, and created a driver to trigger the bug.

Dnsmasq. Dnsmasq is a lightweight DHCP server and caching DNS server which is used in many home routers.[3] In versions prior to 2.78, a bug existed that could cause a stack-based buffer overflow that allowed attackers to execute arbitrary code or to cause denial of service by crafting a DHCPv6 request with a wrong size (see CVE-2017-14493). It occurred in `memcpy()`, to which an incorrect size

[2] https://github.com/santoshn/softboundcets-3.8.0/issues/$x \in \{5, 6, 7, 8\}$
[3] `http://www.thekelleys.org.uk/dnsmasq/doc.html`

argument was passed:

```
state->mac_len = opt6_len(opt) - 2;
memcpy(&state->mac[0], opt6_ptr(opt, 2), state->mac_len);
```

A similar bug could be exploited for denial of service attacks (see CVE-2017-14496). It occurred in `memset()` and was triggered by an integer overflow:

```
/* Clear buffer beyond request to avoid risk of information disclosure. */
memset(((char *)header) + qlen, 0, (limit - ((char *)header)) - qlen);
```

When using our introspection interceptors, all tools continued execution by copying or setting up to as many bytes as the destination buffer could hold. The server stayed fully functional.

Libxml2. Libxml2 is a widely used open-source XML parsing library.[4] For versions up to 2.9.4, a vulnerability in the `xmlSnprintfElementContent()` function enabled attackers to crash the application through a buffer overflow (see CVE-2017-9047). It was caused by an incorrect length validation (at another code location) followed by `strcat()`:

```
if (content->name != NULL)
  strcat(buf, (char *) content->name);
```

The introspection interceptor for `strcat()` mitigated the buffer overflow by restricting the length of the concatenated string in all tools. The application continued execution and printed the truncated string as part of an error message. Although the error message was truncated, the output appeared reasonable from the user's point of view.

GraphicsMagick. GraphicsMagick is a widely used image processing tool.[5] In version 1.3.26, its `DescribeImage()` function allowed attackers to overflow and corrupt the heap to execute arbitrary code or to cause denial-of-service attacks (see CVE-2017-16352). As shown below, the size argument in the call to `strncpy()` did not limit the number of copied bytes to the size of the buffer; instead, the number was calculated by the length of the directory name (which was determined by searching for the newline or `NUL`). Consequently, an overly long directory name could be used to cause an overflow:

```
for (p=image->directory; *p != '\0'; p++) {
  q=p;
  while ((*q != '\n') && (*q != '\0'))
    q++;
  (void) strncpy(image_info->filename,p,q-p);
  image_info->filename[q-p]='\0';
  p=q;
  // ...
}
```

[4] `http://xmlsoft.org/`

[5] `http://www.graphicsmagick.org/`

The introspection interceptor for `strncpy()` successfully restricted the length of the copied string to the length of the destination buffer `image_info->filename`. However, in the line after the call to `strncpy()`, the program attempted to write a `NUL` character to the end of the string, which then caused an out-of-bounds access in the user application. The introspection approach does not protect against buffer overflows that happen in code that does not use introspection; however, we intend introspection to be used together with a bounds-checking tool, which is expected to abort execution for unhandled errors and thus prevent incorrect execution. In fact, all introspection-instrumented tools prevented this buffer overflow by aborting execution.

LightFTP. LightFTP is a small FTP server.[6] A logging function `writelogentry()` in version 1.1 of LightFTP was vulnerable to a buffer overflow that allowed denial of service or remote code execution (see CVE-2017-1000218). As shown below, the program added log entries to a buffer with a hard-coded size; as the log entries depended on user input that was restricted by another, larger constant, a buffer overflow could be triggered:

```
char _text[512];
// ...
if (logtext1)
    strcat(_text, logtext1);
if (logtext2)
    strcat(_text, logtext2);
strcat(_text, CRLF);
```

The introspection interceptor for `strcat()` mitigated the error without crashing the FTP server. Note that our mitigation truncated the log entry, but allowed subsequent requests to be handled successfully.

6 Performance Evaluation

To determine the performance of the introspection-based interceptors, we used LightFTP and Dnsmasq, which are the servers we also investigated in our CVE case study. We selected them for their high attack surface and because they are expected to be highly available. We evaluated the performance of ASan and Intel MPX both with and without the introspection interceptors; SoftBound failed to execute the servers, as explained above. Further, to establish a baseline, we measured the performance of C programs compiled with the Clang compiler [13] without using any bug-mitigation mechanisms. For all systems, we turned on compiler optimizations by using the `-O3` flag. We measured the throughput by means of the load-testing tool JMeter version 3.3. We configured JMeter to use 4 threads, each of which each sent 250 requests to simulate multiple concurrent users using the built-in FTP sampler and the UDP Protocol Support plugin. As the Intel MPX instructions are not thread-safe [19], we also evaluated all tools

[6] `https://github.com/hfiref0x/LightFTP/`

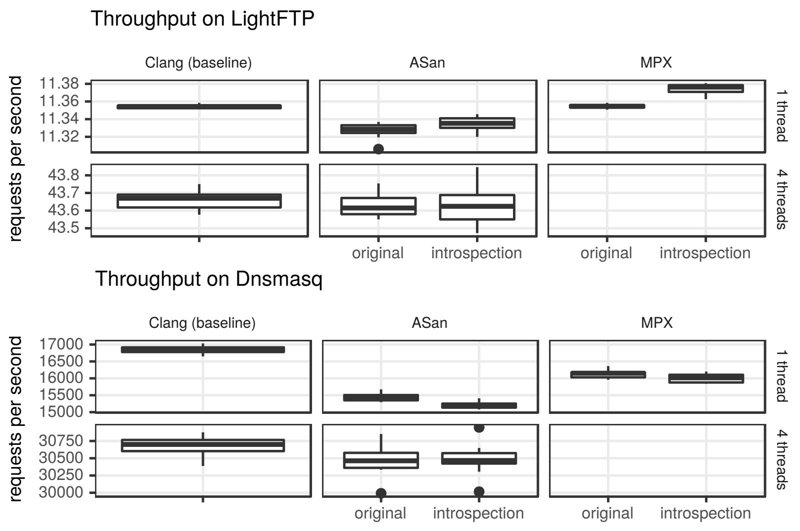

Fig. 1. Throughput on LightFTP and Dnsmasq.

using only 1 thread. We performed each measurement 10 times to account for variability. Our setup consisted of a quad-core Intel Core i7-6700HQ CPU at 2.60GHz on Ubuntu version 17.10 (with kernel 4.13.0-32-generic) with 16 GB of memory.

Figure 1 shows boxplots of the results for LightFTP and Dnsmasq. On LightFTP, the performance overhead for using introspection was below 1% for ASan; MPX was even slightly faster when introspection was used. On Dnsmasq, employing introspection caused a slowdown of around 1% when using only one thread for both ASan and MPX. The performance difference to the baseline was negligible on LightFTP, and up to 11% on Dnsmasq (between Clang and ASan with introspection), which suggests that the applications' performance was dominated by factors other than instrumentation cost (e.g., networking overhead). Thus, our measurements cannot be generalized to CPU-bound benchmarks.

To quantify the overhead on CPU-bound benchmarks, we also evaluated the approaches on the SPEC2006 INT benchmarks, which consist of 12 benchmarks. We excluded all C++ benchmarks (471.omnetpp, 473.astar, and 483.xalancbmk), which we expected to make little use of C functions and thus of our interceptors. Further, we excluded all benchmarks in which the tools detected memory safety errors (400.perlbench and 403.gcc). ASan detected memory leaks in two benchmarks (445.gobmk and 464.h264ref), and since we investigated only buffer overflows in this work, we disabled memory leak detection to also run them. SoftBound in its original and introspection versions detected memory safety errors in all but one benchmark (458.sjeng), which were presumably false

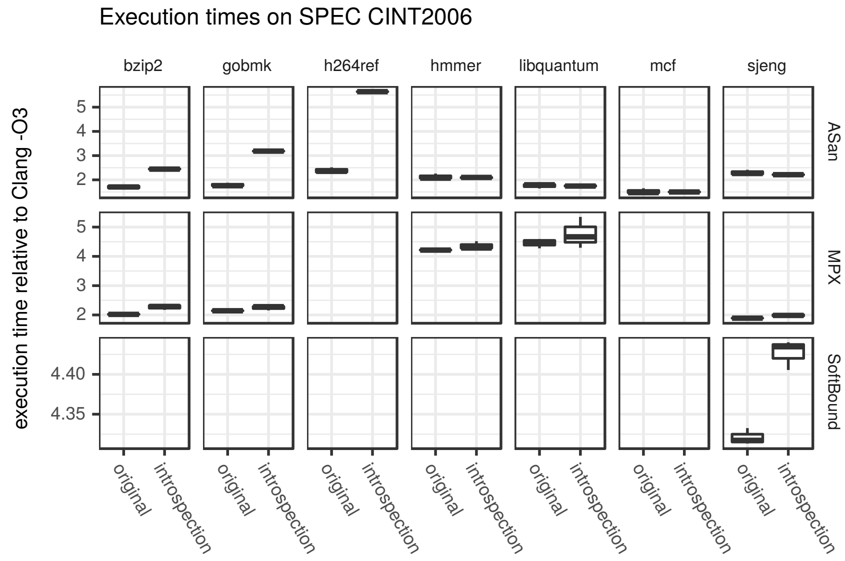

Fig. 2. Execution times on the SpecInt2006 benchmarks.

positives. MPX had an additional known false positive [19] in one benchmark (429.mcf), so we excluded this benchmark for MPX.

Figure 2 shows the execution times of the SPECInt2006 benchmarks relative to Clang -O3 as a baseline. On four of the seven benchmarks (429.mcf, 456.hmmer, 458.sjeng, 462.libquantum), the performance overhead was negligible because no interceptors were executed in code that contributed to the overall run-time performance of the respective benchmark. For SoftBound, the introspection overhead was 3% on the only benchmark that it could execute. Using introspection with ASan resulted in higher overheads, namely 140% on h264ref, 43% on bzip2, and 81% on gobmk. For MPX, the performance overhead of introspection was relatively low, with maximum overheads of 13% on bzip2 and 6% on gobmk.

We also executed micro-benchmarks, measuring the direct overhead of interceptors. For example, we evaluated the performance of the `strlen()` interceptor, which directly relies on `_size_right()` to call the safer `strnlen()` function. For SoftBound, the overhead was not measurable. For Intel MPX, the overhead was 2× for strings with a length of 10; for longer strings (e.g., a length of 1000) the overhead was not measurable. The overhead for ASan was the highest, as our `size_right()` implementation has to traverse the shadow memory, which depends linearly on the length of the string. Its overhead varied between 2× and 10× with different string lengths.

7 Discussion

Availability. We have demonstrated that our introspection-based libc interceptors are an effective means of mitigating the effects of buffer overflows. Our main idea is to use run-time information that is tracked by existing tools to prevent buffer overflows and to increase the availability of applications. Using the introspection-based interceptors is useful only in production, because during development and testing it would be preferable to abort execution so that the programmer can fix bugs that cause errors.

Complementarity. We have designed our approach to complement existing approaches for handling buffer overflows. Our idea is that, for important functions, programmers can implement custom semantics that mitigate the effects of buffer overflows. For buffer overflows in other functions or in user-level code, existing memory tools would continue to detect out-of-bounds accesses and would abort execution in the case of an error. Alternatively, the interceptors could also be used with the original failure-oblivious computing approach as a fallback for functions that are not guarded by introspection checks.

Performance. The overhead of introspection and our interceptors depends mainly on how efficiently a tool tracks bounds information. Our evaluation on servers suggests that the overhead of introspection is often small compared to the cost of network communication, making introspection especially applicable for servers. Our evaluation on the CPU-bound SPEC benchmarks also seems to suggest that libc functions are typically not part of the code that significantly contributes to the overall performance of a program. While the MPX-based introspection overhead was low on all benchmarks, only the ASan-based implementation caused larger overheads on three benchmarks. Overall, introspection-based libc functions are feasible with a low overhead for approaches that maintain explicit bounds information (e.g., Intel MPX or SoftBound), but result in higher overheads for approaches in which bounds information must be computed (e.g., in ASan). Furthermore, our implementation could be made more efficient by using introspection directly in the libc functions.

Implementation. We have demonstrated implementations of the `_size_right()` function for three popular bug-finding and bug-mitigation approaches and believe that implementing this function in many others (e.g., libcrunch [10,11]) is also straightforward. Some tools cannot give precise estimates for all pointers, which makes our approach less effective. For example, binary-instrumentation tools such as Valgrind [18] and Dr. Memory [3] cannot reliably determine the size of buffers located on the stack. Other approaches track run-time information only for specific types of allocations (e.g., stack buffers [2]). Furthermore, some tools give rough estimates in general or round up allocation sizes [1,2,6]; for example, after evaluating our approach with low-fat pointer checking [6,8], we found that rounding up allocation sizes alone mitigated several of the buffer overflows that we investigated.[7] Note that conservative estimates (e.g., the maximum integer value if no information is available) ensure correct execution, but might result in undetected errors.

8 Related Work

Failure-oblivious computing. Rinard et al. coined the term *failure-oblivious computing*, where illegal read accesses yield predefined values and out-of-bounds write accesses are ignored [25]. An extension of this work are *boundless memory blocks*, where out-of-bounds writes store the value in a hash map that can be returned for out-of-bounds reads to that address [4,12,26]. Furthermore, Long et al. extended failure-oblivious computing by also covering divide-by-zero errors and `NULL`-pointer dereferences [14]. In contrast to these approaches, introspection enables programmers to handle out-of-bounds accesses by taking into account the semantics of a function. However, the drawback of our approach is that library developers must implement these checks manually.

Failure-oblivious computing models. Durieux et al. studied failure-oblivious computing behaviors [9]. Their findings suggest that for many failures, multiple alternative strategies exist that can mitigate the error. For example, to mitigate a `NULL`-pointer dereference the access could be ignored, but the pointer could also be initialized with the address of a newly-created or existing object.

Monitored execution. Sidiroglou et al. devised a system that monitors an application for failures such as buffer overflows [29]. If a fault occurs, the current function is aborted and—based on heuristics—an appropriate value is returned. In order to avoid crashes because a pointer returns `NULL`, the heuristics take into account whether the parent function dereferences the pointer thereafter. While this approach takes into account the context of the fault, it lacks the ability of our introspection approach to benefit from programmer knowledge.

Libsafe. Libsafe replaces libc functions with enhanced versions that prevent buffer overflows from going beyond the stack frame [2]. It achieves this by traversing frames to determine their bounds and aborting the program if the bounds are exceeded. While we tried implementing the introspection function using the traversal logic, we found that it is based on assumptions such as the location of the stack, which no longer hold with modern mitigation techniques such as address space layout randomization. Additionally, libsafe does not handle out-of-bounds reads well, for which our approach, in contrast, can compute meaningful results, for example, by letting `strlen()` return the length of the buffer underlying the string if it is unterminated.

[7] EffectiveSan [7], an extension of the low-fat pointer approach, provides accurate bounds but has not been released to the public as of June 2018.

9 Conclusion

In this paper, we have presented how implementation of an introspection function that returns the length of an object can be used to implement failure-oblivious computing mechanisms. We have also shown that such a mechanism is useful in mitigating real-world errors and that the performance overhead when implemented in approaches such as Intel MPX is negligible. For reproducibility and to facilitate further research, we distribute all artifacts and experimentation scripts at `https://github.com/introspection-libc/main`.

References

1. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In: Proceedings of the 18th Conference on USENIX Security Symposium. pp. 51–66. SSYM'09, USENIX Association, Berkeley, CA, USA (2009)
2. Baratloo, A., Singh, N., Tsai, T.: Libsafe: Protecting critical elements of stacks. White Paper (1999), `http://www.research.avayalabs.com/project/libsafe`
3. Bruening, D., Zhao, Q.: Practical memory checking with dr. memory. In: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 213–223. CGO '11, IEEE Computer Society, Washington, DC, USA (2011)
4. Brunink, M., Susskraut, M., Fetzer, C.: Boundless memory allocations for memory safety and high availability. In: Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks. pp. 13–24. DSN '11, IEEE Computer Society, Washington, DC, USA (2011). https://doi.org/10.1109/DSN.2011.5958203
5. Cowan, C., Wagle, P., Pu, C., Beattie, S., Walpole, J.: Buffer overflows: Attacks and defenses for the vulnerability of the decade. In: DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings. vol. 2, pp. 119–129. IEEE (2000)
6. Duck, G.J., Yap, R.H.C.: Heap bounds protection with low fat pointers. In: Proceedings of the 25th International Conference on Compiler Construction. pp. 132–142. CC 2016, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2892208.2892212
7. Duck, G.J., Yap, R.H.C.: Effectivesan: Type and memory error detection using dynamically typed c/c++. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 181–195. PLDI 2018, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3192366.3192388
8. Duck, G.J., Yap, R.H., Cavallaro, L.: Stack bounds protection with low fat pointers. In: Symposium on Network and Distributed System Security (2017)
9. Durieux, T., Hamadi, Y., Yu, Z., Baudry, B., Monperrus, M.: Exhaustive exploration of the failure-oblivious computing search space. In: Proc. of the Int. Conf. on Sotware Testing and Verification (ICST). ICST'18 (Apr 2018)
10. Kell, S.: Towards a dynamic object model within unix processes. In: 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!). pp. 224–239. Onward! 2015, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2814228.2814238

11. Kell, S.: Dynamically diagnosing type errors in unsafe code. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 800–819. OOPSLA 2016, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2983990.2983998
12. Kuvaiskii, D., Oleksenko, O., Arnautov, S., Trach, B., Bhatotia, P., Felber, P., Fetzer, C.: SGXBOUNDS: Memory safety for shielded execution. In: Proceedings of the Twelfth European Conference on Computer Systems. pp. 205–221. EuroSys '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3064176.3064192
13. Lattner, C., Adve, V.: Llvm: a compilation framework for lifelong program analysis transformation. In: CGO 2004. pp. 75–86 (March 2004)
14. Long, F., Sidiroglou-Douskos, S., Rinard, M.: Automatic runtime error repair and containment via recovery shepherding. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 227–238. PLDI '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2594291.2594337
15. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: Softbound: Highly compatible and complete spatial memory safety for c. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 245–258. PLDI '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1542476.1542504
16. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: Cets: Compiler enforced temporal safety for c pp. 31–40 (2010). https://doi.org/10.1145/1806651.1806657
17. Nethercote, N., Seward, J.: How to shadow every byte of memory used by a program. In: Proceedings of the 3rd International Conference on Virtual Execution Environments. pp. 65–74. VEE '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1254810.1254820
18. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 89–100. PLDI '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1250734.1250746
19. Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Felber, P., Fetzer, C.: Intel mpx explained: A cross-layer analysis of the intel mpx system stack. Proc. ACM Meas. Anal. Comput. Syst. **2**(2), 28:1–28:30 (Jun 2018). https://doi.org/10.1145/3224423
20. Rigger, M., Grimmer, M., Wimmer, C., Würthinger, T., Mössenböck, H.: Bringing low-level languages to the jvm: Efficient execution of llvm ir on truffle. In: Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages. pp. 6–15. VMIL 2016, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2998415.2998416
21. Rigger, M., Schatz, R., Grimmer, M., Mössenböck, H.: Lenient execution of c on a java virtual machine: Or: How i learned to stop worrying and run the code. In: Proceedings of the 14th International Conference on Managed Languages and Runtimes. pp. 35–47. ManLang 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3132190.3132204
22. Rigger, M., Schatz, R., Mayrhofer, R., Grimmer, M., Mössenböck, H.: Sulong, and Thanks For All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. In: Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS 2018. https://doi.org/10.1145/3173162.3173174
23. Rigger, M., Schatz, R., Mayrhofer, R., Grimmer, M., Mössenböck, H.: Introspection for C and its Applications to Library Robustness. The Art, Science, and

Engineering of Programming (2) (2018). https://doi.org/10.22152/programming-journal.org/2018/2/4

24. Rinard, M.: Acceptability-oriented computing. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 221–239. OOPSLA '03, ACM, New York, NY, USA (2003). https://doi.org/10.1145/949344.949402
25. Rinard, M., Cadar, C., Dumitran, D., Roy, D.M., Leu, T., Beebee, Jr., W.S.: Enhancing server availability and security through failure-oblivious computing. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6. pp. 21–21. OSDI'04, USENIX Association, Berkeley, CA, USA (2004)
26. Rinard, M.C.: Failure-oblivious computing and boundless memory blocks (2005), technical Report
27. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: A fast address sanity checker. In: USENIX Annual Technical Conference. pp. 309–318 (2012)
28. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security. pp. 552–561. CCS '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1315245.1315313
29. Sidiroglou, S., Locasto, M.E., Boyd, S.W., Keromytis, A.D.: Building a reactive immune system for software services. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. pp. 11–11. ATEC '05, USENIX Association, Berkeley, CA, USA (2005)
30. Song, D., Lettner, J., Rajasekaran, P., Na, Y., Volckaert, S., Larsen, P., Franz, M.: Sok: Sanitizing for security. IEEE Symposium on Security and Privacy (S&P'19) Accepted. To Appear.
31. Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., Walter, T.: Breaking the memory secrecy assumption. In: Proceedings of the Second European Workshop on System Security. pp. 1–8. EUROSEC '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1519144.1519145
32. Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal war in memory. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy. pp. 48–62. SP '13, IEEE Computer Society, Washington, DC, USA (2013). https://doi.org/10.1109/SP.2013.13
33. The MITRE Corporation: Common vulnerabilities and exposures, `https://cve.mitre.org/`
34. van der Veen, V., dutt Sharma, N., Cavallaro, L., Bos, H.: Memory errors: The past, the present, and the future. In: Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses. pp. 86–106. RAID'12, Springer-Verlag, Berlin, Heidelberg (2012), `https://doi.org/10.1007/978-3-642-33338-5_5`
35. Watson, R.N.M., Woodruff, J., Neumann, P.G., Moore, S.W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., Murdoch, S.J., Norton, R., Roe, M., Son, S., Vadera, M.: Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In: 2015 IEEE Symposium on Security and Privacy. pp. 20–37 (May 2015). https://doi.org/10.1109/SP.2015.9
36. Younan, Y., Joosen, W., Piessens, F.: Runtime countermeasures for code injection attacks against c and c++ programs. ACM Comput. Surv. **44**(3), 17:1–17:28 (Jun 2012). https://doi.org/10.1145/2187671.2187679

Chapter 9

A Study on the Use of Inline Assembly

This chapter includes a paper on our analysis on the use of inline assembly in C projects.

Paper: Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. An Analysis of x86-64 Inline Assembly in C Programs. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2018, pages 84–99, New York, NY, USA, 2018. ACM

An Analysis of x86-64 Inline Assembly in C Programs

Manuel Rigger
Johannes Kepler University Linz
Austria
manuel.rigger@jku.at

Stefan Marr
University of Kent
United Kingdom
s.marr@kent.ac.uk

Stephen Kell
University of Cambridge
United Kingdom
stephen.kell@cl.cam.ac.uk

David Leopoldseder
Johannes Kepler University Linz
Austria
david.leopoldseder@jku.at

Hanspeter Mössenböck
Johannes Kepler University Linz
Austria
hanspeter.moessenboeck@jku.at

Abstract

C codebases frequently embed nonportable and unstandardized elements such as *inline assembly code*. Such elements are not well understood, which poses a problem to tool developers who aspire to support C code. This paper investigates the use of x86-64 inline assembly in 1264 C projects from GitHub and combines qualitative and quantitative analyses to answer questions that tool authors may have. We found that 28.1% of the most popular projects contain inline assembly code, although the majority contain only a few fragments with just one or two instructions. The most popular instructions constitute a small subset concerned largely with multicore semantics, performance optimization, and hardware control. Our findings are intended to help developers of C-focused tools, those testing compilers, and language designers seeking to reduce the reliance on inline assembly. They may also aid the design of tools focused on inline assembly itself.

CCS Concepts • **General and reference** → **Empirical studies**; • **Software and its engineering** → **Assembly languages**; **Language features**; • **Computer systems organization** → *Complex instruction set computing*;

Keywords Inline Assembly, C, Empirical Survey, GitHub

ACM Reference Format:
Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. 2018. An Analysis of x86-64 Inline Assembly in C Programs. In *VEE '18: 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, March 25, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3186411.3186418

1 Introduction

Inline assembly refers to assembly instructions embedded in C code in a way that allows direct interaction; for example, they can directly access C variables. Such code is inherently platform-dependent; it uses instructions from the target machine's *Instruction Set Architecture* (ISA). For example, the following C function uses the `rdtsc` instruction to read a timer on x86-64. Its two output operands `tickh` and `tickl` store the higher and lower parts of the result. The platform-specific constraints `=a` and `=d` request particular registers.

```
uint64_t rdtsc() {
  uint32_t tickl, tickh;
  asm volatile ("rdtsc":"=a"(tickl),"=d"(tickh));
  return ((uint64_t)tickh << 32)|tickl;
} /* see §2.5 for detailed syntax */
```

This kind of platform dependency adds to the complexity of C programs. A single complex ISA, such as x86, can contain about a thousand instructions [25]. Furthermore, inline assembly fragments may contain not only instructions, but also assembler directives (such as `.hidden`, controlling symbol visibility) that are specific to the host system's assembler.

It is not surprising that many tools that process C code or associated intermediate languages (such as LLVM IR [38] and CIL [45]) partially or entirely lack support for inline assembly. For example, many bug-finding tools (e.g., the Clang Static Analyzer [70], splint [18, 19, 63], Frama-C [69], uno [26], and the LLVM sanitizers [55, 58]), tools for source translation (e.g., c2go [44]), semantic models for C [36, 43], and alternative execution environments such as Sulong [51–53] and KLEE [12] still lack support for inline assembly, provide only partial support, or overapproximate it (e.g., by analyzing only the side effects specified as part of the fragment), which can lead to imprecise analyses or missed optimization opportunities. How to provide better support depends on the tool, for example, in Sulong, adding support for assembly instructions requires emulating their behavior in Java, while support in a formal model would require specifying the instructions in a language such as Coq.

Literature on processing C code seldom mentions inline assembly except for stating that it is rare [31]. Tool writers would benefit from a thorough characterization of the occurrence of inline assembly in practice, as it would enable them to make well-informed decisions on what support to add. Hence, we analyzed 1264 C projects that we collected from GitHub. We manually analyzed the inline assembly fragments in them (i.e., inline assembly instructions that are part of a single `asm` statement). From these fragments, we created a database with fragments specific to the x86-64 architecture to quantitatively analyze their usage.

We found that:

- Out of the most popular projects, 28.1% contain inline assembly fragments.
- Most inline assembly fragments consist of a single instruction, and most projects contain only a few inline assembly fragments.
- Since many projects use the same subset of inline assembly fragments, tool writers could support as much as 64.5% of these projects by implementing just 5% of x86-64 instructions.
- Inline assembly is used mostly for specific purposes: to ensure semantics on multiple cores, to optimize performance, to access functionality that is unavailable in C, and to implement arithmetic operations.

Our findings suggest that tool writers might want to consider adding support for inline assembly in their C tools, as it is used surprisingly often. We also found that inline assembly is not specific to a small set of domains, but appears in applications in which one might not expect it (e.g., text processing). Since most applications use the same subset of inline assembly instructions, a large proportion of projects could be supported with just a moderate implementation effort. Another finding, however, is that instructions are not all that matters. Rather, assembly instructions are only one of many non-C notations used in C codebases, all of which generally suffer from the same lack of tool support. For example, some uses of `asm` contain no instructions, consisting only of assembler directives and constraints. Others are interchangeable with non-portable compiler intrinsics or pragmas. Yet others gain meaning in conjunction with linker command-line options or scripts. This paper is therefore a first step towards characterizing this larger "soup" of notations that tools must support in order to fully comprehend C codebases.

2 Methodology

To guide tool developers in supporting inline assembly, we posed six research questions. We detail how we scoped the survey, selected and obtained C applications, and finally analyzed their inline assembly fragments.

2.1 Research Questions

To characterize the usage of inline assembly in C projects, we investigated the following research questions (RQs):

RQ1: How common is inline assembly in C programs? Knowing how commonly inline assembly is used indicates to C tool writers whether it needs to be supported.

RQ2: How long is the average inline assembly fragment? Characterizing the length of the average inline assembly fragment gives further implementation guidance. If inline assembly fragments typically contain only a single instruction, simple pattern-matching approaches might be sufficient to support them. If inline assembly fragments are large, numerous, or "hidden" behind macro meta-programming [57], it might be more difficult to add support for them.

RQ3: In which domains is inline assembly used? Answering this question helps if a tool targets only specific domains. It seemed likely that the usage of inline assembly differs across domains. We expected inline assembly in cryptographic libraries because instruction set extensions such as AES-NI explicitly serve cryptographic code [6]. This was supported by a preliminary literature search, as inline assembly is, for example, often mentioned in the context of cryptographic libraries [23, 37, 40, 61]. We also expected it to implement related security techniques, preventing timing-side channels [7] and compiler interference [66, 67]. It was less clear what other domains make frequent use of inline assembly.

RQ4: What is inline assembly used for? Knowing the typical use cases of inline assembly helps tool writers to assign meaningful semantics to inline assembly instructions. It also helps to determine whether alternative implementations in C could be considered. We hypothesized that inline assembly is used—aside from cryptographic use cases—mainly to improve performance and to access functionality that is not exposed by the C language.

RQ5: Do projects use the same subset of inline assembly? Answering this question determines how much inline assembly support needs to be implemented to cope with the majority of C projects. Currently, C tool writers have to assume that the whole ISA needs to be supported. However, one of our assumptions was that most projects—if they use inline assembly—rely on a common subset of instructions. By adding support for this subset, C tool writers could cope with most of the projects that use inline assembly.

2.2 Scope of the Study

Our focus was to quantitatively and qualitatively analyze inline assembly code. For our quantitative analysis, we built a database (using `SQlite3`) of inline assembly occurrences in code written for x86-64, as it is one of the most widely used architectures. The database contains information about each project, inline assembly fragment, and assembly instruction analyzed. We used this database to perform aggregate queries, for example, to determine the most common instructions. The database and aggregation scripts are available at *https://github.com/jku-ssw/inline-assembly* to facilitate further research. Additionally, we qualitatively analyzed all instructions to summarize them in a meaningful way.

2.3 Obtaining the Projects

In our survey, we selected C applications from GitHub, a project hosting website. To gather a diverse corpus of projects, we used two strategies:

We selected all projects with at least 850 GitHub stars—an arbitrary cut-off that gave us a manageable yet sufficiently large sample—which resulted in 327 projects being selected. Stars indicate the popularity of a project and are given by GitHub users [10]. We assumed that the most popular projects reflect those applications that are most likely processed by C tools.

We selected another 937 projects by searching for certain keywords[1] and by taking all matching projects that had at least 10 stars. The goal was to select projects of a certain domain with different degrees of popularity to account for the long tail of the distribution. In order to avoid personal forks, experiments, duplicate projects and the like [32, 41], we did not consider projects that had fewer than 10 stars.

2.4 Filtering the Projects

Our primary goal was to analyze C application-level code which we consider to be of general interest. Consequently, we ignored projects if they were operating systems, device drivers, firmware, and other code that is typically considered part of an operating system. Such code directly interacts with hardware and thus comes with its own special set of issues and usage patterns of inline assembly. Further, to keep the scope manageable, we focused on code for x86-64 Linux systems. Therefore, we excluded projects that worked only for other architectures or other operating systems. Further, we did not consider uncommon x86 extensions such as VIA's Padlock extensions [65].

We restricted our analysis to C code, excluding C++ code. Projects that mixed C/C++ code were also excluded if the C++ LOC were greater in number than the C LOC. We also excluded C/C++ header files (ending with `.h`) when they contained C++ code. A number of projects used C code to implement native extensions for PHP, Ruby, Lua, and other languages; we included such code in our analysis. In a few cases, inline assembly was part of the build process; for example, some `configure` scripts checked the availability of CPU features by using `cpuid`. We discarded these cases because inline assembly was not part of the application; however, we checked whether build scripts generated source files with inline assembly, which we then incorporated in our analysis.

[1]Our keywords were: crc, argon, checksum, md5, base64, dna, web server, compression, math, fft, string, aes, simulation, editor, single header library, parser, debugger, ascii, xml, markdown, smtp, sqlite, mp3, sort, json, bitcoin, udp, random, prng, metrics, misc, tree, parser generator, hash, font, gc, i18, and javascript.

34 projects used inline assembly in fairly large program fragments, notably featuring SIMD instructions and using preprocessor-based metaprogramming. Although written using inline assembly constructs, these fragments have more in common with separate (macro) assembly source files. In particular, supporting these would require a close-to-complete implementation of an ISA. We excluded these fragments from our quantitative analysis.

We performed our analysis on unpreprocessed source code to include all inline-assembly fragments independent of compile-time-configuration factors [62]. This is significant because inclusion of inline assembly is often only conditional, achieved by `#ifdefs` that not only check for various platforms and operating systems, but also for configuration flags, various compilers, compiler versions, and availability of GNU C intrinsics [17]; examining only preprocessed code would have left out many fragments.

2.5 Inline Assembly Constructs

Since inline assembly is not part of the C language standard, compilers differ in the syntax and features provided. In this study, we assume use of the GNU C inline assembly syntax, which is the de-facto standard on Unix platforms, recognizes the `asm` or `__asm__` keywords to specify an inline assembly fragment, and has both "basic" and "extended" flavors. Using basic `asm`, a programmer can specify only the assembler fragment or directive. Use cases for basic assembly are limited; however, in contrast to extended `asm`, basic inline assembly can be used outside of functions. For example,

```
asm(".symver memcpy,memcpy@GLIBC_2.2.5")
```

uses basic inline assembly for a symbol versioning directive (see Section 5).

The more commonly used form is extended `asm`, which also allows specifying output and input operands as well as side effects (e.g., memory modifications). It is specified using

```
asm ( AssemblerTemplate : OutputOperands
      [ : InputOperands [ : Clobbers ] ]).
```

Adding the `volatile` keyword restricts the compiler in its optimization; for example, it prevents reachable fragments from being optimized (e.g., by register reallocation).

2.6 Analyzing the Instructions

Our analysis focused on inline assembly fragments found with `grep` in the source code. We searched for strings containing "asm", which made it unlikely that we missed inline assembly instructions. For the quantitative analysis, we judged whether an inline assembly fragment was used for an x86-64 Linux machine. If so, we manually extracted the fragment and preprocessed it (see the criteria below) using a script created for this purpose.

We assumed that tools would support all addressing modes (e.g., register addressing, immediate addressing, and direct memory addressing) for a certain instruction. Consequently, we did not gather statistics for different addressing modes. Inline assembly can contain assembler directives that instruct the assembler to perform certain actions, for example, to allocate a global variable. We ignored such assembler directives in our quantitative analysis, but discuss them qualitatively. An exception is the `.byte` directive, which is sometimes used to specify instructions using their byte representation (and similar cases, see Section 5), for which we assumed their mnemonic (i.e., their textual) representation.

By default, GCC assumes use of the AT&T dialect [8, 9.15.3.1, 9.15.4.2]; however, some projects enabled the Intel syntax instead. Using the AT&T syntax, a size suffix is typically appended to denote the bitwidth of an instruction. An `add` instruction can, for example, operate on a byte (8 bit), long (32 bit), or quad (64 bit) using `addb`, `addl`, and `addq`, respectively. Using Intel syntax, the size suffix is typically omitted. For consistency, we stripped size suffixes and recorded only the instruction itself (e.g., `add`). We also applied other criteria to group instructions.[2]

3 Quantitative Results

Based on our quantitative analysis, we can answer the first three research questions on the use of inline assembly in C projects, the length of fragments used, and the domains in which they occur.

Projects using inline assembly. Our corpus contained 1264 projects, of which 197 projects (15.6%) contained inline assembly for x86-64. The distribution differed between the popular projects and those selected by keywords. Among the most popular 327 projects, 28.1% contained inline assembly, while of the 937 other projects only 11.2% used inline assembly. One possible explanation for this difference is that the popular projects were larger (69 KLOC on average) than the projects selected by keywords (13 KLOC on average).

Density of inline assembly fragments. The percentage of projects with inline assembly is high, which is surprising because many C tools are based on the assumption that inline assembly is rarely used. Nevertheless, in terms of density, inline assembly is rare, with one fragment per 40 KLOC of C code on average. The density of inline assembly is lower for the popular projects (one fragment per 50 KLOC) than for those selected by keywords (one fragment per 31 KLOC).

[2] The x86 architecture allows adjusting the semantics of an instruction with a prefix. This includes the `lock` prefix for exclusive access to shared memory, and `rep` to repeat an instruction a certain number of times. In inline assembly, these prefixes are denoted as individual instructions (e.g., `lock; cmpxchg`). In our survey, we merged the prefix and its instruction and handled them as a single instruction (e.g., `lock cmpxchg`). The `xchg` instruction has an implicit `lock` prefix when used with a memory operand. For jump-if-condition-is-met instructions and set-on-condition instructions, several mnemonics exist for the same instruction. We grouped such mnemonics and counted them as the same instruction. We also considered different software interrupts as distinct instructions, since their purposes differ markedly.

RQ1.1: 28.1% of the most popular and 11.2% of the keyword-selected projects contained inline assembly.

Number of fragments per project. To measure the number of inline assembly fragments in a project, we considered only unique fragments because duplicates do not increase the implementation effort (see Figure 2). 36.2% of the projects with inline assembly contained only one unique inline assembly fragment. 93.3% of them contained up to ten unique inline assembly fragments. On average, projects analyzed in detail contained 3.7 unique inline assembly fragments (with a median of 2).

RQ1.2: C projects with inline assembly which were analyzed in detail contained on average 3.7 unique inline assembly fragments (median of 2)

Overview of the fragments. In total, we analyzed 1026 fragments, of which 607 were unique per project. Projects that used inline assembly tended to bundle instructions for several operand sizes in the same source file; consequently, we found 715 fragments that were unique within a single file. Overall, we found 197 unique inline assembly fragments.

Analysis of the fragments. Of the 197 projects with inline assembly, we analyzed the inline assembly in 163 projects (82.7%) in detail. To this end, we extracted each fragment and added it together with metadata about the project to our database, which we then queried for aggregate statistics (e.g., the frequency of instructions). The 34 projects that we did not analyze used complicated macro metaprogramming and/or contained an excessive number of large inline assembly fragments, which made our manual analysis approach infeasible. We call these "big-fragment" codebases. They consisted mostly of mature software projects (such as video players) that used inline assembly for SIMD operations, for which they provided several alternative implementations (e.g., AVX, SSE, SSE2). We assumed that tools need to provide close-to-complete SIMD inline assembly support for these projects, and thus omitted them from the detailed analysis.

RQ2.1: 17.3% of all C projects with inline assembly contained macro-metaprogramming and many large inline assembly fragments that were omitted from our detailed analysis.

Instructions in a fragment. When analyzing instructions in inline assembly fragments, we again considered those fragments that were unique to a project. Typically, they were very short (see Figure 1). 390 (64.3%) of them had only one instruction. 73.3% had up to two instructions. However, we also found inline assembly fragments with up to 438 instructions. The average number of instructions in an inline assembly

Table 1. The 10 most common file names that contained inline assembly and their average numbers of instructions

file name	projects	instr.	file name	projects	instr.
sqlite3.c	10	1.0	inffas86.c	4	1.0
atomic.h	8	3.4	mb.h	4	1.0
SDL_endian.h	4	2.0	timing.c	4	1.0
atomic-ops.h	4	2.0	util.h	4	1.0
configure.ac	4	1.0	utils.h	4	2.2

fragment was 9.9; the median was 1. In total, we found only 167 unique instructions, which contrasts with the approximately 1000 instructions that x84-64 provides [25].

RQ2.2: Inline assembly fragments contained on average 9.9 instructions (median of 1) per fragment.

Duplicate fragments. It has been shown that file duplication among GitHub projects—mainly targeting popular libraries copied into many projects—is a common phenomenon [41], which we also observed for the projects we analyzed (see Table 1). For example, many projects contained `sqlite3.c`, which corresponds to the database with the same name (which uses the `rdtsc` instruction), `SDL_endian.h` for the SDL library (which uses inline assembly for endianness conversions), and `inffas86.c` (which implements a compression algorithm using inline assembly). We did not try to eliminate such duplicate files in the analysis, because the duplication is significant: tool authors have a stronger incentive to implement those inline assembly instructions that are used by many projects.

Project domains. Table 2 classifies the projects into domains and shows how many projects per domain contained inline assembly. We created this table by manually labelling the projects using an ad-hoc vocabulary of seventeen domain labels. Note that the domains differ in extent and intersect in some cases. As expected, the majority of projects were `crypto` libraries (with SSL/TLS libraries as a subdomain). However, in general, the domains were relatively diverse. In addition to the eleven domains in the table, we also used seven other domain labels[3] which had fewer than 7 projects each and were omitted for brevity.

RQ3: Inline assembly is used in many domains, most commonly in projects for crypto, networking, media, databases, language implementations, concurrency, ssl, string and math libraries.

[3]These were: games, general-purpose libraries, reverse engineering, garbage collection, monitoring, and virtualization.

Table 2. Domains of projects that used inline assembly (each domain containing at least 7 projects)

domain	projects		description
	#	%	
crypto	23	11.7	encryption and decryption algorithms, cryptographic hashes, non-cryptographic hashes, base64 encodings
networking	20	10.2	protocols, email systems, chat clients, port scanners
media	17	8.6	video and music players and encoders, audio processing software, image libraries
database	16	8.1	databases, key/value storages, other in-memory data structures
language implementation	15	7.6	compilers, interpreters, virtual machines
misc	13	6.6	projects not assigned to any domain
concurrency	9	4.6	concurrency libraries, concurrent data structures
ssl	8	4.1	SSL/TLS libraries
string library	8	4.1	string algorithms, converters between different formats, parsers
math library	7	3.6	scientific applications, math libraries
web server	7	3.6	

4 Use Cases of Inline Assembly Instructions

We identified four typical use cases for inline assembly. One was to prevent instruction reorderings, either in the compiler (prevented by "compiler barriers") or in the processor, both in single-core execution and between multiple cores (prevented by memory barriers and atomic instructions—see Section 4.1). The second use case was performance optimization, for example, for efficient endianness conversions, hash functions, and bitscans (see Section 4.2). The third use case was to interact with the hardware, for example, to detect CPU features, to obtain precise timing information, random numbers, and manage caches (see Section 4.3). The fourth use case was for more general "management" instructions, for example, moving values, pushing and popping from the stack, and arithmetic instructions (see Section 4.4).

Note that there might be more than one reason for using assembly code: for example, programmers might read the elapsed clock cycles using the `rdtsc` instruction because

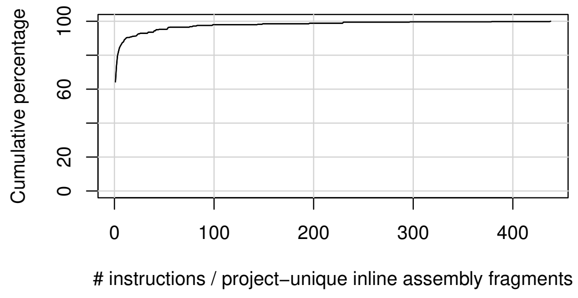

Figure 1. Inline assembly fragment lengths

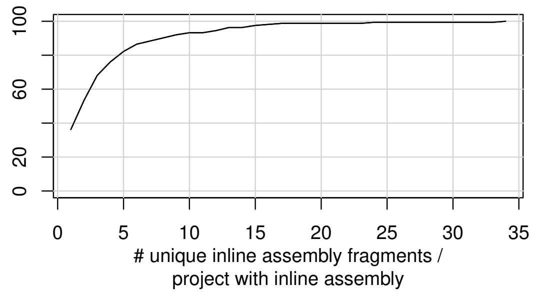

Figure 2. Number of fragments per project

similar C timing functions might not provide the same accuracy; however, they might also use it for efficiency because it has a lower overhead than those functions.

RQ4: Inline assembly is used to ensure correct semantics on multiple cores, for performance optimization, to access functionality that is unavailable in C, and to implement arithmetic operations.

For each use case, we denoted in parentheses the percentage of projects that relied on at least one instruction. Some instructions were counted for several use cases; for example, `xchg` can be used to exchange bytes to convert the endianness of a 16-bit value and has an implicit `lock` prefix when applied to a memory operand, which is why it can also be used to implement an atomic operation.

We found that most inline assembly instructions can also be issued using compiler intrinsics instead of inline assembly (compiler barriers being the only exception). Although compiler intrinsics are specific to a compiler, they are easier to support in tools because they follow the same conventions as C functions, both syntactically and semantically. For example, unlike inline assembly, compiler intrinsics cannot modify local variables.

4.1 Instruction Reordering and Multicore Programming

For threading and concurrency control, most C programs rely on libraries (such as pthreads [9]), compiler intrinsics, and inline assembly instructions. Intrinsics and assembly instructions are used mainly for historical reasons, since atomic operations became standardized only in 2011 [29].

In this section, we describe how inline assembly was used to perform atomic operations and to control the ordering of instructions at the compiler and processor levels.

Atomic instructions (24.0%). In 24.0% of the projects, instructions were prefixed to execute atomically to prevent races when data is accessed by multiple threads (see Table 3). More recent code uses C11 atomic instructions as an alternative; for example, the add-and-fetch operation, which is equivalent to `lock xaddq`, can also be implemented using the C11 `atomic_fetch_add` function.

Compiler barriers (24.0%). C compilers are permitted to reorder instructions that access memory. Unless specially directed, these reorderings are allowed to assume single-threaded execution. A common use case of inline assembly is to implement such a special directive, called a "compiler barrier", telling the compiler to assume an arbitrary side effect to the memory, hence preventing reorderings around the barrier. This is expressed as follows (a `memory` clobber):

```
asm volatile("" : : : "memory");
```

Such barriers are often necessary in lock-free concurrent programming.

Additionally, compiler barriers were used to prevent the compiler from optimizing away instructions. For example, in Listing 1, the compiler is prevented from removing `memset` to implement a `secure_clear` function that can be used to clear sensitive data from memory. A compiler could remove the `memset` call, for example, if the call is inlined and the memory freed, because the compiler can assume that it is no longer accessible [16]. If so, attackers could exploit a buffer overflow at another location in the program to read the data. Note that the C11 standard specifies the function `memset_s`, which provides the same guarantees as the `secure_clear` implementation.

Listing 1. Implementing a secure memory-zeroing function

```
void secure_clear(void *ptr, size_t len) {
    memset(ptr, 0, len);
    asm volatile("" : : "r"(ptr) : "memory");
}
```

Memory barriers (11.2%). Not only compilers, but also processors reorder instructions. Memory barriers are used to prevent reorderings by the processor (see Table 4). On x86, they are mostly needed for special cases (e.g., write-combining memory or non-temporal stores), as all memory accesses except store-load are ordered, so a compiler barrier is often sufficient to ensure the desired ordering [1].

Spin loop hints (15.1%). We found that 27 projects used the `pause` instruction as a processor hint in busy-waiting loops. Busy waiting refers to a tight loop in which a check is performed repeatedly. For example, in spinlocks, a thread repeatedly tries to acquire a lock that has potentially already been acquired by another thread. To remedy the costs of busy waiting in terms of performance and energy, `pause` causes a short delay and controls speculative execution [48].

4.2 Performance Optimizations

Several inline assembly instruction categories were used to optimize performance, even when the code could have been written in pure C.

SIMD instructions (6.1% + 34 projects). In the quantitative analysis, only a few SIMD instructions ranked among the most common instructions, for example, `pxor` and `movdqa` (used in 9 and 8 projects, respectively). However, the actual number of projects using SIMD instructions was higher because the 34 "big-fragment" projects that we did not analyze mostly targeted various SIMD instruction sets (e.g., MMX, SSE, and AVX);

Endianness conversion (25.7%). A common use case of inline assembly is to change the byte order of a value (see Table 5), for example, when the file format of a read file and the processor differ in their endianness. On x86, the `xchg` instruction can be used to swap the bytes of 16-bit integers, because x86 allows both the higher and lower byte of a 16-bit register to be addressed. A less common alternative is to use rotation left or right (`rol` or `ror`) by eight places. For 32-bit and 64-bit values, the `bswap` instruction is used instead. Half of the projects with instructions for endianness conversions included the SDL library [54] as source files in the repository tree. Using inline assembly to implement endianness conversion is most likely a performance optimization that is no longer needed, because state-of-the-art compilers produce as efficient code [22].

Hash functions (15.6%). A number of projects used inline assembly to implement hash functions. This included the `crc32` instruction as well as the `rol`, `ror`, and `shl` instructions to compute the CRC32 and SHA hashsums (see Table 6). The shift and rotate instructions could also simply be implemented in C, and current C compilers produce efficient machine code for them [49].

Bit scan (7.8%). Several projects used bit-scan instructions to determine the most significant one-bit using `bsr` (in 12 projects) or the least significant one-bit using `bsf` (in 7 projects). Both instructions have many applications [68, Sections 5.3 and 5.4]. As `bsr` corresponds to a `log2` function that rounds the result down to the next lower integer, the instruction was often used for this purpose. For an input `value`, it is also possible to round the result up by providing the input `(value<<1)-1`. Bitscan instructions were mostly used by memory allocators such as jemalloc [20] (which was included in four projects) or dlmalloc as well as by compression and math libraries.

Advanced Encryption Standard (AES) instructions (2.2%). We found that 2.2% of the projects used inline assembly to speed up AES using AES-NI instructions.

4.3 Functionality Unavailable in C

Feature detection (28.5%). The `cpuid` instruction allows programs to request information about the processor. It was often used to check cache size, facilities for random-number generation, or support for SIMD instructions such as SSE and AVX. Also, perhaps surprisingly, `cpuid` is defined as a "serializing instruction" in the processor's out-of-order execution semantics, guaranteeing that all instructions preceding it have been executed and none is moved above it.

Clock cycle counter (60.9%). Inline assembly was most commonly used for accurate time measurement using the `rdtsc` instruction. The `rdtsc` instruction reads the timestamp counter provided by the CPU. This instruction is both efficient and accurate for measuring the elapsed cycles, which makes it suitable for benchmarking [27]. As the CPU's out-of-order execution could move the code-to-be-benchmarked before the `rdtsc` instruction, it is typically used together with a serializing instruction (such as `cpuid`) when measuring the elapsed clock cycles. To minimize the overhead when measuring the end time, the `rdtscp` instruction can be used, which also reads the timestamp counter but has serializing properties; to prevent subsequent code from being executed between the start- and end-measuring instructions, another `cpuid` instruction is needed.

Debug interrupts (3.9%). Some projects used an interrupt to programmatically set a breakpoint in the program. If a debugger, such as GDB, is attached to the program, executing a breakpoint causes the program to pause execution. A definition of a breakpoint, for example,

```
#define BREAKPOINT asm("int $0x03")
```

is often selectively enabled through ifdefs, depending on whether the debugging mode in the project is enabled.

Prefetching data (3.9%). The `prefetch` instruction was used in 7 projects. It is a hint to the processor that the memory specified by the operand will be accessed soon, which typically causes it to be moved to the cache. Using a `prefetch` instruction timely can improve performance, because the latency of fetching data can be bridged. However, as processors provide prefetching mechanisms in hardware, using them correctly requires a thorough understanding of cache mechanisms [39]. For example, software prefetches that are issued too early can reduce the effectiveness of hardware prefetching by evicting data that is still being used.

Random numbers (3.4%). The `rdrand` instruction was used in 6 projects. It computes a secure random number with an on-chip random-number generator that uses statistical tests to check the quality of the generated numbers [28].

Table 3. Instructions for atomics (with at least 4 projects using them)

instruction	% projects
lock xchg	14.2
lock cmpxchg	13.2
lock xadd	8.6
lock add	3.0
lock dec	2.5
lock inc	2.5
lock bts	2.0

Table 4. Instruction for fences

instruction	% projects
mfence	6.6
sfence	6.6
lfence	5.6

Table 5. Instructions for endianness conversion

instruction	% projects
lock xchg	14.2
bswap	9.1
ror	5.6
rol	4.6

Table 6. Instructions for hash functions

instruction	% projects
shl	6.1
ror	5.6
rol	4.6
crc32	2.5

Table 7. Instructions for timing

instruction	% projects
rdtsc	27.4
cpuid	25.4
rdtscp	2.5

Table 8. Instructions for feature detection

instruction	% projects
cpuid	25.4
xgetbv	4.1

Table 9. Instructions to move around data

instruction	% projects
mov	24.9
pop	7.1
push	7.1
pushf	1.5
popf	1.0

Programmers can verify successful random-number generation by checking the carry flag (`CF`), for example, by writing its value to a variable (using the `setc` instruction).

4.4 Supporting Instructions

Some instructions were most commonly used together with other instructions, and we therefore classify them as "supporting instructions".

Moving and copying data (30.2%). Some inline assembly fragments, mainly those larger in size, contained instructions to copy data to a register before some other instruction accessed this register (see Table 9). While the `mov` instruction was also used in smaller fragments for that purpose, the instruction could in many cases have been omitted entirely, simply by correctly specifying the input and output constraints and letting the compiler generate the data-movement code. In rarer cases, `mov` was also used to build a stack trace by retrieving the value of `%rbp`. Additionally, the `push` and `pop` instructions were used to save register values on the stack and restore them. The `pushf` and `popf` instructions were used to save and restore processor flags.

Arithmetic operations (21.2%). Arithmetic instructions (see Table 10) were used in larger inline assembly fragments, for example, in vector-reduction arithmetic (e.g., vector summation, inner product, and vector chain product) [47] in crypto and math libraries. Additionally, they were used to implement operations that are not available in standard C.

Table 10. Instructions for arithmetics

instruction	% projects
xor	12.7
add	10.7
mul	6.6
sub	6.6
adc	6.1
lea	5.6
or	5.6
and	4.6
inc	3.6
dec	3.0
neg	3.0

Table 11. Instructions for control flow (with at least 4 projects using them)

instruction	% projects
jmp	9.1
cmp	6.6
jz/je	5.6
jne/jnz	5.1
test	4.6
jb/jnae/jc	4.1
jnb/jae/jnc	3.6
ja/jnbe	2.0
jbe/jna	2.0

Table 12. Instructions that set a value based on a flag

instruction	% projects
sete/setz	5.1
setc/setb	3.6
setne/setnz	2.0

Table 13. Instructions with `rep` prefixes

instruction	% projects
rep movs	3.0
cld	2.0
rep stos	2.0

An example is the `mulq` instruction, which can be used to obtain a 128-bit result when multiplying two 64-bit integers.

Another example is use of the `add` instruction for implementing signed integer addition with wraparound semantics, because signed integer overflow has undefined behavior in C [15]. Inline assembly was also used to implement operations on large integer types; for example, `adc` was used for multi-word additions, because it adds the value of the carry flag to the addition result (e.g., see [13]).

Control-flow instructions (13.4%). Control-flow-related instructions were mostly confined to larger inline assembly fragments (see Table 11). Some of these instructions compute condition values (`test` and `cmp`), while others transfer control flow (e.g., `jmp`). However, they were also used for indirect calls, for example, when implementing `setjmp` and `longjmp` for coroutines. Another example was retrying the `rdrand` instruction using `jnc` because it sets `CF≠1` if unsuccessful.

Set-byte-on-condition (10.6%). Several projects used instructions that extract a value from the flags register (see Table 12). They were typically used together with instructions that indicate their success via a flag. For example, `rdrand` sets `CF=1` on success, and the flag's value can be used from C by loading it into a variable using `setc`. As another example, `cmpxchg` sets `ZF=1` if the values in the operand and destination are equal, which can be checked using `setz`.

No-ops (3.9%). The `nop` operation was used in 7 projects and does not have any semantic effects. Normally, it is used for instruction alignment to improve performance.

Rep instructions (3.4%). Instructions with a `rep` prefix were used to implement string operations (see Table 13). The `rep` prefix specifies that an instruction should be repeated a specified number of times. To control the direction of repetition, `cld` was used to clear the direction flag.

4.5 Implementing Inline Assembly

One goal was to determine the "low-hanging fruits" when implementing inline assembly. Therefore, the question was how many projects could be supported by implementing only 5% of all x86-64 instructions (i.e., 50 instructions). The result is shown in Table 14. It groups similar instructions that can be easily implemented together in an order that maximizes the number of supported projects with each new group. Note that the order of the implementation makes a difference because a project is considered to be supported only if all the instructions it uses are supported.

First, the timing instructions should be implemented; although `rdtscp` is seldom used, it is similar to `rdtsc` and could be implemented together with it. Next would be the feature detection instructions. For tools that execute C code, the feature detection instructions could also be used to indicate that certain features are missing (e.g., SIMD support), which could then guide the program not to use inline assembly for these features. Some instructions could be implemented as "no-ops", as they either have no semantic effect (e.g., `prefetch`) or are important only when multithreaded execution needs to be modeled or analyzed (e.g., memory fences). Implementing bit operations and atomics, would

Table 14. Instruction groups and the percentage of projects covered by them

Instruction group	Instructions	Supported Projects
Timing	rdtsc, rdtscp	11.0%
Feature detection	cpuid, xgetbv	18.4%
"No-ops"	<compiler barrier>, mfence, sfence, lfence, prefetch, nop, int $0x03, pause, ud2	28.2%
Bit operations	bsr, bsf, or, xor, neg, bswap, shl, rol, ror	41.1%
Atomics	lock xchg, lock cmpxchg, lock xadd, lock add, lock dec, lock inc	51.5%
Moving data	mov, push, pop	58.9%
Checksum	crc32	62.0%
Flag operations	sete/setz, setc/setb, setne/setnz, stc	67.5%
Arithmetics	add, sub, mul, adc, lea, div, imul, sbb, inc, dec	70.6%
Random numbers	rdrand	72.4%
Control flow	jmp, jnb/jae/jnc	76.7%
String operations	rep movsb	77.9%

allow half of the projects to be supported. Finally, by implementing the other instructions in the table (50 in total), tool writers could support 77.9% of the projects that we analyzed in detail and 64.5% when counting also the projects that we did not analyze in detail. An alternative to implementing `rep movsb` would be `int $0x80`; however, we thought that this instruction is difficult to implement because it is used for system calls, and thus preferred `rep movsb`. In general, we believe that the semantics of most instructions in the table are relatively straightforward to support in comparison with some other portions of the instruction set, such as extensions for hardware transactional memory [72].

RQ5: By implementing 50 instructions (5% of x86-64's total number of instructions) tool writers could support 64.5% of all projects that contain inline assembly.

Note that, depending on the tool, another order could be more suitable—tool writers can consult the database to determine the order that best suits their project.

5 Declarative Use Cases of Inline Assembly

Our syntactic analysis naturally turned up uses of the `asm` keyword, but, perhaps surprisingly, not all of these were for inserting instructions. A small number of projects used it instead for declarative means, for example, to control the behavior of the linker. While many tools can ignore or work around these usages of inline assembly, we discuss some of the examples found as a first step towards characterizing the remaining "soup" of non-C notations used in C codebases, as noted in the *Introduction*. Additionally, we discuss examples in which a mix of instruction representations was used to encode certain instructions.

Specifying assembler names. Some projects use the inline assembly `asm` keyword to specify the names of symbols, thus preventing name mangling. For example,

```
AES_ECB_encrypt(...) asm("AES_ECB_encrypt");
```

is a function declaration with an inline assembly label that specifies its symbol name in the machine code. In this example, the function was implemented in macro assembly, so, in order to guarantee binary compatibility, the name must not be mangled. Labels are also used when the symbol cannot be written in plain C (e.g., because it contains special characters that are forbidden in C), and when symbol names need to be accessible by a native function interface.

Linker warnings. A few projects used inline assembler directives to emit linker warnings when incompatible or deprecated functions of a library were included. C library implementations often make use of this; for example, using

```
asm(".section .gnu.warning.gets; .ascii
  \"Please do not use gets!\"; .text");
```

at global scope causes the linker to emit a warning when the unsecure `gets` function is linked.

Symbol versioning. Several libraries used symbol versioning to refer to older libc functions in order for code compiled on a recent platform to work also on older platforms. A common example is `memcpy`, where current Linux versions link to the relatively new glibc function `memcpy@@GLIBC_2.14`. Most other standard library functions link to older glibc versions; for example, `memset` links to `memset@@GLIBC_2.2.5`. If the most recent `memcpy` is not needed, and older platforms should be supported, one can directly bind `memcpy` to the older 2.2.5 version, for example, using

```
asm(".symver memcpy,memcpy@GLIBC_2.2.5").
```

Register variables. Programmers can use inline assembly to associate local or global variables with a specific register [24]. For example, one could store an interpreter's program counter in the `%rsi` register:

```
register unsigned char *pc asm("%rsi");
```

Such code was used for performance optimization.

Instruction representations. Inline assembly instructions are normally written using mnemonics, which are textual representations of the assembly instructions. However, 16.6% of the projects with inline assembly (27 of 163) deviated from this, either by avoiding instruction mnemonics entirely or by combining mnemonics to surprising effect.

A number of projects denoted the `pause` instruction as `rep; nop`. Even though the `rep (0xF3)` prefix is unspecified for `nop (0x90)`, the resulting opcode corresponds to that of the `pause (0xFE90)` instruction. This works because portions of the prefix-opcode space are, in effect, aliased, and the assembler will accept an aliased combination in place of the more direct encoding. In other cases, instructions were directly specified by their opcode, for example, `.byte 0x0f, 0x01, 0xd0` to represent the `xgetbv` instruction. Some projects even mixed both representations within an instruction; for example, `.byte 0x66; clflush %0` was used to specify `clflushopt`, because prepending `0x66` to the opcode of `clflush (0x0FAE)` yields the opcode for `clflushopt (0x660FAE)`.

Programmers resort to such notations to allow use of older assemblers which fail to recognize mnemonics, but can process opcodes or simpler instructions. Such notations were also used for less common architectures, for example, for VIA's Padlock extensions [65]. While tool writers could treat common patterns not specified by their mnemonics as special cases, canonicalizing them would be more comprehensive, as also rare or unknown combinations of instruction-representations could be supported.

6 Threats to Validity

We used a standard methodology [21] to identify validity threats, which we mitigated where possible. We considered internal validity (i.e., whether we controlled all confounding variables), construct validity (i.e., whether the experiment measured what we wanted to measure), and external validity (i.e., whether our results are generalizable).

6.1 Internal Validity

The greatest threat to internal validity is posed by errors in the analysis. We used a manual best-effort approach to analyze x86-64 inline assembly fragments detected by our string search. It cannot be ruled out that we incorrectly included inline assembly that works only for other architectures (e.g., x86-32), or, conversely, that we rejected some erroneously. To address this, we carefully analyzed inline assembly fragments and repeated analyses when we had doubts or when we found a single inline assembly fragment in several projects, so we believe that errors in the analysis have little impact on the result. A threat in the qualitative analysis is that biases in our judgements influenced the outcome of the study; however, since we also used a quantitative approach, gross distortions or misinterpretations are unlikely.

6.2 Construct Validity

The main threat to construct validity is that we used a source-code-based search to determine the usage of inline assembly. This approach enabled us to analyze the usage of inline assembly independent of conditions such as operating system, compiler and its version, platform, and availability of functions and intrinsics. However, while conducting this survey, we found that some system library headers (which are not part of the project repository) contained inline assembly in macros. For example, GCC provides a `cpuinfo.h` header file that wraps the `cpuid` instruction. We ignored such system header libraries, and inspected only the source code of the projects. While we recognize that this could have had a minor impact on the quantitative analysis, we would expect the qualitative analysis to remain unaffected, as the macros were used for the same purposes as inline assembly fragments. Note that, if the goal had been to analyze what inline assembly instructions are actually executed on a given system, a binary-level approach would have been more appropriate. Similarly, to analyze which instructions appear in built binaries in any particular configuration, analysis after C preprocessing would have been more useful.

6.3 External Validity

There are several threats to external validity, which are given by the scope of our work.

Sample Set. One problem could be that the set of projects is not representative of user-level C. To mitigate this and increase the variety of projects, we employed two different strategies to collect samples for analysis, one based on GitHub stars as a proxy for popularity and one based on keywords. Nevertheless, the number of stars of a project might not reflect its popularity, and our search keyword could also bias the results. While inline assembly could differ in domains not represented in the survey, we believe that the overall results would differ only marginally, given the large body of source code that we examined (1264 projects and 56 million LOC).

OS software. We excluded projects with software that typically forms part of an operating system, which we would expect to use more inline assembly than typical user applications, for example, in order to implement interrupt logic, context switches, clearing pages, and for virtualization extensions [5, 42]. The usage of inline assembly in such projects would best be analyzed separately (especially when considering the size of operating systems), which we will consider as part of future work. The findings of our survey are thus not generalizable to such software.

Macro assembly code. We analyzed only inline assembly in detail and not macro assembly, which is stored in separate files. Macro assembly is used to implement larger program parts. This is reflected in the high average number of 888.3 LOC of macro assembly in the 7.8% of projects that used macro assembler. Note that projects with inline assembly were likely to also contain macro assembly, namely with 33.5%. While inline assembly is syntactically and semantically embedded into C code (e.g., C code can access registers, and inline assembly can access local C variables), macro assembler communicates only via the calling convention of the platform. As macro assembly can be called via native function interfaces by C execution environments and allows modular reasoning by analysis tools, we generally ignored it. Our findings are not generalizable to macro assembly.

Architectures. In our study, we focused on x86 inline assembly. However, when inline assembly was used for a particular use case, it was typically implemented for several common architectures (e.g., x86, ARM, and PowerPC). Most projects provided both x86-32 and x86-64 implementations, which were either the same or only slightly different (also see [30]). In rare cases, x86 lacked an inline assembly implementation that other architectures provided; for example, reversing the individual bits in an integer is available on ARM using the instruction `rbit`, with no equivalent x86 instruction. However, in general, we believe that we would have come to similar conclusions regarding the usage of inline assembly for other mainstream architectures.

GitHub. We performed the survey on open-source GitHub projects, and our findings might not apply to proprietary projects. Additionally, our findings might not be generalizable to older code, where inline assembly may be more frequent, since 89.1% of the projects we analyzed had their first commit in 2008 or later (the year GitHub was launched).

7 Related Work

To the best of our knowledge, inline assembly has to date attracted little research attention, and consequently we consider a wider context of related work.

Linux API usage. Our methodology was inspired by a study of Linux API usage which analyzed the frequency of system calls at the binary level to recommend an implementation order [64]. While we adopted a similar perspective, we analyzed the usage of inline assembly in C projects. Additionally, we directly analyzed the source code because we were interested in inline assembly usage independent of, for example, compilers and compiler versions.

Inline assembly and teaching. Anguita et al. discussed student motivation when learning about assembly-level machine organization in computer architecture classes [2]. In these classes, students were taught instructions that high-level languages lack (e.g., `cpuid` and `rdtsc`) and those that can improve the performance of a program (e.g., by prefetching data or using SIMD instructions). We found strong similarities between those instructions and the most frequently used inline assembly instructions, which further supports the validity of both studies.

Linker. Kell et al. studied the semantic role of linkers in C [34]. As with inline assembly, linker features are used in C programs, but transcend the language. Furthermore, some linker-relevant functionality, such as symbol versioning, is expressed in inline assembly.

C preprocessor. Ernst et al. explored the role of the C preprocessor [17]. As with linker features, the C preprocessor is also relied upon by C programs, but is not part of the language. They found that the preprocessor served—among other purposes—to include inline assembly.

Formal verification. Some formal verification approaches support inline assembly and/or macro assembly [5]. For example, Vx86 translates macro assembly to C code by abstracting its functionality [42]. Manual approaches assume that such inline assembly portions need to be converted to C functions [31]. Note that it is more straightforward to translate macro assembler, because C code mixed with assembler typically exchanges values between registers and variables.

Binary analysis. Tools that analyze or process binaries are widely established [3, 4, 11, 35, 46, 56] and could analyze C projects after they have been compiled to machine code. However, they are not always applicable, for example, when analyzing the high-level semantics of a program or when converting between different source languages.

8 Conclusion

We analyzed 1264 GitHub projects to determine the usage of inline assembly in C projects using both quantitative and qualitative analyses.

Our results demonstrate that inline assembly is relatively common in C projects. 28.1% of the most popular C projects contain inline assembly fragments, even when operating-system-level software, which might be more likely to contain inline assembly, is excluded. Inline assembly fragments typically consist of a single instruction, and most projects with inline assembly contain fewer than ten fragments. We found that the majority of projects use the same subset of instructions: by implementing 50 instructions, tool writers could support as much as 64.5% of all projects that contain inline assembly. 17.3% of the remaining projects use macro-metaprogramming techniques and/or many inline assembly fragments, for example, to benefit from SIMD instruction set extensions. By implementing the remainder of the total of 167 instructions and the SIMD instruction set extensions, tool writers could support the majority of projects "in the wild". Another challenge to implementing inline assembly is that invalid combinations of mnemonics are used that form valid opcodes when converted to machine code.

We found that inline assembly is often used in cryptographic applications. However, networking applications, media applications, databases, language implementations, concurrency libraries, math libraries, text processing and web servers also contain inline assembly. It is therefore likely that tools have to deal with inline assembly, even if they are intended for a specific domain. Inline assembly is used for multicore programming, for example, to implement compiler barriers, memory barriers, and atomics. It is employed for performance optimization, namely for SIMD instructions, endianness conversions, hash functions, and bitscan operations. Further, it is used when a functionality is unavailable in C, for example, for determining the elapsed clock cycles, for feature detection, debug interrupts, data prefetching, and generating secure random numbers. Finally, larger inline assembly fragments use moves, arithmetic instructions, and control flow instructions as "filler" instructions. Interestingly, the inline assembly syntax of compilers is not only used to insert instructions but also to control symbol names, linker warnings, symbol versioning, and register variables.

We believe that the results of our study are important to tool writers who consider adding support for inline assembly. Our study gives guidance on the need for such support and helps to plan and prioritize the implementation of instructions. Additionally, this study could be useful to language designers, as it reveals where plain C is inadequate to a task and where developers fall back on assembler instructions. Finally, compiler writers could obtain feedback on which instructions are frequently used, for example, to handle them specifically in compiler warnings [59] (e.g., by analyzing whether constraints and side effects are specified correctly).

9 Future Work

Our study opened up several directions for future work. One question is how inline assembly influences program correctness, since its use is error-prone; for example, undeclared side effects are not detected by state-of-the-art compilers and might remain as undetected faults or hard-to-debug errors in the source code. This question might be addressed by novel bug-finding tools that specifically target inline assembly. Similarly, an open question is whether compilers handle inline assembly correctly in every case. In recent years, random program generators for testing compilers [50, 60, 71] and other tools [14, 33] have been successful in identifying bugs. Future work could investigate whether generating programs with inline assembly could expose additional compiler bugs. While investigating inline assembly, we found that many programs use compiler intrinsics as an alternative to inline assembly. However, we did not investigate the usage of compiler intrinsics, which could be done as part of a future study. Finally, we believe that our study could be extended, for example, by investigating inline assembly in software (I) that is close to the machine (e.g., in operating systems), (II) in other languages (e.g., in C++), and (III) for other architectures (e.g., for ARM), and by investigating macro assembly.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions, and Ingrid Abfalter for her proofreading and editorial assistance. We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute of System Software at Johannes Kepler University Linz for their support and contributions. The authors from Johannes Kepler University Linz are funded in part by a research grant from Oracle. Kell was supported by EPSRC Programme Grant "REMS: Rigorous Engineering for Mainstream Systems", EP/K008528/1.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *Computer* 29, 12 (Dec. 1996), 66–76. https://doi.org/10.1109/2.546611

[2] Mancia Anguita and F. Javier Fernández-Baldomero. 2007. Software Optimization for Improving Student Motivation in a Computer Architecture Course. *IEEE Transactions on Education* 50, 4 (Nov 2007), 373–378. https://doi.org/10.1109/TE.2007.906603

[3] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. CodeSurfer/x86—A Platform for Analyzing x86 Executables. In *Proceedings of the 14th International Conference on Compiler Construction (CC'05)*. Springer-Verlag, Berlin, Heidelberg, 250–254. https://doi.org/10.1007/978-3-540-31985-6_19

[4] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst.* 32, 6, Article 23 (Aug. 2010), 84 pages. https://doi.org/10.1145/1749608.1749612

[5] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. 2009. Better avionics software reliability by code verification. In *Proceedings, embedded world Conference, Nuremberg, Germany*.

[6] Ryad Benadjila, Olivier Billet, Shay Gueron, and Matt J. Robshaw. 2009. The Intel AES Instructions Set and the SHA-3 Candidates. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology (ASIACRYPT '09)*. Springer-Verlag, Berlin, Heidelberg, 162–178. https://doi.org/10.1007/978-3-642-10366-7_10

[7] Daniel J. Bernstein. 2005. Cache-timing attacks on AES. (2005).

[8] binutils. 2017. Using as. (2017). https://sourceware.org/binutils/docs/as/index.html (Accessed October 2017).

[9] Hans-J. Boehm. 2005. Threads Cannot Be Implemented As a Library. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 261–268. https://doi.org/10.1145/1065010.1065042

[10] Hudson Borges, André C. Hora, and Marco Tulio Valente. 2016. Understanding the Factors That Impact the Popularity of GitHub Repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. 334–344. https://doi.org/10.1109/ICSME.2016.31

[11] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 213–223.

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224.

[13] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. 2014. Verifying Curve25519 Software. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 299–309. https://doi.org/10.1145/2660267.2660370

[14] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *Proceedings of the 4th International Conference on NASA Formal Methods (NFM'12)*. Springer-Verlag, Berlin, Heidelberg, 120–125. https://doi.org/10.1007/978-3-642-28891-3_12

[15] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2012. Understanding Integer Overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 760–770.

[16] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *Proceedings of the 2015 IEEE Security and Privacy Workshops (SPW '15)*. IEEE Computer Society, Washington, DC, USA, 73–87. https://doi.org/10.1109/SPW.2015.33

[17] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng.* 28, 12 (Dec. 2002), 1146–1170. https://doi.org/10.1109/TSE.2002.1158288

[18] David Evans, John Guttag, James Horning, and Yang Meng Tan. 1994. LCLint: A Tool for Using Specifications to Check Code. (1994), 87–96. https://doi.org/10.1145/193173.195297

[19] David Evans and David Larochelle. 2002. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Softw.* 19, 1 (Jan. 2002), 42–51. https://doi.org/10.1109/52.976940

[20] Jason Evans. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD. (2006). https://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf (Accessed October 2017).

[21] Robert Feldt and Ana Magazinius. 2010. Validity Threats in Empirical Software Engineering Research - An Initial Survey. In *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010), Redwood City, San Francisco Bay, CA, USA, July 1 - July 3, 2010*. 374–379.

[22] Mike Frysinger. 2015. Amd64 [un]fixes in SDL_endian.h. (2015). https://discourse.libsdl.org/t/amd64-un-fixes-in-sdl-endian-h/11792 (Accessed October 2017).

[23] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Thomsen Søren S. 2009. Grøstl - a SHA-3 candidate. In *Symmetric Cryptography (Dagstuhl Seminar Proceedings)*, Helena Handschuh, Stefan Lucks, Bart Preneel, and Phillip Rogaway (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2009/1955

[24] GCC Manual. 2017. Variables in Specified Registers. (2017). https://gcc.gnu.org/onlinedocs/gcc/Explicit-Register-Variables.html (Accessed October 2017).

[25] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 237–250. https://doi.org/10.1145/2908080.2908121

[26] Gerard J Holzmann. 2002. UNO: Static source code checking for user-defined properties. In *Proc. IDPT*, Vol. 2.

[27] Intel. 2010. How To Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. (2010). https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf (Accessed October 2017).

[28] Intel. 2014. Intel® Digital Random Number Generator (DRNG) Software Implementation Guide. (2014). https://software.intel.com/sites/default/files/managed/4d/91/DRNG_Software_Implementation_Guide_2.0.pdf (Accessed October 2017).

[29] International Organization for Standardization. 2011. ISO/IEC 9899:2011. (2011).

[30] Andreas Jaeger. 2003. Porting to 64-bit GNU/Linux Systems. In *Proceedings of the GCC Developers Summit*. 107–121.

[31] Rob Johnson and David Wagner. 2004. Finding User/Kernel Pointer Bugs with Type Inference. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, Berkeley, CA, USA, 9–9.

[32] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 92–101. https://doi.org/10.1145/2597073.2597074

[33] Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 590–600.

[34] Stephen Kell, Dominic P. Mulligan, and Peter Sewell. 2016. The Missing Link: Explaining ELF Static Linking, Semantically. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 607–623. https://doi.org/10.1145/2983990.2983996

[35] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. 2002. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 191–206.

[36] Robbert Krebbers and Freek Wiedijk. 2015. A Typed C11 Semantics for Interactive Theorem Proving. In *Proceedings of the 2015 Conference on Certified Programs and Proofs (CPP '15)*. ACM, New York, NY, USA, 15–27. https://doi.org/10.1145/2676724.2693571

[37] John B. Lacy. 1993. CryptoLib: Cryptography in Software. In *Proceedings of the 4th USENIX Security Symposium, Santa Clara, CA, USA, October 4-6, 1993*.

[38] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–88.

[39] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. *ACM Trans. Archit. Code Optim.* 9, 1, Article 2 (March 2012), 29 pages. https://doi.org/10.1145/2133382.2133384

[40] A. Liu and P. Ning. 2008. TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. In *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*. 245–256. https://doi.org/10.1109/IPSN.2008.47

[41] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéJàVu: A Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 84 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133908

[42] Stefan Maus, Michal Moskal, and Wolfram Schulte. 2008. Vx86: X86 Assembler Simulated in C Powered by Automated Theorem Proving. (2008), 284–298. https://doi.org/10.1007/978-3-540-79980-1_22

[43] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the De Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/2908080.2908081

[44] mrigger. 2017. Inline Assembler. (2017). https://github.com/elliotchance/c2go/issues/228 (Accessed October 2017).

[45] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. 213–228. https://doi.org/10.1007/3-540-45937-5_16

[46] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 89–100. https://doi.org/10.1145/1250734.1250746

[47] Lionel M. Ni and Kai Hwang. 1985. Vector-Reduction Techniques for Arithmetic Pipelines. *IEEE Trans. Comput.* C-34, 5 (May 1985), 404–411. https://doi.org/10.1109/TC.1985.1676580

[48] Joe Olivas, Mike Chynoweth, and Tom Propst. 2015. Benefitting Power and Performance Sleep Loops. (2015). https://software.intel.com/en-us/articles/benefitting-power-and-performance-sleep-loops (Accessed October 2017).

[49] John Regehr. 2013. Safe, Efficient, and Portable Rotate in C/C++. (2013). https://blog.regehr.org/archives/1063 (Accessed October 2017).

[50] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

[51] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL 2016)*. ACM, New York, NY, USA, 6–15. https://doi.org/10.1145/2998415.2998416

[52] Manuel Rigger, Roland Schatz, Matthias Grimmer, and Hanspeter Mössenböck. 2017. Lenient Execution of C on a Java Virtual Machine: Or: How I Learned to Stop Worrying and Run the Code. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes (ManLang 2017)*. ACM, New York, NY, USA, 35–47. https://doi.org/10.1145/3132190.3132204

[53] Manuel Rigger, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. Sulong, and Thanks For All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2018)*. https://doi.org/10.1145/3173162.3173174

[54] SDL. 2017. Simple DirectMedia Layer. (2017). https://www.libsdl.org/ (Accessed October 2017).

[55] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*. 309–318.

[56] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS '08)*. Springer-Verlag, Berlin, Heidelberg, 1–25. https://doi.org/10.1007/978-3-540-89862-7_1

[57] Henry Spencer and Geoff Collyer. 1992. #ifdef Considered Harmful, or Portability Experience with C News. In *USENIX Summer 1992 Technical Conference, San Antonio, TX, USA, June 8-12, 1992*. https://www.usenix.org/conference/usenix-summer-1992-technical-conference/ifdef-considered-harmful-or-portability

[58] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11,*

2015. 46–55. https://doi.org/10.1109/CGO.2015.7054186

[59] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and Analyzing Compiler Warning Defects. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 203–213. https://doi.org/10.1145/2884781.2884879

[60] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 849–863. https://doi.org/10.1145/2983990.2984038

[61] Piotr Szczechowiak, Leonardo B. Oliveira, Michael Scott, Martin Collier, and Ricardo Dahab. 2008. NanoECC: Testing the Limits of Elliptic Curve Cryptography in Sensor Networks. (2008), 305–320. https://doi.org/10.1007/978-3-540-77690-1_19

[62] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2012. Configuration Coverage in the Analysis of Large-scale System Software. *SIGOPS Oper. Syst. Rev.* 45, 3 (Jan. 2012), 10–14. https://doi.org/10.1145/2094091.2094095

[63] Lucas Torri, Guilherme Fachini, Leonardo Steinfeld, Vesmar Camara, Luigi Carro, and Érika Cota. 2010. An evaluation of free/open source static analysis tools applied to embedded software. In *2010 11th Latin American Test Workshop*. 1–6. https://doi.org/10.1109/LATW.2010.5550368

[64] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. 2016. A Study of Modern Linux API Usage and Compatibility: What to Support when You're Supporting. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 16, 16 pages. https://doi.org/10.1145/2901318.2901341

[65] VIA. 2005. New VIA PadLock SDK Extends Security Support in VIA C7®/C7®-M Processors for Windows and Linux Software Developers. (2005). https://www.viatech.com/en/2005/11/new-via-padlock-sdk-extends-security-support-in-via-c7c7-m-processors-for-windows-and-linux-software-developers/ (Accessed October 2017).

[66] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined Behavior: What Happened to My Code?. In *Proceedings of the Asia-Pacific Workshop on Systems (APSYS '12)*. ACM, New York, NY, USA, Article 9, 7 pages. https://doi.org/10.1145/2349896.2349905

[67] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 260–275. https://doi.org/10.1145/2517349.2522728

[68] Henry S Warren. 2013. *Hacker's delight.* Pearson Education.

[69] Deng Xu. 2011. [Frama-c-discuss] inline assembly code. (2011). https://lists.gforge.inria.fr/pipermail/frama-c-discuss/2011-March/002589.html (Accessed October 2017).

[70] Zhongxing Xu, Ted Kremenek, and Jian Zhang. 2010. A Memory Model for Static Analysis of C Programs. In *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*. 535–548. https://doi.org/10.1007/978-3-642-16558-0_44

[71] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[72] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 19, 11 pages. https://doi.org/10.1145/2503210.2503232

Appendix

Table 15 shows the instructions sorted by their frequency.

Table 15. Instruction table with instructions that were contained in at least 2 projects

instruction	# projects	% projects	instruction	# projects	% projects	instruction	# projects	% projects
rdtsc	54	27.4	test	9	4.6	aeskeygena	4	2.0
cpuid	50	25.4	jc	8	4.1	cld	4	2.0
mov	49	24.9	movdqa	8	4.1	ja	4	2.0
	43	21.8	shr	8	4.1	jbe	4	2.0
lock xchg	28	14.2	xgetbv	8	4.1	lock bts	4	2.0
pause	27	13.7	bsf	7	3.6	lods	4	2.0
lock cmpxchg	26	13.2	call	7	3.6	pclmulqdq	4	2.0
xor	25	12.7	inc	7	3.6	pslldq	4	2.0
add	21	10.7	int $0x03	7	3.6	psllq	4	2.0
bswap	18	9.1	jnc	7	3.6	psrldq	4	2.0
jmp	18	9.1	nop	7	3.6	rep stos	4	2.0
lock xadd	17	8.6	por	7	3.6	sar	4	2.0
pop	14	7.1	prefetch	7	3.6	setnz	4	2.0
push	14	7.1	setc	7	3.6	stos	4	2.0
cmp	13	6.6	dec	6	3.0	imul	3	1.5
mfence	13	6.6	lock add	6	3.0	lock or	3	1.5
mul	13	6.6	neg	6	3.0	lock sub	3	1.5
sfence	13	6.6	rdrand	6	3.0	movzb	3	1.5
sub	13	6.6	rep movs	6	3.0	pand	3	1.5
adc	12	6.1	crc32	5	2.5	pushf	3	1.5
bsr	12	6.1	lock dec	5	2.5	shrd	3	1.5
shl	12	6.1	lock inc	5	2.5	div	2	1.0
jz	11	5.6	movdqu	5	2.5	emms	2	1.0
lea	11	5.6	pshufd	5	2.5	fldcw	2	1.0
lfence	11	5.6	psrlq	5	2.5	int $0x80	2	1.0
or	11	5.6	rdtscp	5	2.5	jl	2	1.0
ror	11	5.6	ret	5	2.5	ldmxcsr	2	1.0
jnz	10	5.1	aesdec	4	2.0	lock and	2	1.0
setz	10	5.1	aesdeclast	4	2.0	popf	2	1.0
and	9	4.6	aesenc	4	2.0	punpcklb	2	1.0
pxor	9	4.6	aesenclast	4	2.0	punpckldq	2	1.0
rol	9	4.6	aesimc	4	2.0	stmxcsr	2	1.0

Chapter 10

A Study on the Use of GCC Builtins

This chapter includes a paper on our analysis on the use of GCC builtins in C projects and how well they are supported in tools.

Paper: Manuel Rigger, Stefan Marr, Bram Adams, and Hanspeter Mössenböck. Understanding GCC Builtins to Develop Better Tools, 2019. Under Review. Currently under review.

Understanding GCC Builtins to Develop Better Tools

Manuel Rigger
Johannes Kepler University Linz
Austria
manuel.rigger@jku.at

Stefan Marr
University of Kent
United Kingdom
s.marr@kent.ac.uk

Bram Adams
Polytechnique Montréal
Canada
bram.adams@polymtl.ca

Hanspeter Mössenböck
Johannes Kepler University Linz
Austria
hanspeter.moessenboeck@jku.at

***Abstract*—C programs can use compiler builtins to provide functionality that the C standard library lacks. On Linux, GCC provides several thousand builtins that are also supported by other mature compilers, such as Clang and ICC. Maintainers of other tools lack guidance on whether and which builtins should be implemented to support popular projects. To assist tool developers who want to support GCC builtins, we analyzed builtin use in 4,912 C projects that we obtained from GitHub. We found that 38% of these projects relied on at least one builtin. Supporting an increasing proportion of projects requires support of an exponentially increasing number of builtins; however, implementing only 10 builtins already covers over 30% of the projects. Since we found that many builtins in our corpus remained unused, the effort needed to support 90% of the projects is moderate, requiring about 110 builtins to be implemented. For each project, we analyzed the evolution of builtin usage over time and found that in most cases projects added calls to builtins. This suggests that builtins are not a legacy feature and must be supported in future tools. Systematic testing of builtin support in existing tools revealed that many lacked support for builtins either partially or completely; we also discovered incorrect implementations in various tools, including the formally verified CompCert compiler.**

***Index Terms*—GCC builtins, compiler intrinsics, GitHub**

I. Introduction

Most C programs consist not only of C code, but also of other elements, such as preprocessor directives, freestanding assembly code files, inline assembly, compiler pragmas, and compiler builtins. While recent studies have highlighted the role of linker scripts [1] and inline assembly [2], compiler builtins have so far attracted little attention. Builtins resemble functions or macros; however, they are not provided by libc, but are directly implemented in the compiler. The following code fragment shows the usage of a GCC builtin that returns the number of leading zeroes in an integer:

```c
int leading_zeroes = __builtin_clz(INT_MAX); // returns 1
```

On Linux, we observed that GCC builtins are widely used and seem to be supported also by other mature compilers, such as Clang [3] and ICC [4].

For developers working on tools that process C code, implementation and maintenance of GCC builtins is a large effort, as we identified a total number of 12,126 GCC builtins, all of which are potentially used by projects. Hence, to assist developers of tools that process C code, the goal of this study was to investigate the use of builtins and how current tools support them. To this end, we analyzed the builtin use of 4,912 projects from GitHub and implemented a builtin test suite, which we used to test popular tools employed by C developers. By combining quantitative and qualitative analyses, we could answer the following research questions (RQs):

RQ1: How many builtins do exist? Answering this question can help tool writers to estimate the effort of providing complete support for GCC compiler builtins. We initially thought that we could obtain a list of builtins from the documentation or source code. However, GCC's organic growth has led to some builtins being omitted from the documentation, and others are considered internal, even when they are widely relied upon by other projects.

RQ2: How frequently do projects use builtins? Knowing the prevalence of builtins helps tool writers to judge the importance of implementing support for them. We hypothesized that builtins are used by many projects, and that any program that processes C code will therefore encounter them, yet—similar to inline assembly [2]—we expected that they are used in only a few source-code locations.

RQ3: For what purposes are builtins used? Knowing the primary use cases for builtins helps tool developers to judge whether their tools can support them. For example, static analysis tools might lack support for multithreading and hence be unable to deal with atomic builtins used for synchronization.

RQ4: How many builtins must be implemented to support most projects? Tool authors who have decided to support GCC builtins would find it helpful to know the implementation order that would maximize the number of projects supported at a given implementation stage.

RQ5: How does builtin usage develop over time? Understanding the usage of builtins over time could tell us whether projects continue to add builtins or remove them. If builtins were a legacy feature of compilers that projects sought to remove, the incentive of tool developers to implement them would be low.

RQ6: How well do tools support builtins? To determine the room for improvement in tools, we examined how well existing tools support builtins. Our assumption was that state-of-the-art compilers such as GCC, Clang, and ICC provide full support, while other tools provide partial or no support.

We found the following:

- 12,126 GCC builtins exist, but only 3,084 were used in our corpus of projects;
- 38% of the projects used builtins. When projects used architecture-specific builtins, they were often used in large numbers.
- Projects primarily used architecture-independent builtins, for example, to interact with the compiler, for bit-level operations, and for atomic operations.
- While mature compilers seem to provide full support for builtins, most other tools lack some builtins or have some incorrectly implemented. Notably, we found two incorrectly implemented GCC builtins in an unverified part of the formally verified CompCert compiler.
- The effort of supporting a specific number of projects is exponential; for example, to support half of the projects only 32 builtins are needed. Supporting 99% of the projects, however, requires about 1,600 builtins.
- Over time, most of the projects increasingly used builtins; nevertheless, a number of projects removed builtin usages to reduce maintenance effort.

Our results are expected to help tool developers in prioritizing implementation effort, maintenance (including deprecation of unused builtins), and optimization of builtins. Thus, this study facilitates the development of compilers such as GCC, Clang [5], ICC, and the formally verified CompCert compiler [6], [7]; of static-analysis tools such as the Clang Static Analyzer [8], splint [9], [10], Frama-C [11], and uno [12]; of semantic models for C [13], [14]; and of alternative execution environments and bug-finding tools such as KLEE [15], Sulong [16], [17], the LLVM sanitizers [18], [19], and SoftBound [20], [21]. We believe that our results are also useful to language designers, as they show which functionality plain C lacks, and can help with testing compilers [22], [23], [24] and other tools that process C code [25], [26]. For reproducibility and verifiability, we provide the database with GCC builtin usage, test suite, tools used for the analysis, and an online appendix with more details on a website yet to be determined.

II. Methodology

To answer our research questions, we analyzed builtin usage in a large number of C projects and populated a SQLite3 database with the extracted data. This section explains how we selected the projects, filtered them, and searched for builtins.

Selecting the Projects. We analyzed projects from GitHub, a code-hosting service. We downloaded all C projects starting from 80 GitHub stars, an arbitrary cutoff value that yielded a large number of projects for our study. As the number of stars is a metric for popularity [27], this cutoff point prevented the inclusion of personal projects, homework assignments, and forks [28]. In total, we downloaded 4,997 GitHub projects that contained in total 1,124 million lines of C code and occupied 409 GB of disk space. This strategy allowed us to obtain a diverse set of projects (see Table I).

Filtering the Projects. From the downloaded projects, we selected 4,912 by filtering out those that did not meet our needs. First, we filtered out all projects that had fewer than 100 LOC, as we considered them too small to constitute C projects. GCC, forks of GCC[1], and other compilers (such as ROSE [29]) implement the GCC builtins themselves, use them internally, and exercise them in their test suites. To avoid a high number of false positives, we excluded these projects; they were easy to identify, as they had the most unique builtins.

TABLE I: Overview of the projects obtained (after filtering); the first commit in 1984 stems from a project that was converted from another version-control system.

Metric	Minimum	Maximum	Average	Median
C LOC	100	37M	228k	10k
# commits	1	668k	4873	1147
# committers	2	17k	121	55
first commit	1984-02-21	2017-11-06	-	2011-04-12
last commit	2003-12-08	2017-11-24	-	2017-11-07

Searching within the Projects. We searched all C files for the names of 12,126 builtins described by the GCC documentation and used in the GCC source code (see Section III-A). Note that we considered only occurrences where the builtin name was not part of another identifier. For each builtin that we found, we created a record in our database, thus obtaining 630k builtin entries.

Filtering the Builtin Records. We used several strategies to eliminate false positives in the builtin records. While investigating the projects with the highest numbers of unique builtins—mostly operating systems—we found that many of them included the source code of Clang or GCC. As with GCC forks, we excluded directories that started with `gcc`, `clang` or `llvm` (excluding 45% of our records); however, we continued to analyze source files in other directories of such projects. We excluded builtin occurrences that were enclosed in double quotes, as this indicates that they are part of a string (excluding 1% of the records). To exclude builtins in comments, we did not consider builtins found in lines that started with `/*`, `*`, or `//` (which excluded 2% of the records). Finally, we skimmed over the builtin occurrences and created a list of 4,026 one-line code fragments that indicated false positives, such as inline assembly with an instruction mnemonic that corresponded to a builtin name (excluding 1% of the records). In total, these measures reduced the number of records to 320k (51% of the original number). Note that the exclusion criteria overlapped for some records.

III. Results

To answer our research questions, we analyzed the data gathered as follows.

A. *RQ1: How many builtins do exist?*

To determine the number of builtins in GCC, we investigated (I) builtins listed in the GCC documentation and (II)

[1]The projects filtered out included the GCC fork for the Xtensa processor (https://github.com/jcmvbkbc/gcc-xtensa), and a fork that is based on GCC to dump an XML description of C++ code (https://github.com/gccxml/gccxml).

builtins internal to GCC, which we obtained from GCC's source code (including test cases for its builtins).

(I) Builtins from the documentation. Initially, we assumed that we could answer RQ1 by extracting the list of builtins from the GCC documentation. The GCC documentation stated that some builtins are internal, which we did not want to include as we expected that other projects would not use them. Extracting the builtins was a best-effort approach, since the list of builtins in the GCC documenation appeared to be manually derived and contained duplicates and errors. This worked well with architecture-independent builtins, but GCC also provides builtins that are specific to an architecture. For example, `__builtin_ia32_paddq` allows the use of x86's `paddq` instruction. In some cases, architecture-specific builtins were not described by the documentation, but referred to vendor documentation, for example, the ARM C Language Extensions. For these builtins, the documentation of GCC version 4.8 contained a list of builtins, which we used instead. However, in some cases, obtaining such a list was impractical, for example, for the TILE-Gx and TILEPro processor builtins. As we expected little influence on the results—overall, architecture-specific builtins were used infrequently (see Section III-C)—we omitted analyzing such builtins. In total, this process yielded 6,040 builtins, of which 560 were architecture-independent and 5,480 were architecture-specific.

(II) Builtins from the GCC source code To verify that we did not omit any commonly used builtins, we searched the projects for strings starting with `__builtin_`. We found that many projects relied on a small number of GCC's internal (i.e., undocumented) builtins. We assumed that tool developers also need to support these builtins, and added them to our search terms by including all additional `__builtin_` functions that we found in the GCC source code and test suite (6,067 additional builtins). In a number of cases, GCC implemented public builtins using undocumented internal builtins; this was a potential problem in our study, as public and internal builtins would be counted as separate even if they implemented the same semantics. However, since the number of internal builtins actually used was relatively small, we did not attempt to match public builtins with internal ones in our quantitative analysis.

In total, we considered 12,126 builtins in our analysis.

B. *RQ2: How frequent are builtins?*

To answer RQ2, we considered both duplicate and unique builtin usages per project. Counting usages—even if they were duplicated within a project—allowed us to measure the overall prevalence of builtin use. Counting project-unique usages better reflected the implementation effort needed to support a project, because duplicates do not increase the implementation effort.

Overall usage. In total, 1,847 of the projects (38% of all projects) used a common subset of 3,084 builtins. The frequency of compiler builtins varied strongly, depending on the project, and ranged from one builtin every 7 LOC to one every 1,680,582 LOC. The median frequency of builtins was one every 5,737 LOC (on average one builtin every 20846 LOC). Figure 1 shows boxplots to illustrate the builtin usage by the projects, and breaks their usage up into architecture-specific and architecture-independent usages, considering both unique and non-unique builtin occurrences within a project.

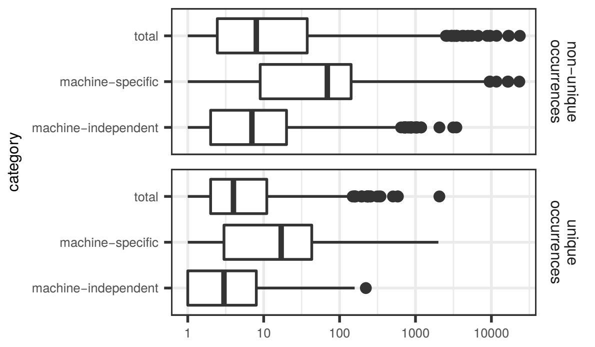

Fig. 1: Number of architecture-specific and architecture-independent builtins per project that used builtins of the respective category. The plots use a logarithmic scale to accommodate the large variations between projects.

Non-unique occurrences. The median number of builtin calls in a project that used builtins was 9, the average was 174. Where they were used in projects, architecture-specific builtins were employed in greater numbers (median = 69); in contrast, when architecture-independent builtins were used, their numbers were far lower (median = 7). However, since use of architecture-specific builtins was less common (see Section III-C), the overall result is dominated by the architecture-independent builtins. We investigated the 15 projects with the highest numbers of builtins and found that audio/video players and codecs lead the ranking (9/15), followed by operating systems (3/15), a game engine, a software library specialized for ARM processors, and a libc implementation.

Unique occurrences. Of the 320k builtin calls, 30k were project-unique; that is, the others were duplicated within a project. The median number of unique builtins used by projects with builtins was low, with a median of 4 and an average of 17. As with non-unique builtins, projects that used architecture-specific builtins had more such builtins (median = 17) than projects that used architecture-independent builtins (median = 3). The projects with the highest numbers of unique builtins were, again, in most cases audio/video players and codecs (6/15). However, operating systems (2/15), game engines (2/15), language implementations and compilers (2/15), a messenger, and an image codec also ranked among the top 15.

Reoccurring files. We observed that files with particular names, primarily header files, were more likely to contain calls to builtins. One reason for this was that, consistent with findings by Lopes et al. [30], files were copied from other projects. The majority of these files originated either from the GNU C library `glibc` or from Linux-based operating systems. While they were used primarily in operating system implementations, they were also copied to projects with application code. As another example, the frequently used `sqlite3.c` and `SDL_stdinc.h` files even contained the projects' names as part of the file name: SQLite is a popular database, and SDL a commonly used media library. In other cases, duplicate file names indicated the use case for the builtin usage. For example, builtin-based atomicity support was often implemented in files named `atomic.h`, and math builtins were used in files named `math.h`.

C. RQ3: For what purposes are builtins used?

To identify the purpose for which builtins are used, we explored their usage at different levels of granularity: First, we considered the differences in usage between architecture-independent and architecture-specific builtins. We subsequently examined the usage of architecture-independent builtins and of architecture-specific builtins in more detail. The results are summarized in Figure 2. Finally, we analyzed builtins that remained unused in our corpus.

Architecture-specific and -independent builtins. The GCC documentation categorizes builtins into architecture-specific and architecture-independent ones, which we used as a basis for discussion. While 1,779 projects used at least one architecture-independent builtin, we found architecture-specific builtins in only 424 projects. That architecture-independent builtins are more common was unexpected, since we found only 86k architecture-independent builtin uses, but 214k architecture-specific ones. However, as discussed in Section III-B, a project using architecture-specific builtins is likely to use more such builtins than projects that use architecture-independent builtins.

Usage of architecture-independent builtins. The builtin category "other", which contained miscellaneous builtins, was the most common category of GCC builtins, even though it comprised only 68 builtins—21 of which were among the 50 most frequently used. Since these builtins were the most common, we further analyzed their use, and classified them into the following subcategories: (I) direct compiler interaction, (II) bit and byte operations, (III) special floating-point values, and (IV) dynamic stack allocation.

(I) Direct compiler interaction. Some builtins allow direct interaction with the compiler, for example, to improve performance; the most frequently used builtin was `__builtin_expect`, which communicates expected branch probabilities to the compiler, which can exploit this information for optimization. The `__builtin_unreachable` builtin can be used to silence warnings by informing the compiler that code is unreachable, which is useful when the compiler cannot deduce this. Some of the builtins in this subcategory can also be used for metaprogramming; the `__builtin_constant_p` builtin is resolved at compile time and allows programmers to query whether a pointer is known by the compiler to be constant. As another example, `__builtin_types_compatible_p` queries whether two input types passed to the builtin are the same. Plain C does not offer similar functionality.

(II) Bit and byte operations. Some builtins process integers at the level of bits and bytes. The second-most frequently used builtin was `__builtin_clz`, which counts the leading zeroes in an `unsigned int`; its variants for other data types also ranked among the most commonly used builtins overall. Similarly frequent were builtins for computing the position of the least significant one-bit, for counting the number of one-bits in an integer, and for reversing the bytes of an integer. We believe that these builtins were used for convenience and performance optimizations, as the same functionality could be implemented in plain C.

(III) Special floating-point values. Some builtins generate special values for various floating-point types. For example, the `__builtin_inf` builtin generates a positive infinity `double` value. As another example, `__builtin_nan` returns a not-a-number value. Recent C standards specify macros and functions for obtaining such values.

(IV) Dynamic stack allocation. The `__builtin_alloca` builtin allocates the specified number of bytes of stack memory. Since C99, variable length arrays have offered a similar functionality, as the size of an allocated array can depend on a run-time value.

Synchronization and atomics. After "other", the next common builtin category was synchronization ("sync") with 11 of the 50 most common builtins. In this category, the most frequently used builtin was `__sync_synchronize`, which issues a full memory barrier to restrict the order of execution in out-of-order CPUs. Builtins for atomically executing operations were also common (e.g., `__sync_fetch_and_add`). These builtins were designed for the Intel Itanium ABI and were deprecated in favor of the builtins contained in the "atomic" category. The builtins in the "atomic" category additionally allow specifying the memory order of the operation, but were not that frequently used; nevertheless 7 builtins of this category ranked among the 100 most common builtins. Note that C11 introduced synchronization primitives, which are alternatives to these builtins.

Libc functions. GCC provides builtins for many functions of the standard C library—4 such builtins were the 100 most common builtins. An example is `__builtin_memcpy`, which implements the semantics of `memcpy`. The builtin version of the libc function is useful when compiling a program assuming a C dialect in which a function is not yet available; for example, when compiling under the C90 standard (`-std=c90`), the newer C99 function `log2` cannot be used; however, the prefixed version `__builtin_log2` can still be used. Furthermore, they enable bare-metal programs, which are compiled *freestanding* and therefore do not have access to libc functions, unless they use compiler builtins.

GCC internal functions. Several builtins were used by projects although they were not documented—4 ranked among the top 100 frequently used builtins. These most frequently used builtins, namely `__builtin_va_start`, `__builtin_va_end`, `__builtin_va_arg`, and `__builtin_va_copy`, were used only to implement the vararg macros of the C standard; for example,

```
#define va_start(v,l) __builtin_va_start(v,l)
```

Function return address and offsetof. The "introspection" category—with 3 of the top 100 builtins—enables programmers to query (I) the address to which a function returns and (II) the address of the current frame (i.e., the area where local variables are stored). To this end, GCC provides `__builtin_return_address`, `__builtin_frame_address` and other builtins. Another, similar category is "offsetof" with a single builtin `__builtin_offsetof`, which was one of the top 100 builtins. It determines the offset of a struct or array member from the start address of the struct or array.

Object size and safe integer arithmetics. The builtin `__builtin_object_size` in the "object-size" category enables programmers to query the size of an object, which is useful when implementing bounds checks. To implement this builtin, GCC relies on static analysis to determine the size of an object where possible. The "overflow" category—of which no builtin ranked among the top 100—provides wraparound semantics for overflow in signed-integer operations (e.g., `__builtin_add_overflow` for addition), which would otherwise induce undefined behavior in C [31].

Usage of architecture-specific builtins. Of the 100 most-frequent builtins, 44 were specific to an architecture. Most frequent were the builtins for the PowerPC family—17 of which were among the top 100 builtins. The most frequent PowerPC builtins were those implementing vector operations such as `vec_perm`, which implements a vector permutation. The second category were ARM C NEON extensions—25 of which were among the top 100 builtins—that also implement vector operations. On x86, which ranked next, the most common builtin was `__builtin_cpu_supports` followed by `__builtin_cpu_init`, which allow programmers to query the availability of CPU features such as SIMD support. In x86-64 inline assembly, the equivalent `cpuid` instruction ranked among the most commonly used instructions [2]. Other x86 builtins were quite diverse and less frequent. For brevity, the less frequently used architecture-specific builtin categories are omitted. However, they are included in the full list of commonly used builtins in the online appendix.

Unused builtins. To identify unused builtins, we considered only those described in the GCC documentation (i.e., the public ones). Surprisingly, we found that half of them, namely 3,030 (50%), were not used in our corpus. The distribution differed between architecture-specific and architecture-independent builtins. From the architecture-independent builtins, 379 of 560 were used, which corresponds to 32% unused builtins. We characterize these builtins below. From the architecture-specific builtins, only 2,630 of 5,480 builtins were used, which means that more than half of them (52%) were not used in any project; this is why we do not characterize them in detail.

We contacted the GCC developers to report our findings[2]; they responded that builtins could not be removed from the documentation due to vendor guarantees (for architecture-specific builtins) and because they might still be used in closed-source software or by projects not hosted on GitHub. Note that builtins that are used only internally could still be removed from the public documentation, which explicitly states that it does not document such builtins.

Unused architecture-independent builtins. None of the projects used any of the 11 bounds-checking builtins for controlling the Intel MPX-based pointer-bounds-checker instrumentation, which is based on a hardware extension in Intel processors. One reason for this is that they are used by a pass within GCC and have received only little further attention [32], as Intel MPX-based approaches perform only about as fast as pure software approaches [33]. Four of the object-size-checking builtins were not used, namely a subset of those for printing format strings (e.g., `__builtin___vfprintf_chk`). The builtins of this category were derived from library functions (e.g., `memcpy`), but require an additional size argument (e.g., `__builtin___memcpy_chk`). The intended use of these builtins is to prevent buffer overflow attacks, since object accesses that exceed the size of the object can be prevented. We speculate that these builtins were not frequently used because such checks can only be enforced completely at run time [34].

None of the 13 builtins of the Cilk Plus C/C++ language extensions [35], which offer a mechanism for multithreading, were used. In 2017—the year this study was conducted—Cilk Plus was deprecated, and in November 2017 GCC removed its implementation[3]. Of the prefixed libc functions, 37% were unused. Most programs are probably compiled in hosted mode, where compilers can substitute calls to the libc functions with these builtins. Another reason could be that some of them are used only internally. Nevertheless, they were documented in the public API.

Of the unused builtins in the "other" category, the majority were narrowly specialized builtins such as `__builtin_inffn`, which generates an infinity value for the data type `_Floatn`. Further, `__builtin___clear_cache` for flushing the processor's instruction cache remained unused. The unused `__builtin_call_with_static_chain` enables calls to languages that expect static chain pointers, such as Go.

D. *RQ4: How many builtins must be implemented to support most projects?*

In order to provide tool developers with a recommended implementation order for builtins, we considered two implementation scenarios. The first scenario considered all builtins as implementation candidates. The second considered only architecture-independent builtins, which can be relevant when only a subset of architectures is to be supported. Additionally, we assumed two pragmatic strategies for the order of implementation: an order based on the frequency of builtins, and one based on a greedy algorithm.

[2]https://gcc.gnu.org/ml/gcc/2018-01/msg00166.html

[3]https://gcc.gnu.org/ml/gcc-patches/2017-11/msg01345.html

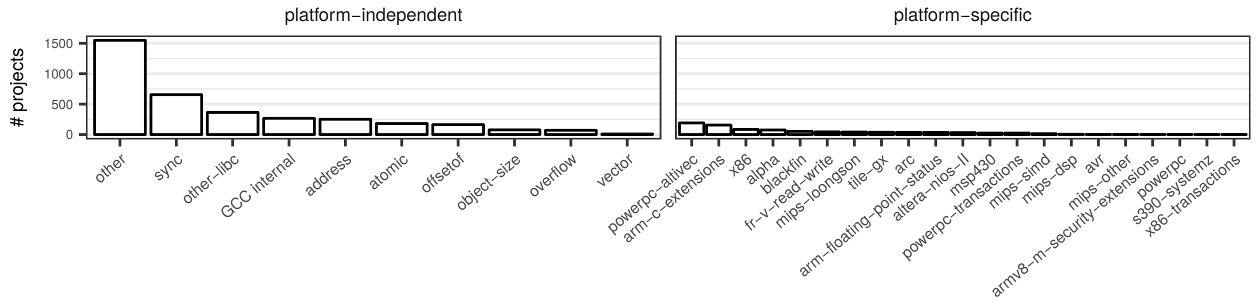

Fig. 2: The number of projects that rely on architecture-specific and architecture-independent builtins.

Frequency order. Using this strategy, we assumed that the builtins used by the highest number of projects are to be implemented first. Thus, this strategy follows the order given by Table II. This order is not generally optimal, because it does not take into account that, in order for a project to be supported, all builtins used must be implemented.

Greedy order. For rapid experimentation, it can be beneficial to quickly support as many projects as possible. To this end, we implemented a greedy order where the next builtin to be implemented is selected such that it enables support of the largest number of additional projects. If no such builtin exists, the next builtin is selected using the frequency order.

Results. Implementing builtins takes an exponential implementation effort in terms of number of builtins that must be implemented to support a specific number of projects (see Figure 3). The greedy order for implementing builtins performs better than the frequency order, a trend that is more clear-cut when considering all builtins rather than just architecture-independent ones. To support half of the projects, in both scenarios and using both strategies, no more than 32 builtins need to be implemented. However, supporting 90% of the projects requires 106 builtins to be implemented for the greedy approach and 112 builtins for the frequency strategy when considering only architecture-independent builtins. When considering all builtins, more than 850 builtins must be implemented for the frequency strategy, and more than 600 for the greedy strategy. To support 99% of the projects, the greedy algorithm is better: when considering only architecture-independent builtins, around 250 instead of 300 builtins must be implemented, compared to 1,600 instead of 3,000 builtins when considering all builtins.

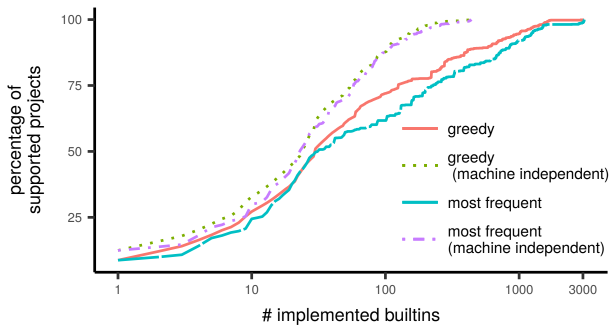

Fig. 3: The numbers of builtins needed to support an increasing number of projects increases exponentially; note the exponential x-axis.

TABLE II: The 10 most frequent builtins.

builtin	category	projects
__builtin_expect	other (compiler interaction)	891 / 48.2%
__builtin_clz	other (bitwise operation)	542 / 29.3%
__builtin_bswap32	other (bitwise operation)	483 / 26.2%
__builtin_constant_p	other (compiler interaction)	431 / 23.3%
__builtin_alloca	other (stack allocation)	375 / 20.3%
__sync_synchronize	sync	357 / 19.3%
__builtin_bswap64	other (bitwise operation)	347 / 18.8%
__sync_fetch_and_add	sync	333 / 18.0%
__builtin_ctz	other (bitwise operation)	324 / 17.5%
__builtin_bswap16	other (bitwise operation)	303 / 16.4%

E. RQ5: How does builtin usage develop over time?

To understand whether builtin usage is an ongoing concern of software projects or just a form of technical debt (introduced temporarily before being removed), we studied the development of builtin usage over time in the projects that used builtins. For this, we analyzed all commits by iterating from the latest commit to the oldest commit—including merge commits (represented by the union of all commits that are merged)—always by following the first parent (i.e., staying on the master branch). We analyzed 1,839 projects (100% of all projects that used builtins); the rest could not be processed in a feasible amount of time.

We first classified projects manually as having an *increasing*, *decreasing*, or *inconclusive* trend of builtin usage (see Table III). Amongst others, we considered those trends as inconclusive whose projects had just 5 or fewer commits that added or removed builtin calls; surprisingly, this was the case for the majority of the projects (1,145)[4]. We then improved our classification iteratively by forming subcategories. To find reasons for changes in the numbers of builtins, we analyzed commit messages and commit changes, then identified common cases. Finally, we selected four projects

[4] More than 80% of them only added builtins and never removed any. If we counted these projects, the increasing trend would be more significant.

TABLE III: Builtin trends in projects.

trend	classification	#/% builtins		median commits
Increasing	mostly increasing	175	10%	11
	monotonically increasing	55	3%	6
	increasing, with stable parts	88	5%	10
Decreasing	increasing, then decreasing	97	5%	9
Inconclusive	fewer than 5 commits	1,145	64%	1
	no clear pattern	179	10%	10
	initial commit	48	3%	12

that were representative of the most important trends in order to qualitatively explain the development of builtins in them.

Classification. We found that the majority of projects with a conclusive trend showed increasing usage of builtins. 55 projects only added builtins, but never removed any; however, the number of builtin-related commits was low in this category, with a median of 6 of such commits. 175 projects—the highest number in a conclusive subcategory—had a mostly increasing usage of builtins, which was the case for many large projects; this is also demonstrated by the high median number of builtin-related commits. Finally, we identified 88 projects with an increasing trend, but either long periods without builtin-related commits or only marginal increases.

Decreasing trends—preceded by an increasing trend—were relatively rare (97 projects), which suggests that a significant number of builtins was removed only in a few cases. 179 projects varied between increasing and decreasing trends, so we could not characterize their development. 48 projects were left uncharacterized as they added a significant number of builtins as part of their first commit. This is typical for cases where an existing project was imported in a version control system without its prior commit history.

Reasons for builtin additions. The majority of sharp increases in the number of builtins was caused by the inclusion of third-party libraries that call builtins internally, as indicated by commits such as *"update packaged sqlite to 3.8.11.1"* or *"Added latest stb_image."* In some cases, only single existing header files were included, as indicated by commit messages such as *"add atomic.h that wraps GCC atomic operations"* or *"Copy over stdatomic.h from freebsd."*

Builtins, both architecture-specific and -independent ones, were used for performance optimizations. Architecture-independent optimizations were described as *"popcount() optimization for speed"* (using `__builtin_popcount`), *"Use __builtin_expect in scanline drawers to help gcc predict branching"*, and *"A prefetch of status->last_alloc_tslot saved 5%"* (using `__builtin_prefetch`). Examples of architecture-specific builtin commits were *"VP9 common for ARMv8 by using NEON intrinsics"* and *"30% encoding speedup: use NEON for QuantizeBlock()"*.

Builtins were also used because they conveniently supported required functionality in commits such as *"bitmap – Add few helpers for [bit] manipulations"*. They were often used for atomics, as in *"GCC 4.1 builtin atomic operations"* and *"Adding atomic bitwise operations api and rwlocks support"*. They enabled metaprogramming techniques, for example, by enabling macros to handle various data types: *"util: Ensure align_power2() works with things other than uint. This uses a casacading set of if (__builtin_types_compatible_p()) statements to pick the correct alignment function tailored to a specific type [...]"*.

Finally, builtins were employed to reduce the usage of inline assembly in commits such as *"avoid inline assembly in favor of gcc builtin functions"* and *"Padlock engine: make it independent of inline assembler."*, or as an alternative to architecture-specific system libraries, such as *"alloca fallback for gcc"*, which added a usage of `__builtin_alloca` when the platform did not provide a header file that implements `alloca`.

Reasons for builtin removals. Removals of third-party libraries accounted for the most significant number of removals of builtins, as indicated by commits such as *"Remove thirdparties"* or *"Removed outdated headers and libraries."* Individual files or functions that used builtins were removed as side effects of refactoring or cleanup in commits with messages such as *"General cleanup of the codebase, remove redundant files."* or *"tools: Remove unused code."* Auto-generated files were removed, for instance, in the commit *"Removed getdate.c as it is regenerated from getdate.y"*.

A number of removals were related to technical debt [36]. Projects removed builtins for old architectures for which they dropped support, for instance, in *"avr32: Retire AVR32 for good. AVR32 is gone. [...]"* or *"Blackfin: Remove. The architecture is currently unmaintained, remove"*. In other cases, builtins for certain architectures were removed due to their maintenance effort: *"Remove support for altivec using gcc builtins, since these keep changing across gcc versions. [...]"*. Usages of builtins were hidden behind a macro, to concentrate their usage to a single location in the source code: *"Convert remaining __builtin_expect to likely/unlikely [...]"* (for `__builtin_expect`) and *"Use the new sol-atomic.h API instead of directly GCC intrinsics"* (for atomic operations).

In other cases, a usage of `__builtin_expect` was removed because it did not improve performance: *"[...] It had no reliably measurable performance improv[e]ment, at least on an i7 960 and within a microbenchmark."*.

Case study. Finally, we examined the builtin development in four projects whose trends we considered both representative and insightful for our case study (see Figure 4). First, we selected libucl, a configuration library parser, which is representative of the *increasing* trend. Like the majority of projects that we examined, it added a small number of builtin calls for various tasks. We selected libav, a collection of cross-platform tools to process multimedia formats and protocols, to represent the *decreasing* category (increasing, then decreasing). As is typical of a media library, it contained a number of builtin-related commits that improved performance by adding calls to architecture-specific builtins, but also systematically removed them to reduce maintenance effort. We selected cpuminer,

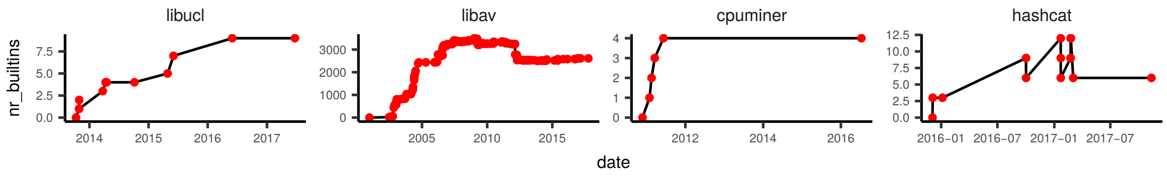

Fig. 4: Builtin development in cpuminer, hashcat, libav, and libucl.

an application for mining bitcoins, as a representative of projects with *fewer than five commits*, the class to which the majority of projects belonged. As a second representative of the inconclusive category, we selected hashcat, a tool for password recovery, which we classified as having *no clear pattern*.

libucl (mostly increasing). The builtin additions in libucl were in most cases related to hashing. The first two builtin-related commits of libucl imported a hash algorithm from third-party libraries which used `__builtin_clz` and `__builtin_swap32` in their hashing computations. Subsequently, a third-party library hashing implementation was replaced with a custom implementation, removing a builtin usage. Subsequent commits were also related to finding better hashing algorithms, resulting in additions of calls to byteswap builtins and checks for SIMD support using `__builtin_cpu_init` and `__builtin_cpu_supports`. Additionally, the library added a reference-counting scheme to free memory when an allocation is no longer referenced, whose implementation depended on atomics.

libav (increasing then decreasing). In the first half of libav's development, its usage of builtins mainly increased, mostly due to Altivec-specific builtins used to optimize computation-intensive operations, but also due to architecture-specific builtins of other architectures such as PowerPC or ARM. In a few cases, calls to architecture-independent builtins were added, for example for atomics. In the second half of the project, refactorings reduced the number of builtin calls. In 2009, calls to 236 Altivex-specific builtins were removed to reduce technical debt and improve the maintainability of the Snow codec (which was removed in 2012): *"Remove AltiVec optimizations for Snow. They are hindering the development of Snow, which is still in flux."* In 2012, calls to 233 builtins were removed as part of a cleanup that dropped an unused function; in the same year, a library was removed that used 469 builtins. In 2013, another smaller, but interesting, commit removed calls to 23 Alpha-specific builtins, as the platform was no longer considered important: *"Remove all Alpha architecture optimizations. Alpha has been end-of-lifed and no more test machines are available."*

cpuminer (Fewer than five commits). Cpuminer had four builtin-related commits in the initial stage of the project. Two of them introduced macros for performance optimizations that used `__builtin_expect` to communicate branch probabilities to the compiler. The other two commits added usages of `__builtin_alloca` and `__builtin_bswap32`. They were included only when the `alloca.h` and `byteswap.h` header files were not available on the given platform. Until the latest commit in 2017, no further builtin calls were added or removed.

hashcat (no clear pattern). In hashcat, an initial commit added a helper macro that wrapped `__builtin_bswap32`. Subsequently, this logic was reimplemented in the file `bitops.c`—which contained fallbacks in plain C—and the macro was removed, causing the first drop in builtin usage. The project added more hash modes, and byteswap macros that directly used GCC builtins were added in `interface.c` (accounting for the first maximum). However, they caused problems and were replaced with the byteswap implementations in `bitops.c`: *"Fix travis-ci error caused by __builtin_bswapXX()"*. The author eventually decided to remove the usage of GCC builtins altogether and to use a plain C implementation instead: *"Simply do not use __builtin_bswap16() this causes all kinds of problems, use our own implementation"*.

The next series of commits was related to overflow checking. The project added `__builtin_mul_overflow` to sanitize user input, and another commit added checks with `__builtin_add_overflow` (resulting in a second maximum). However, portable solutions seemed to be preferred, as a commit reduced the use of overflow-related builtins by employing C code. However, to extract the number of leading zeroes, `__builtin_clz` and `__builtin_clzll` were added: *"Add some compiler independent integer overflow functions"*. Ultimately, these were also replaced with plain C code: *"Replace __builtin_clz() and __builtin_clzll() with some straightforward solution"*.

Discussion. The four representative projects gave insights into how projects added and removed builtin usages. Like the majority of projects we examined, libucl, cpuminer, and hashcat had few commits related to architecture-independent builtins. These builtins were used in various use cases, for instance, to improve the performance of code using `__builtin_expect`, to test for CPU features, to implement hash computations, and as a fallback when architecture-specific builtins were missing. Libav was one of the relatively few projects that had a large number of commits related to architecture-specific builtins, and it reduced their number during code refactorings. For hashcat and libav, builtins were also removed to reduce technical debt; in hashcat, builtins were quickly replaced by implementing their functionalities in C, and in libav they were removed since an outdated architecture was no longer supported.

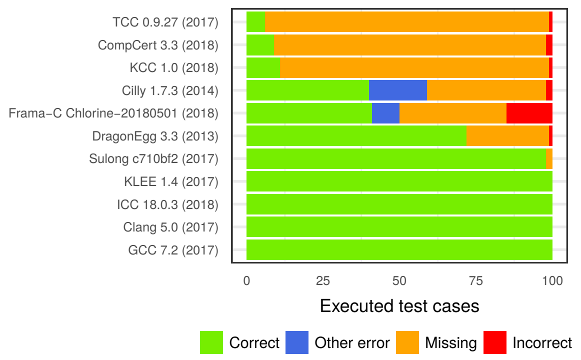

Fig. 5: The test-case results before bugs were fixed or builtins implemented. The order of the legend items corresponds to the order of the stacked plots. "Other errors" refers to failed test cases that are unrelated to builtin support.

F. *RQ6: How well do tools support builtins?*

To determine how well current tools support GCC builtins, we manually implemented a test suite that tests the correct implementation of the 100 most commonly used architecture-independent builtins (cf. RQ2), which would support the architecture-independent portion of almost 90% of the builtin-using projects (see Section III-D). We omitted testing architecture-specific builtins due to the variability in architectures supported. For each builtin, we used its type information and documentation to determine both typical inputs and corner cases, then wrote test cases for them. As tools to be tested, we selected popular and widely used mature compilers, special-purpose compilers, source-to-source translators, alternative execution environments, and static analysis tools. Figure 5 shows the results.

Mature compilers. We tested the mature compilers GCC and Clang [5], the most widely used open-source compilers on Linux, and the commercial ICC. They all executed the test cases successfully, which demonstrates that mature tools should support them.

Special-purpose compilers. We tested the special-purpose compilers CompCert [6], [7] and TCC. CompCert is a compiler used in safety-critical applications and has been formally verified to be correct, which, however, excludes its implementation of builtins. We found that CompCert correctly executed only 9 builtin test cases, supporting 5 out of the 10 most frequently used builtins. Both `__builtin_clzl` and `__builtin_ctzl` computed an incorrect result for large input values.[5] After reporting the bugs detected by our test suite, they were fixed within a day with the note that "we need more testing here".

The TCC compiler is a small compiler developed to compile code quickly. It successfully ran only six builtin test cases. While most tests failed with a build error, the `__builtin_types_compatible_p` builtin produced an incorrect result when comparing enumerations.[6]

C front end. The C Intermediate Language (CIL) [37] is a front end for the C language that facilitates program analysis and transformation. We tested its driver, called cilly, which can also be used as a drop-in replacement for GCC. It successfully executed 40 builtin test cases. The `__builtin_bswap16` and `__builtin_types_compatible_p` builtins produced incorrect results.[7] Cilly also failed on 34 atomic test cases, on 15 test cases due to a failure to parse a system library, on 5 test cases due to unrecognized builtins, and on 4 test cases due to warnings for the `long double` type.

Source-to-source translators. We evaluated DragonEgg as a representative of source-to-source translators. DragonEgg compiles source languages supported by GCC to LLVM IR. Although it has not been updated for several years, it successfully executed more than two thirds of the test cases. It failed to translate more recent builtins (e.g., from the "atomic" category) that were added to GCC after the last commit in DragonEgg.

Static analysis. We tested Frama-C [38], [39], a static-analysis framework. By default, it assumes code to be portable, and supports compiler extensions only with an option. For 41 test cases, Frama-C's analysis did not trigger a warning or error [8]. 9 test cases failed because its standard library lacked macros for `INFINITY` and `NAN`, which were used in the test cases. 14 test cases for `__sync` builtins were generally supported, but incorrectly implemented for the long type. Furthermore, `__builtin_object_size` was implemented to always return -1—signifying that the size of an object could not be determined—but referred to an undefined variable in its macro, which resulted in an error.

Alternative execution environments. We tested Sulong [16], [17], an interpreter with dynamic compiler for LLVM-based languages, and KCC [40], [41], a commercial interpreter for C that was automatically derived from a formal semantics for C and detects Undefined Behavior. Sulong successfully executed all but two test cases, namely for `__builtin_fabsl` and `__builtin___clear_cache`, which were not implemented.[9] Note that we found these errors with a preliminary version of the test suite, and consequently contributed implementations for the two missing builtins.

KCC successfully executed test cases for 10 builtins, but, since it is based on CIL, it had the same error in the implementation of the `__builtin_types_compatible_p` builtin. The KCC developers also mentioned that they have "recently been trying to add more supports for gnuc builtins."[10]

Symbolic execution engine. We tested KLEE [15], a symbolic execution for LLVM-based languages. KLEE executed all test cases successfully when executed with concrete inputs.

[5]https://github.com/AbsInt/CompCert/issues/243

[6]http://lists.nongnu.org/archive/html/tinycc-devel/2018-07/msg00007.html

[7]https://github.com/cil-project/cil/issues/44

[8]https://lists.gforge.inria.fr/pipermail/frama-c-discuss/2018-July/005483.html

[9]https://github.com/graalvm/sulong/pull/807

[10]https://github.com/kframework/c-semantics/issues/318

IV. Threats to Validity

Internal Validity. The main threat to internal validity (i.e., whether we controlled for all confounding variables) is that we relied on a source-based heuristic approach to determine the usage of GCC builtins, namely by searching for identifiers of known builtins in the source files. We could have mistakenly recorded a builtin usage when the builtin was enclosed in a comment, or when an identifier with the same name as a builtin was used for another purpose. However, as described, we used several mitigation strategies to address such "deceiving" usages. Conversely, we could have missed builtin usages if their names consisted of strings that were concatenated by using preprocessor macros; however, we expect such usages to be uncommon.

External Validity. Several threats to external validity (i.e., whether our results are generalizable) are related to the scope of our analyses. First, besides C code, C++ code also can access GCC builtins, which we considered beyond our scope, so our results cannot be generalized to C++ projects. We analyzed open-source GitHub projects, hence our findings might not apply to proprietary projects. Furthermore, they do not necessarily apply to projects hosted on sites other than GitHub; this biases our results as, for example, GNU projects other than GCC are often hosted on Savannah and could potentially rely more strongly on GCC builtins. Additionally, our results cannot be generalized to the builtins of compilers other than GCC. Finally, we investigated the usage of builtins at the source level, which might be different from the usage in the compiled binary (e.g., because their usage could be influenced by macro metaprogramming) and the usage during execution of the program.

V. Related Work

Studies of inline assembly and linkers. Besides compiler builtins, C projects also contain other elements not specified by the C standard. Rigger et al. investigated the use of x86-64 inline assembly [2] and found that around 30% of popular C projects use it. In this paper, we demonstrated that GCC builtins are used more frequently than inline assembly, which provides even stronger incentives to implement support by C tools. Other studies focused on the role of linkers [1] and the preprocessor [42]. C projects are often built using Makefiles, whose feature usage has also been investigated [43].

Studies of other language features. This paper fits into a recent stream of empirical studies of programming language feature usage, all of which share a methodology of mining software repositories to determine the popularity of features in large sets of open-source projects and/or evaluate the "harmfulness" of features in terms of potential for bugs. Most of this work has focused on general-purpose programming languages, and research has evolved from more common to lesser known features. For example, for Java the usage of general language features [44], [45], fields [46], inheritance [47], exception handling [48], [49], [50], lambda features [51] and async constructs on Android [52] have been studied. For C++ projects, the usage of templates [53], generic constructs [54], concurrency constructs [55] and asserts [56] have been studied. The latter also considered C projects, similar to Nagappan et al.'s study [57] of the usage and harmfulness of the goto construct.

However, to the best of our knowledge, a study of the usage of compiler builtins has not yet been conducted, and as such fits into the line of research into C programming language features. Analysis of GCC builtins warrants analysis, since developers of tools for C need to deal with them and since they affect the maintenance cost of projects due to potential for vendor lock-in. Furthermore, builtins are not documented well, as we demonstrated in Section III-A.

VI. Conclusions

We have presented an empirical study of the usage of GCC builtins in a corpus of 4,912 open-source C projects retrieved from GitHub. We found that 12,126 GCC builtins exist that tool developers potentially need to consider, but that only about 3,000 of these are used. Although a builtin is typically found only once every 5,737 lines of C code, 38% of all popular projects rely on compiler builtins, thus strongly incentivizing their implementation in analysis and other tools.

The use of GCC builtins was dominated by architecture-independent builtins for direct interaction with the compiler, for bit-and-byte operations, atomic operations, and libc equivalents. Depending on the tool, different builtin categories could be supported to different degrees; for example, static analysis tools that do not analyze the semantics of multithreaded atomic operations might eschew implementing the atomic builtins or implement them assuming only one thread. Architecture-specific builtins were used by fewer projects, but in greater number than architecture-independent builtins; they were used for SIMD instructions, to determine CPU features, and to access platform-specific registers.

Mature compilers such as GCC, Clang, and ICC support the most common builtins. However, in other popular tools, such as CompCert, TCC, KCC, and Frama-C, we found missing and erroneous implementations. We speculate that builtin support could be improved also in less commonly used tools. Tools based on existing mature compiler infrastructure—such as KLEE and Sulong, which are based on LLVM—seem to have a better builtin support, partly because some builtins are handled in the compiler's front end.

We have shown that it is sufficient for compilers and other tools to support only a small number of builtins to be able to handle a large percentage of all C projects. We suggest that tool developers use a greedy approach: by implementing 30 builtins, half of the projects are supported, about 600 builtins are needed to support 90% of all projects, and about 1,600 builtins are required to support 99% of projects. When considering only architecture-independent builtins, only about 250 builtins need to be implemented.

We analyzed the development history of builtins in projects and found that more projects (143) added them than removed them (97) over time. Calls to builtins were added for performance optimizations, atomic implementations, to enable metaprogramming techniques, and others; they were removed, for example, due to their maintenance cost and through refactorings. Overall, it seems that compiler builtins are not a legacy feature from times when compilers applied less sophisticated optimizations; tool developers must expect that contemporary and future code will use them.

References

[1] S. Kell, D. P. Mulligan, and P. Sewell, "The missing link: Explaining elf static linking, semantically," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: ACM, 2016, pp. 607–623.

[2] M. Rigger, S. Marr, S. Kell, D. Leopoldseder, and H. Mössenböck, "An Analysis of x86-64 Inline Assembly in C Programs," in *Virtual Execution Environments*, ser. VEE 2018.

[3] Clang Team, "Clang language extensions. builtin functions," 2018. [Online]. Available: https://clang.llvm.org/docs/LanguageExtensions.html

[4] J. O'Neill, "Intel compilers for linux: Compatibility with gnu compilers," 2006.

[5] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–.

[6] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009.

[7] ——, "A formally verified compiler back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 363–446, 2009.

[8] Z. Xu, T. Kremenek, and J. Zhang, "A memory model for static analysis of c programs," in *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I*, ser. ISoLA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 535–548.

[9] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Softw.*, vol. 19, no. 1, pp. 42–51, Jan. 2002.

[10] D. Evans, J. Guttag, J. Horning, and Y. M. Tan, "Lclint: A tool for using specifications to check code," pp. 87–96, 1994.

[11] D. Xu, "[frama-c-discuss] inline assembly code," 2011, (Accessed October 2017). [Online]. Available: https://lists.gforge.inria.fr/pipermail/frama-c-discuss/2011-March/002589.html

[12] G. J. Holzmann, "Uno: Static source code checking for user-defined properties," in *Proc. IDPT*, vol. 2, 2002.

[13] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell, "Into the depths of c: Elaborating the de facto standards," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: ACM, 2016, pp. 1–15.

[14] R. Krebbers and F. Wiedijk, "A typed c11 semantics for interactive theorem proving," in *Proceedings of the 2015 Conference on Certified Programs and Proofs*, ser. CPP '15. New York, NY, USA: ACM, 2015, pp. 15–27.

[15] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[16] M. Rigger, M. Grimmer, C. Wimmer, T. Würthinger, and H. Mössenböck, "Bringing low-level languages to the jvm: Efficient execution of llvm ir on truffle," in *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL 2016. New York, NY, USA: ACM, 2016, pp. 6–15.

[17] M. Rigger, R. Schatz, R. Mayrhofer, M. Grimmer, and H. Mössenböck, "Sulong, and Thanks For All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2018.

[18] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*. IEEE, 2015, pp. 46–55.

[19] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*, 2012, pp. 309–318.

[20] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 245–258.

[21] ——, "Cets: Compiler enforced temporal safety for c," *Proceedings of the International Symposium on Memory Management*, vol. 45, no. 8, pp. 31–40, Jun. 2010.

[22] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294.

[23] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: ACM, 2016, pp. 849–863.

[24] ——, "Finding and analyzing compiler warning defects," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 203–213.

[25] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang, "Testing static analyzers with randomly generated programs," in *Proceedings of the 4th International Conference on NASA Formal Methods*, ser. NFM'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 120–125.

[26] T. Kapus and C. Cadar, "Automatic testing of symbolic execution engines via program generation and differential testing," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 590–600.

[27] H. Borges, A. C. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of github repositories," in *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, 2016, pp. 334–344.

[28] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 92–101.

[29] D. Quinlan, "Rose: Compiler support for object-oriented frameworks," in *Proceedings of Conference on Parallel Compilers (CPC2000), Aussois, France*, ser. Parallel Processing Letters, vol. 10. Springer Verlag, 2000.

[30] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: A map of code duplicates on github," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 84:1–84:28, Oct. 2017.

[31] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in c/c++," pp. 760–770, 2012.

[32] M. Rigger, D. Pekarek, and H. Mössenböck, "Context-aware failure-oblivious computing as a means of preventing buffer overflows," *arXiv preprint arXiv:1806.09026*, 2018.

[33] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches," *CoRR*, vol. abs/1702.00719, 2017.

[34] M. Rigger, R. Schatz, R. Mayrhofer, M. Grimmer, and H. Mössenböck, "Introspection for C and its Applications to Library Robustness," *The Art, Science, and Engineering of Programming*, no. 2, 2018.

[35] Intel, "Cilk plus." [Online]. Available: https://www.cilkplus.org/

[36] W. Cunningham, "The wycash portfolio management system," vol. 4, pp. 29–30, 04 1993.

[37] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02. London, UK, UK: Springer-Verlag, 2002, pp. 213–228.

[38] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c," in *Software Engineering and Formal Methods*, G. Eleftherakis, M. Hinchey, and M. Holcombe, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 233–247.

[39] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Aspects of Computing*, vol. 27, no. 3, pp. 573–609, May 2015.

[40] C. Ellison and G. Rosu, “An executable formal semantics of c with applications,” in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’12. New York, NY, USA: ACM, 2012, pp. 533–544.

[41] C. Hathhorn, C. Ellison, and G. Roşu, “Defining the undefinedness of c,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’15. New York, NY, USA: ACM, 2015, pp. 336–345.

[42] M. D. Ernst, G. J. Badros, and D. Notkin, “An empirical analysis of c preprocessor use,” *IEEE Trans. Softw. Eng.*, vol. 28, no. 12, pp. 1146–1170, Dec. 2002.

[43] D. H. Martin, J. R. Cordy, B. Adams, and G. Antoniol, “Make it simple: An empirical analysis of gnu make feature use in open source projects,” in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ser. ICPC ’15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 207–217.

[44] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, “Mining billions of AST nodes to study actual and potential usage of Java language features,” in *36th International Conference on Software Engineering*, ser. ICSE’14, June 2014, pp. 779–790.

[45] D. Qiu, B. Li, E. T. Barr, and Z. Su, “Understanding the syntactic rule usage in java,” *Journal of Systems and Software*, vol. 123, pp. 160–172, 2017.

[46] E. D. Tempero, “How fields are used in java: An empirical study,” in *20th Australian Software Engineering Conference (ASWEC 2009), 14-17 April 2009, Gold Cost, Australia*, 2009, pp. 91–100.

[47] E. Tempero, J. Noble, and H. Melton, “How do java programs use inheritance? an empirical study of inheritance in java software,” in *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ser. ECOOP ’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 667–691.

[48] M. Asaduzzaman, M. Ahasanuzzaman, C. K. Roy, and K. A. Schneider, “How developers use exception handling in java?” in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR ’16. New York, NY, USA: ACM, 2016, pp. 516–519.

[49] S. Nakshatri, M. Hegde, and S. Thandra, “Analysis of exception handling patterns in java projects: An empirical study,” in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR ’16. New York, NY, USA: ACM, 2016, pp. 500–503.

[50] D. Sena, R. Coelho, U. Kulesza, and R. Bonifácio, “Understanding the exception handling strategies of java libraries: An empirical study,” in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR ’16. New York, NY, USA: ACM, 2016, pp. 212–222.

[51] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig, “Understanding the use of lambda expressions in java,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 85:1–85:31, Oct. 2017.

[52] S. Okur, D. Dig, and Y. Lin, “Study and refactoring of android asynchronous programming,” 2015.

[53] D. Wu, L. Chen, Y. Zhou, and B. Xu, “An empirical study on the adoption of C++ templates: Library templates versus user defined templates,” in *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013.*, 2014, pp. 144–149.

[54] A. M. Sutton, R. Holeman, and J. I. Maletic, “Identification of idiom usage in C++ generic libraries,” in *The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010*, 2010, pp. 160–169.

[55] D. Wu, L. Chen, Y. Zhou, and B. Xu, “An extensive empirical study on C++ concurrency constructs,” *Information & Software Technology*, vol. 76, pp. 1–18, 2016.

[56] C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, and B. Ray, “Assert use in github projects,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE ’15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 755–766.

[57] M. Nagappan, R. Robbes, Y. Kamei, E. Tanter, S. McIntosh, A. Mockus, and A. E. Hassan, “An empirical study of goto in c code from github repositories,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 404–414.

Part III

Future Work and Conclusion

Chapter 11

Future Work

This chapter discusses potential future research directions and ongoing work based on the results of this thesis. The topics are categorized according to the thesis' contributions into those related to Safe Sulong, to introspection, and to empirical studies of unstandardized elements in C/C++.

11.1 Safe Sulong

The evaluation of Safe Sulong could be extended to incorporate larger programs, by running a complete libc on it. To improve efficiency, Safe Sulong's warm-up costs could be reduced. Furthermore, it could be enhanced to detect further kinds of undefined behavior and an execution mode could be added that tracks the flow of undefined behavior through the program. Since Safe Sulong cannot call binary code contained in native libraries, future research could explore possibilities to interact with such code. Finally, Safe Sulong could be paired with a fuzzer to cover more paths in the tested program, to have a higher chance of finding bugs.

Extending the evaluation The evaluation of Sulong could be extended into various directions. In our evaluation, we tested Safe Sulong only on small to medium-sized benchmarks and programs. While the performance of previous Truffle implementations scaled also with their completeness [148], evidence is required to determine if Safe Sulong executes as efficiently on larger benchmarks and programs. Safe Sulong could also be evaluated on other unsafe languages supported by LLVM, such as Fortran (which we evaluated on Native Sulong [103]) and C++. When evaluating Safe Sulong's performance on C++, it would be interesting to determine how speculative function pointer inlining (i.e., inlining of virtual calls) influences performance, which, to the best of our knowledge, has not yet been researched for C++. Finally, Safe Sulong could be evaluated as a native function interface used by Java programs or by other Truffle language implementations; currently, Native Sulong is already used for the implementation of the native function interface in TruffleRuby and GraalPython.

Improving warm-up performance One challenge for executing some larger programs are long warm-up times. Sulong does not yet support On-Stack Replacement (OSR) [41, 3], a technique that allows transferring control to a compiled version of a loop while the loop is running in the interpreter, to reduce the time being spent in its unoptimized version. In fact, Sulong interprets some of the loops in the SPEC benchmarks for a long time before they are compiled and their machine code is executed. As part of future work, OSR is currently being implemented in Sulong to support the efficient execution of such benchmarks.

Benchmarks for evaluating dynamic compilation systems for C/C++ New benchmark suites for evaluating dynamic compilation for C/C++ could be developed. Since the peak performance of dynamically compiled code is evaluated using a large number of runs to account for warm-up time and nondeterminism [8], the SPEC benchmarks, where a single benchmark execution can take hours, makes evaluations time-consuming. Shorter-running benchmarks, which could then be executed a larger number of times, would be better suited for evaluating such systems.

Executing a complete libc Missing features, such as unimplemented library functions, constitute a problem for executing larger benchmarks. We currently use a custom libc that relies on functions similar to system calls, which are implemented as Java methods. However, this library only provides the most commonly used library functions, and lacks functions that are difficult to implement in Java (e.g., `mmap` for mapping memory). Recently, we have implemented the Linux syscall interface in Java, which allows Native

Sulong to execute the musl libc.[1] As part of future work, Safe Sulong could be extended to support the execution of such a libc implementation.

Detecting further occurrences of undefined behavior Currently, Safe Sulong can detect various memory errors and illegal inputs for arithmetic operations. It could be extended to add checks for other instances of undefined behavior. For example, libc functions could be enhanced to detect input that is considered illegal by the C standard. Additionally, Safe Sulong could be enhanced to detect memory leaks, for example, by adding a field to all dynamically-allocated objects that indicates whether an object has been freed; when an object is eventually collected by the garbage collector, a Java finalizer could check whether it has actually been freed.

Tracking the flow of undefined behavior Safe Sulong's *Lenient C* defines semantics for undefined behavior in C in order to predictably execute programs with such undefined behavior. However, since undefined behavior constitutes an error, it would still be desirable to determine whether values affected by undefined behavior propagate to safety-critical or security-critical parts of an application. To achieve this, operations could, when causing undefined behavior, produce a special undefined behavior value that could be tracked through the program (known as a taint value). Every operation that depends on this value would then be marked as unsafe, which could be visualized for the programmer. Furthermore, security-critical operations (e.g., memory allocations) could reject tainted values to lower the probability of a critical error, as has been proposed for integer overflows [138].

Calling native code Safe Sulong currently does not support calling functions contained in binaries, which is a problem when the source code of a library is no longer available. Using a native function interface like the Java Native Interface[2] would cause significant overhead, as Java objects passed to the native side would need to be moved to unmanaged memory. To tackle this, we have been working on an x86 Truffle interpreter that can be used as a native function interface by Sulong. By using this interpreter, Sulong does not need to move Java objects to unmanaged memory, because the

[1]`https://www.musl-libc.org/`

[2]`https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html`

x86 interpreter is implemented in Java itself and also uses Java objects to represent user objects. Furthermore, using a Java interpreter for x86 code prevents memory errors from crashing or corrupting the virtual machine. Instead, such errors result in Java exceptions that can be handled appropriately. A similar approach has also been implemented on top of the RPython tool chain [110, 2], albeit it was proposed for simulating processor architectures [84]. Alternatively, binary code could be "lifted" to LLVM IR, which Safe Sulong could then execute.

Implementing a hybrid Sulong system A hybrid version that combines the advantages of Native Sulong and Safe Sulong could be implemented. Native Sulong provides seamless native interoperability, while Safe Sulong provides memory safety. These approaches could be combined by allocating either a Java object or unmanaged memory for a user allocation, depending on the context. The decision could be based on heuristics, run-time feedback, or static analysis. For example, if an object is possibly passed to a native library, it would be beneficial for the run-time performance to allocate this object as an unmanaged object to avoid conversion. Note that such a hybrid version would no longer be completely safe, as objects allocated in unmanaged memory would be accessed without bounds checks.

Implementing a Sulong-specific fuzzer Safe Sulong detects only those errors that are triggered during the execution of a program. Thus, its bug-finding capabilities depend on the coverage and quality of test cases. A technique to improve the coverage is fuzzing, where sample input to the program is mutated randomly. There are three different categories of fuzzers that make no assumptions about the structure of inputs: blackbox, whitebox, and greybox fuzzers. Blackbox fuzzers [89] are the least-sophisticated kind, as they generate and mutate the input without considering the paths triggered in the program. Whitebox fuzzers [45] are based on symbolic execution, to collect path constraints, negate them, and pass them to a SAT solver to systematically generate new test cases. Greybox fuzzers [14] combine both approaches; while they incorporate program feedback like whitebox fuzzers, they are almost as fast as blackbox fuzzers since they only use lightweight instrumentation. The lightweight instrumentation consists of counters that are inserted to determine which paths are executed. While any kind of fuzzer could be implemented in Sulong, we believe that a greybox fuzzer would be most suitable, since it could benefit from dynamic compilation. For example, it could use dynamic optimizations to selectively enable instrumentation for paths that are seldom executed and disable it for paths that are often executed. This would likely improve the performance of fuzzing, while still being able to guide the fuzzer to generate “interesting” inputs. Challenges for implementing fuzzing in Safe Sulong (and other Truffle languages) would be to find out how to efficiently reset global state after each run to achieve a high fuzzing throughput.

11.2 Introspection

We believe that introspection could be used to increase the effectiveness of fuzzers, and also to implement special-purpose applications that could not be easily implemented without introspection.

Introspection to assist fuzzing Fuzzers generate random program input that can trigger errors in applications by causing them to crash or time out. Frequently, they are also used together with dynamic bug-finding tools that detect illegal operations. As part of our work on introspection, we have demonstrated that introspection functions are applicable to being used in assertions to, for example, verify bounds. A hypothesis is that the use of such introspection-based assertions could be effective to expose bugs when used together with fuzzers. This could be tested in a case study, in which such assertions would be added to real-world programs, which would then be fuzzed.

Introspection for special-purpose applications Being able to query object metadata allows the implementation of various applications that are difficult (or impossible) to write in C. For example, serializing objects in C requires programmers to write a custom serialization logic that requires knowledge about the structure of each object. Using introspection, the size and types of each object could be determined, which would allow for an automatic serialization mechanism. We believe that also other introspection-based special-purpose applications could be implemented and evaluated.

11.3 Empirical Studies

In terms of empirical studies, the investigation on the use of inline assembly and GCC builtins could be extended for better generalizability, and also the use of unstandardized elements (e.g., compiler flags) could be studied. Furthermore, the findings could be used to improve the support in tools.

Extensions of the inline assembly and builtin studies We analyzed the usage of x64-86 inline assembly and GCC builtins in open-source GitHub C projects. It is unclear whether the results are generalizable to other languages such as C++, to closed-source projects, to projects hosted on other sites, and to other architectures and compiler builtins. Future work could investigate this. Furthermore, we analyzed inline assembly and GCC builtins on the source level; analyzing them in the resulting binary (or during compilation) would be a better indicator of how often they are used, since they can be excluded by preprocessor macros and by similar mechanisms. However, note that the results would only apply to specific versions of evaluated compilers, since preprocessor macros that decide on whether to include builtins or inline assembly often check for specific compilers and compiler versions. Finally, it could be researched whether the instructions resulting from the compiler builtins are executed during typical program runs.

Other unstandardized elements in C projects Besides GCC builtins and inline assembly, also other unstandardized elements in C/C++ projects exist (e.g., function attributes, variable attributes, and compiler pragmas). These elements could be researched to determine their maintenance cost and the cost needed to support them in tools. For Lenient C, we assigned semantics for otherwise undefined operations ad-hoc while executing non-portable programs by Safe Sulong, and based on the results of related work. As part of future work, it could be researched whether these assumptions hold for real-world programs. One way to test this hypothesis would be to analyze the use of compiler flags in practice. GCC provides, for example, the `-fwrapv` flag to define signed integer overflow to wrap around. If this and other flags are commonly used, it would support the decision to make this the default behavior in Lenient C, or even in an upcoming C standard.

Testing the support of unstandardized elements In our studies, we have shown that many tools lack support for unstandardized elements in C projects or that they implement them incorrectly. To improve the support of such elements, tools could be systematically tested. For example, random program generators such as Csmith [126] could be extended to generate unstandardized elements.

Chapter 12

Conclusion

Programs written in unsafe languages like C/C++ can cause undefined behavior, which poses a problem for the programs' correct execution and the systems' security. To tackle undefined behavior, this thesis has contributed in three areas.

Safe Sulong First, we have proposed a safe execution system that has an interpreter written in a safe programming language at its core. We implemented this approach as a tool, called Safe Sulong. Our evaluation demonstrated that Safe Sulong detects bugs in corner cases that other tools overlook (e.g., because instrumentation was omitted for corner cases); Safe Sulong's advantage is that it is based on automatic run-time checks of the underlying virtual machine and that it employs an optimizer that does not exploit undefined behavior. In terms of peak performance, the results demonstrated that programs executed by Safe Sulong, which uses a dynamic compiler, execute similarly efficient as executables generated by static compilers. However, Safe Sulong is still a research prototype and it yet needs to be extended in terms of completeness to evaluate its behavior on large programs.

Introspection Second, we have proposed an approach to provide metadata tracked by existing dynamic bug-finding tools to the programmer, who can use this metadata to improve a library's robustness. We have shown that introspection is applicable to various tools, as we implemented introspection functions in Safe Sulong, LLVM's AddressSanitizer [112], SoftBound [94], and GCC's Intel MPX-based bounds instrumentation [99]. Furthermore, we implemented a subset of libc, which we enhanced by introspection-based functionality and studied how introspection could improve its robustness. Based on these results, we proposed context-aware failure-oblivious computing as an introspection-based technique to continue program execution in the presence of a buffer overflow. We evaluated this approach in a case study of real-world bugs in popular applications, and in terms of performance. The results indicate that context-aware failure-oblivious computing is effective to continue normal program behavior after mitigating an error. For tools where introspection access could be implemented efficiently, the performance overhead of introspection was low. Thus, we believe that context-aware failure-oblivious computing could be used in a real-world scenario.

Empirical studies Third, we have analyzed the use of C language extensions, namely inline assembly and GCC builtins, in C projects. We found that these extensions are commonly used, which provides high incentives to tool authors to support them in existing bug-finding tools. However, due to the high implementation effort, fully implementing them is sometimes infeasible, as, for example, over 10,000 GCC builtins exist. Our findings suggest that by implementing only a small subset of GCC builtins and inline assembly, already a large number of projects could be supported, as most projects use a specific subset of these elements. Our analysis on the development of GCC builtins over time suggests that they are not legacy features, so that tools are likely required to support them also in the future.

Impact The work done as part of this thesis has already had an impact on industry and research. Sulong was productized as a part of GraalVM, which is a multi-lingual virtual machine that is maintained by Oracle Labs.[1] In GraalVM, Native Sulong has been used to implement native extensions in Truffle-based language implementations, for example, for the Ruby[2] and Python[3] Truffle implementations. Sulong has been used for other research projects within Oracle, for example, Iraklis et al. used it to implement a smart array data structure [101]. From the Johannes Kepler University

[1]`http://www.graalvm.org/`
[2]`https://github.com/oracle/truffleruby`
[3]`https://github.com/graalvm/graalpython`

Linz, for example, Kreindl et al. used Sulong as a platform to implement a mechanism for debugging native extensions [78] and Mosaner et al. used it to demonstrate the implementation of an On-stack Replacement mechanism for Truffle implementations of unstructured languages [90]. Gaikwad et al. from the University of Manchester evaluated their approach on visualizing the performance of Truffle-based languages also by using Sulong [43]. While evaluating Safe Sulong, we found and reported bugs and missing features, both in projects and in other bug-finding tools; we also implemented missing features in other tools, for example, in LLVM's AddressSanitizer.[4] We found incorrectly implemented builtins in our GCC builtin study, which, for example, resulted in a bug fix in the formally-verified CompCert compiler.[5] Native Sulong, the data sets and scripts used in our empirical studies, and the implementation for context-aware failure-oblivious computing are available online to facilitate further research.

[4]`https://github.com/google/sanitizers/issues/766`
[5]`https://github.com/AbsInt/CompCert/issues/243`