Sichere Ausführung von LLVM-basierten Sprachen auf der Java Virtual Machine[1]

Manuel Rigger[2]

Abstract: Sprachen wie C/C++ sind unsicher, da gewisse ihrer Operationen zu *undefiniertem Verhalten* führen können. Undefiniertes Verhalten ist problematisch, da es oft von AngreiferInnen ausgenutzt wird und zu schwer auffindbaren Fehlern führen kann. Um dieses Problem zu lösen haben wir Safe Sulong entwickelt, ein Werkzeug, das es erlaubt, unsichere Sprachen in einem sicheren Modus auszuführen. Die Grundidee dabei ist, die unsichere Sprache in einem Interpreter auszuführen, der in einer sicheren Sprache geschrieben ist, und damit automatisch undefiniertes Verhalten zu eliminieren. Die Evaluierung von Safe Sulong zeigt, dass es Fehler findet, die von anderen Werkzeugen übersehen werden, während Sulongs Ausführungsgeschwindigkeit vergleichbar mit der anderer Werkzeuge ist. Um die Implementierung von Inline-Assembly und Compiler-Builtins in Safe Sulong zu unterstützen, haben wir ferner empirische Studien durchgeführt, in denen wir diese Elemente untersuchten. Außerdem haben wir Introspektionsfunktionen entwickelt, mit denen man Metadaten von Fehlerfindungstools abfragen kann.

1 Problemkontext

In unsicheren Sprachen wie C ist die Semantik von Operationen nur für gültige Eingabewerte spezifiziert. Während z.B. das Dereferenzieren eines gültigen Zeigers spezifiziert ist, ist das Dereferenzieren eines Zeigers, der außerhalb eines Objektes zeigt (was als *Pufferüberlauf* bezeichnet wird), nicht spezifiziert. Solch ein Fehler verursacht *undefiniertes Verhalten*. Compiler sind nicht verpflichtet, Maschinencode zu produzieren der undefiniertes Verhalten erkennt, daher fügen sie z.B. keine Überprüfungen ein, die Pufferüberläufe verhindern. In der Tat optimieren weit verbreitete Compiler wie Clang or GCC Programme unter der Annahme, dass undefiniertes Verhalten nie auftritt [Wa12].

Das Ausführen von Programmen mit undefiniertem Verhalten, die mit Clang or GCC kompiliert wurden, kann zu verschiedenen unbeabsichtigten oder sogar katastrophalen Ergebnissen führen. Zum Beispiel kann das Lesen außerhalb von Objekten sensitive Daten preisgeben, die zu anderen Objekten gehören, auch wenn diese nicht explizit referenziert wurden. Aktuelle Beispiele für solche Fehler sind *Heartbleed* in der OpenSSL Bibliothek und *Cloudbleed* im Cloudfare Online-Service, die es beide ermöglichten sensitive Daten auf Webservern zu entwenden, was potenziell Millionen von Nutzern betraf. Pufferüberläufe können aber nicht nur dazu ausgenutzt werden, um private Daten zu lesen; Schreibzugriffe außerhalb von Objekten können benutzt werden um Kontrollflussdaten (z.B. Funktionsadressen) zu überschreiben, was AngreiferInnen ausnutzen können, um die Kontrolle über den Prozess zu erlangen.

Undefiniertes Verhalten ist auch dann problematisch, wenn es nicht von AngreifernInnen ausgenutzt wird. Programmfragmente, die undefiniertes Verhalten auslösen, können zu Maschinencode kompiliert werden, der sich bei der Ausführung anders verhält als erwartet. Solche Fragmente können auch komplett vom Compiler entfernt (d.h. wegoptimiert) werden, was zu „fehlerhaft kompiliertem" Code führt, der schwer zu debuggen ist. Manchmal ist undefiniertes Verhalten nicht unmittelbar ein Problem, z.B. wenn eine Operation so kompiliert wird, dass sie zufällig das erwartete Verhalten aufweist. Zum Beispiel kann eine Ganzzahladdition zu einer x86-`add`-Instruktion kompiliert werden,

[1] Englischer Titel der Dissertation: „Safe Execution of LLVM-based Languages on the Java Virtual Machine"

[2] Johannes Kepler Universität Linz, `manuel.rigger@jku.at`

die bei einem arithmetischen Überlauf das Vorzeichen wechselt). Solche Fehler sind jedoch „tickende Zeitbomben“, da zukünftige Compiler undefiniertes Verhalten ausnutzen könnten, um stärker zu optimieren.

2 Stand der Technik

Sowohl die Industrie als auch die Wissenschaft haben undefiniertes Verhalten, insbesondere Pufferüberläufe, seit Jahrzehnten behandelt. Daher gibt es eine Fülle von Ansätzen, die es auf unterschiedliche Weise angehen. Ein wichtiger Ansatz sind dynamische Fehlersuchwerkzeuge (sogenannte *Sanitizer*), die bestimmte Klassen von undefiniertem Verhalten erkennen, indem sie das Programm instrumentieren und die Ausführung bei einem Fehler abbrechen. Diese Werkzeuge erkennen Fehler während der Ausführung des Programms und müssen daher mit konkreten Programmeingaben (z. B. Programmargumenten oder Kommandozeileneingaben) versorgt werden. Sie erkennen Fehler in den Programmen ohne Fehlalarme, d.h. Fehler, die von solchen Werkzeugen angezeigt werden, sind immer echte Fehler.

Sanitizer können entweder mittels dynamischer Binärcode-Instrumentierung oder mittels Instrumentierung zur Compilezeit entwickelt werden. Dynamische Binärcode-Instrumentierung fügt Prüfungen zum Maschinencode hinzu, nachdem das Programm gestartet wurde. Ein Beispiel dafür ist Valgrind [NS07], das z.B. Speicherfehler im Programm erkennt. Ein großer Vorteil dieses Ansatzes ist, dass Werkzeuge keinen Zugriff auf den Quellcode benötigen, da sie direkt auf dem Binärcode arbeiten. Ein Nachteil ist jedoch, dass beim Übersetzungsvorgang Informationen verloren gehen, so dass nur bestimmte Fälle von undefiniertem Verhalten erkannt werden. Da es unser Ziel ist, möglichst viele Fehler zu finden, konzentrieren wir uns auf die Instrumentierung zur Compilezeit, die Prüfungen in den Quellcode oder in die Zwischensprache des Compilers einfügen und deshalb typischerweise alle Fehler unterschiedlicher Fehlerkategorien während der Ausführung erkennen können. Ein Beispiel für diesen Ansatz ist LLVM’s AddressSanitizer (ASan), der Pufferüberläufe und andere Fehler erkennt [Se12].

3 Herausforderungen

Unser Hauptziel war es, vorhandene Sanitizer zu verbessern und Programme sicher auszuführen, auch wenn sie undefiniertes Verhalten verursachen. *Sicher* bedeutet in diesem Kontext, dass Operationen mit undefiniertem Verhalten eine definierte Semantik zugeordnet wird; z.B. soll eine undefinierte Operationen zum kontrollierten Beenden des Programms mit einer Fehlermeldung führen. Wir haben drei Bereiche identifiziert, in denen aktuelle Ansätze Schwächen aufweisen, die wir in Angriff nehmen wollten.

Unsichere Optimierungen. Wenn Compileroptimierungen aktiviert sind, können Sanitizer nicht alle Fehler erkennen, da die Compiler undefiniertes Verhalten für Optimierungen nutzen; wenn Optimierungen jedoch deaktiviert sind, ist das kompilierte Programm langsam. Außerdem ist das Hinzufügen von Laufzeitprüfungen sowohl im Compiler als auch im Binärcode fehleranfällig, da die Instrumentierung für Sonderfälle vergessen werden kann, was die Fehlererkennung beeinträchtigt. Wir glauben daher, dass ein neuartiger Ansatz erforderlich ist, der umfassenden Schutz vor undefiniertem Verhalten bietet, indem undefiniertes Verhalten nicht wegoptimiert werden kann, das entstehende Programm aber dennoch schnell genug ist, um in der Praxis eingesetzt zu werden.

Fehlendes Wissen über Inline-Assembly und Compiler-Builtins. Werkzeuge für unsichere Sprachen wie C erfordern einen hohen Implementierungsaufwand. C-Programme enthalten beispielsweise unstandardisierte Elemente wie Inline-Assembly und Compiler-Builtins. Nach bestem Wissen und Gewissen erkennen aktuelle Sanitizer undefiniertes Verhalten in ihnen nicht. Eine vollständige Implementierung solcher Elemente würde die Unterstützung verschiedener

Maschinenarchitekturen mit Hunderten oder Tausenden verschiedener Instruktionen oder Builtins erfordern. Es wäre hilfreich, ihre Verwendung in der Praxis zu untersuchen, da Werkzeugentwickler dadurch die Implementierung von Fehlerprüfungen für solche Elemente priorisieren können.

Fehlende Programmierschnittstellen. Sanitizer zeichnen Metadaten auf (z.B. Objektgrößen) und verwalten diese, um Fehlerprüfungen zu implementieren. Sie stellen diese Informationen jedoch nicht ProgrammiererInnen zur Verfügung, die diese Information beispielsweise verwenden könnten, um Invarianten im Programm zu überprüfen. Wir glauben, dass es hilfreich wäre, solche Metadaten über eine Programmierschnittstelle zur Verfügung zu stellen, die von verschiedenen Werkzeugen implementiert werden könnte. Das würde ProgrammierInnen helfen, undefiniertes Verhalten manuell zu verhindern.

4 Wissenschaftliche und Technische Beiträge

Die vorliegende Arbeit weist drei Beiträge zur Lösung der oben beschriebenen Probleme auf: *Safe Sulong*, *Introspektion* und *Empirische Studien*.

Safe Sulong. Wir stellen einen neuartigen Ansatz vor, um potenziell unsichere LLVM-basierte Sprachen sicher auszuführen, indem sie auf einer virtuellen Maschine für Java ausgeführt werden. Wir haben diese Idee in einem Werkzeug namens *Safe Sulong* umgesetzt. LLVM ist ein Compiler-Framework, das die Übersetzung vieler unsicherer Eingabesprachen in eine Zwischendarstellung ermöglicht, die wir ausführen. Indem wir uns auf den Just-in-Time-Compiler der Java Virtual Machine (JVM) verlassen, der für eine sichere Sprache entwickelt wurde, können wir unsichere Sprachen sicher optimieren (d.h. ohne undefiniertes Verhalten in unseren Optimierungen ausnutzen), während die spezifizierte Semantik der unsicheren Sprachen eingehalten wird. Wir haben zwei Arten entwickelt, wie undefiniertes Verhalten in diesem Ansatz behandelt wird. Wir stellen einen Sanitizer-Modus vor, der ungültige Speicherzugriffe und andere Fehler erkennt. Wir haben ihn in Hinblick auf seine Wirksamkeit beim Finden von Fehlern in Open-Source Projekten und in Bezug auf seine Ausführungsgeschwindigkeit evaluiert. Außerdem haben wir einen *Lenient C* Modus entwickelt, der undefiniertes Verhalten in C so spezifiziert, wie es häufig von ProgrammierInnen erwartet wird.

Empirische Studien. Wir präsentieren zwei empirische Studien zur Verwendung unstandardisierter Elemente in C-Projekten, um WerkzeugentwicklerInnen zu helfen, diese Elemente bei der Implementierung von Tools für C-Code zu priorisieren. In diesen Studien haben wir die Verwendung von x86-64 Inline-Assembly und die Verwendung von GCC-Builtins in einer großen Anzahl von Open-Source-Projekten analysiert. Nach unserem besten Wissen sind dies die ersten Studien zur Verwendung von Inline-Assembly und Compiler-Builtins.

Introspektion. Wir stellen einen neuen Ansatz vor, mit dem ProgrammiererInnen die Robustheit ihrer C-Bibliotheken erhöhen können. Dieser Ansatz basiert auf einer Introspektions-Schnittstelle, die ProgrammiererInnen das Abfragen von Metadaten wie z.B. Objektgrößen und -typen ermöglicht. Wir haben diesen Ansatz in einer Fallstudie einer *libc*-Implementierung für Safe Sulong evaluiert. Außerdem haben wir einen Ausführungsgmodus entwickelt, der Fehler abwehren kann ohne die Ausführung zu stoppen (namens *Context-aware Failure-oblivious Computing*). Wir haben gezeigt, dass Introspektion nicht nur in Safe Sulong, sondern auch in anderen Sanitizern wie ASan [Se12], MPX-basiertes Prüfen von Objektgrenzen und SoftBound+CETS [RPM18] implementiert werden kann. Wir haben sowohl die Geschwindigkeit des Ansatzes, als auch die Wirksamkeit in einer Studie von Fehlern in weit verbreiteten Programmen evaluiert.

Publikationen, Auszeichnungen und Vorträge. Im Zuge der Dissertation haben wir einen Journal-, fünf Konferenz-, zwei Workshopartikel und fünf weitere begutachtete Beiträge publiziert. Desweiteren befinden sich zur Zeit zwei Manuskripte unter Begutachtung. Den wichtigsten Beitrag der Dissertation, Safe Sulong, haben wir in *ASPLOS '18* publiziert, einer Konferenz, die eine Akzeptanzrate von 18% hatte. Der Autor gewann die *ACM Student Research Competition* auf der *Programming '18* Konferenz, wurde zu Vorträgen an der *University of Cambridge*, am *Imperial College London* und an der *Universität Salzburg* eingeladen und hielt Vorträge an diversen Entwicklerkonferenzen.

Auswirkungen auf die Industrie. Sulong wurde in die *GraalVM* integriert,[1], eine mehrsprachige virtuelle Maschine, die von Oracle entwickelt wurde. Hierbei wird Sulong zum Ausführen von LLVM-basierten Sprachen verwendet und wird aktuell von mehreren Oracle-Ingenieuren weiterentwickelt. Durch den Mechanisms für Sprachinteroperabilität von GraalVM können andere Sprachimplementierungen Sulong als sichere native Funktionsschnittstelle [Gr15] verwenden. Der Kern von Sulong ist auf *GitHub* verfügbar und ist mit über 600 Sternen sehr beliebt.[2] In der Evaluierung von Safe Sulong haben wir außerdem insgesamt 68 Fehler in kleinen Open-Source Projekten gefunden, für die wir eine Fehlerbeschreibung erstellt und eine Behebung des Problems vorgeschlagen haben.[3] Desweiteren haben wir unimplementierte Funkionalität und Probleme in anderen Werkzeugen wie z.B. ASan entdeckt, die wir den Entwicklern gemeldet haben.[4] Die empirischen Studien haben uns dabei geholfen, Fehler in Tools zu finden, die für sicherheitskritische Anwendungen verwendet werden, z. B. fehlerhafte GCC-Builtin Implementierungen in Frama-C[5] [Cu12] und CompCert[6] [Le09].

5 Safe Sulong

Um undefiniertes Verhalten in C Programmen in Angriff zu nehmen haben wir einen Ansatz für die Ausführung unsicherer Sprachen auf der JVM entwickelt. Die Kernidee dieses Ansatzes ist, dass die Semantik eines unsicheren Programms auf die sichere Semantik eines Java-Programms abgebildet werden kann. Durch das Übersetzen einer unsicheren Operation auf eine Folge äquivalenter Java-Operationen verhalten sich beide Ausführungen für legale Eingaben gleich. Da Java jedoch eine sichere Sprache ist und die Semantik der Operationen vollständig spezifiziert ist, verhält sich der Java-Code auch für Operationen, die im C-Code unzulässig sind, in einer definierten Weise. Ein Pufferüberlauf in Java resultiert beispielsweise immer in einer *Exception*, die die Ausführung in einer definierten Weise abbricht. Die Java Virtual Machine optimiert das Programm basierend auf der klar definierten Java-Semantik. Pufferüberläufe und andere Fehler werden somit nicht wegoptimiert. Wir haben diesen Ansatz als ein System namens *Safe Sulong* implementiert.

Sichere Ausführung. Safe Sulong verfügt über zwei Ausführungsmodi, die auf undefiniertes Verhalten unterschiedlich reagieren. Der erste Ausführungsmodus erkennt undefiniertes Verhalten und bricht die Ausführung in einem solchen Fall ab. Der Modus stützt sich auf die automatischen Laufzeitprüfungen der zugrunde liegenden JVM, die z.B. durchgeführt werden, um Pufferüberläufe zu erkennen. In diesem Modus kann Safe Sulong als Sanitizer verwendet werden. Dies ist besonders für ProgrammiererInnen hilfreich, die Fehler in ihren Programmen während der Entwicklung und beim Testen korrigieren können. Der Modus verhindert jedoch, dass Software mit undefiniertem Verhalten ausgeführt werden kann, auch wenn beispielsweise SystemadministratorInnen dieses in bestimmten Fällen als harmlos betrachten. Um die Ausführung solcher Programme zu

[1] `http://www.graalvm.org/`
[2] `https://github.com/graalvm/sulong`
[3] `http://ssw.jku.at/General/Staff/ManuelRigger/ASPLOS18-SafeSulong-Bugs.csv`
[4] `https://github.com/google/sanitizers/issues?utf8=%E2%9C%93&q=author%3Amrigger`
[5] `https://lists.gforge.inria.fr/pipermail/frama-c-discuss/2018-July/005483.html`
[6] `https://github.com/AbsInt/CompCert/issues/243`

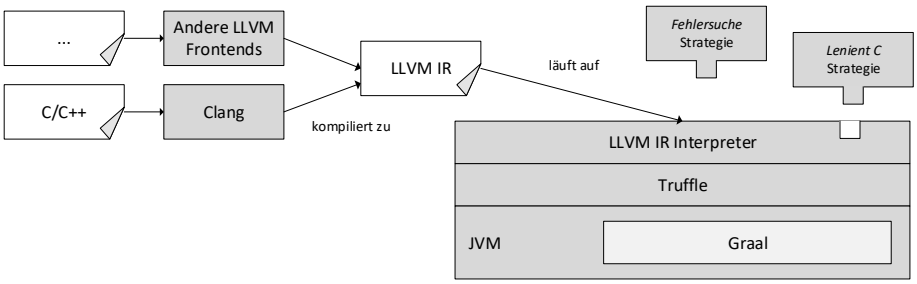

Figure 1: Überblick über die Architektur von Safe Sulong

ermöglichen, haben wir einen zweiten *Lenient C* Ausführungsmodus entwickelt, der eine Semantik für undefinierte Operationen definiert, basierend auf den Erwartungen von ProgrammiererInnen. Bei einem arithmetischen Überlauf definiert Lenient C beispielsweise ein Verhalten, das äquivalent zu dem von x86 Prozessoren ist.

Architektur. Abbildung 1 stellt die Architektur von Safe Sulong überblicksmäßig dar. Safe Sulong basiert auf existierenden Sprach-Frontends des LLVM [LA04]-Compiler-Frameworks, z.B. Clang, um unsichere Eingabesprachen in LLVM-Zwischencode (LLVM IR) zu übersetzen. Beim Kompilieren einer unsicheren Sprache nach LLVM IR deaktivieren wir alle Optimierungen, sodass Operationen die undefiniertes Verhalten verursachen nicht wegoptimiert werden. Der RISC-ähnliche Zwischencode wird auf dem LLVM-IR-Interpreter, den wir in Java implementiert haben (er umfasst ca. 60.000 Codezeilen), ausgeführt. Wie bereits beschrieben, umfasst der Interpreter verschiedene Modi, wie den Lenient-C-Modus und den Fehlersuche-Modus, die als Laufzeitstrategien konfiguriert werden können. Die Implementierung des Interpreters basiert auf dem Sprachimplementierungs-Framework Truffle, das den Just-in-Time-Compiler Graal verwendet, um häufig ausgeführte Funktionen zu Maschinencode zu kompilieren [Wu13]. Da Graal so entwickelt wurde, dass er effizient Prüfungen optimiert (z. B. durch das Entfernen redundanter Begrenzungsprüfungen gegen Pufferüberläufe), erreicht unser Prototyp eine Ausführungsgeschwindigkeit, die mit bestehenden Compilern, die unsicheren Code erzeugen, konkurrenzfähig ist. Wir haben Safe Sulong implementiert und im Hinblick auf die Effektivität des Fehlersuchmodus und der Leistung evaluiert.

Wirksamkeit. Um zu testen, ob Safe Sulong ein wirksames Werkzeug zur Fehlersuche ist, haben wir C-Projekte von Github ausgewählt und mit Safe Sulong ausgeführt, um potenzielle Fehler darin zu finden. Wir wollten auch zeigen, dass in aktuellen Ansätzen Fehler übersehen werden, die jedoch Safe Sulong erkennt. Zu diesem Zweck haben wir jedes der fehlerhaften Programme unter den gleichen Bedingungen mit ASan und Valgrind ausgeführt, d.h. mit den gängigsten Werkzeugen der Compilezeit- und Laufzeitinstrumentierung, um zu überprüfen, ob auch sie die von Safe Sulong erkannten Fehler finden. Insgesamt fanden wir mit Safe Sulong 68 Fehler, wobei es sich beim Großteil um Pufferüberläufe handelte. Danach haben wir die Programme mit Clang und ASan ohne Optimierungen (-O0) kompiliert, da wir auch mit den anderen Werkzeugen möglichst viele Fehler finden wollten. Um zu zeigen, dass das Anwenden von Optimierungen dazu führt, dass bestimmte Fehler nicht erkannt werden, haben wir zusätzlich die Programme auf der Optimierungsstufe -O3 für ASan und Valgrind kompiliert. Wie erwartet, fand Valgrind in beiden Konfigurationen nur etwa die Hälfte der Fehler, da es Pufferüberläufe am Stack und auf globalen Variablen nicht zuverlässig erkennt. ASan ohne Optimierungen hat 60 der 68 Fehler gefunden. Mit Optimierungen wurden nur 56 Fehler gefunden, da die anderen von Clang wegoptimiert wurden. Insgesamt konnten von den 68 Fehlern, die Safe Sulong entdeckte, 8 weder von Valgrind noch von ASan in allen Konfigurationen gefunden werden. Einerseits konnten diese Werkzeuge Fehler aufgrund grundlegender Ein-

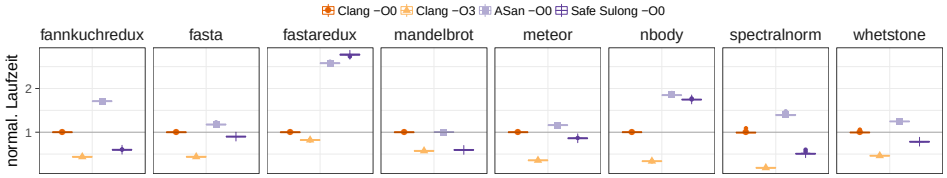

Figure 2: Geschwindigkeit auf den *Computer Language Game Shootouts*.

schränkungen ihrer Ansätze nicht erkennen und weil sie auf Compilern basieren, die undefiniertes Verhalten unter Umständen wegoptimieren. Diese Probleme können nicht ohne weiteres gelöst werden. Andererseits übersahen sie Fehler aufgrund von Problemen in ihrer Implementierung, was durch Verbesserungen oder Korrekturen behoben werden könnte. Allerdings ist unser Ansatz von Natur aus zuverlässiger, da er auf der gut getesteten Implementierung von Java und seiner klar definierten Semantik beruht, was das Wegoptimieren von Fehlern vermeidet.

Geschwindigkeit. In unserer Geschwindigkeitsevaluierung wollten wir zeigen, dass Safe Sulong vergleichbar schnell wie andere Werkzeuge läuft. Ein direkter Vergleich der Laufzeitleistung zwischen verschiedenen Tools ist jedoch nicht fair, da diese unterschiedliche Funktionen bieten. Unsere Messungen sollen daher nur zeigen, dass unser Ansatz praxistauglich ist. Neben der Geschwindigkeit von Safe Sulong haben wir daher auch die von ASan, basierend auf LLVM Version 3.9, und die von Valgrind, Version 3.12, gemessen. Als Baseline haben wir auch die Geschwindigkeit von Programmen gemessen, die mit Clang Version 3.9 kompiliert wurden (jeweils mit und ohne Optimierungen), deren Ausführung daher unsicher ist. Für alle anderen Werkzeuge (auch für Safe Sulong) haben wir die Optimierungen von Clang ausgeschalten, da die Annahme weiterhin galt, dass es unser Ziel ist, so viele Fehler wie möglich zu finden. Wir evaluierten die Werkzeuge auf den Benchmarks des *Computer Language Benchmark Games*, die entworfen wurden, um die Geschwindigkeitsunterschiede zwischen verschiedenen Programmiersprachen zu zeigen. Wir mussten die adaptiven Kompilierungstechniken von Truffle und Graal berücksichtigen, indem wir einen Benchmark-Harness implementierten, der Aufwärmiterationen durchführte. Durch die Ausführung von 50 In-Prozess-Warmup-Iterationen haben wir sichergestellt, dass jeder Benchmark einen stabilen Zustand erreicht hat. Abbildung 2 zeigt Box-Plots für die Ausführungsgeschwindigkeit im Verhältnis zu `Clang -O0`. Wir haben Valgrind vom Graphen ausgeschlossen, da es `10`× bis `58`× langsamer war als `Clang -O0` bei 5 Benchmarks. Die geringste Verlangsamung stellten wir auf `spectralnorm`, `fasta`, und `fannkuchredux` fest (`2.3`, `3.6` und `5.1`). Die Ergebnisse für den Benchmark `binarytrees` haben wir auch ausgeschlossen, weil ASan `14`× langsamer und Valgrind `58`× langsamer als `Clang -O0` war. Diese Verlangsamung war darauf zurückzuführen, dass `binarytrees` zuweisungsintensiv war, was darauf hindeutet, dass die derzeitigen Fehlersuchansätze mit allokationsintensiven Benchmarks nicht gut umgehen können. Auf diesem Benchmark war Safe Sulong nur `1.7`× langsamer als `Clang -O0`. Auf fast allen Benchmarks war Safe Sulong schneller als `ASan`; nur auf `fastaredux` waren beide Werkzeuge etwa gleich schnell. Safe Sulong war außerdem schneller als Programme kompiliert mit `Clang -O0`, außer bei den `fastaredux` und `nbody` Benchmarks. Auf `fannkuchredux` und `mandelbrot` war Safe Sulong sogar gleich schnell wie Programme die mit `Clang -O3` kompiliert wurden. Die langsamste Geschwindigkeit hatte Safe Sulong bei `fastaredux`, wo es `2.5`× langsamer war als `Clang -O0`. Insgesamt zeigen die Ergebnisse, dass die Leistung von Safe Sulong mit anderen Werkzeugen konkurrenzfähig ist und in einigen Fällen sogar die Leistung von Programmen erreicht, die mit unsicheren Compilern kompiliert wurden.

6 Empirische Studien

Als wir Safe Sulong verwendeten um Programme „aus der echten Welt“ auszuführen stellen wir fest, dass viele Programme *unstandardisierte Elementen* wie Inline-Assembly und GCC-Compiler-

Builtins verwendeten. Sie sind Compiler-Erweiterungen und nicht Bestandteil eines offiziellen C-Standards. Inline-Assembly ermöglicht das Einbetten von Maschinencode-Anweisungen direkt in den C-Code. Compiler-Builtins ähneln Standard-Bibliotheksfunktionen, werden jedoch direkt im Compiler implementiert. Unter Linux sind die GCC-Builtins weit verbreitet. Der Aufwand für die Implementierung unstandardisierter Elemente in einem Werkzeug (z. B. einem Compiler oder einem Sicherheitsanalyse-Werkzeug) ist umständlich, da selbst eine Architektur mit einem einzigen komplexen Befehlssatz, wie z. B. x86, etwa 1.000 Anweisungen enthält, und da über 12.000 GCC-Builtins existieren. Um die Implementierung unstandardisierter Elemente in Safe Sulong und anderen Tools zu priorisieren, haben wir deren Verwendung in C/C++-Open-Source-Projekten von GitHub mit Repository-Mining-Methoden untersucht. Wir haben mehr als 1.200 Projekte für die Inline-Assembly Studie analysiert, und fast 5.000 Projekte für die Studie zu Compiler-Builtins.

Häufigkeit. Wir haben festgestellt, dass sowohl Inline-Assembly Fragmente als auch GCC-Builtins weit verbreitet sind; 38% der Projekte verwendeten GCC-Builtins, während 28% der populärsten Projekte Inline-Assembly verwendeten. Normalerweise werden sie jedoch nur an wenigen Stellen im Quellcode verwendet. GCC-Builtins werden normalerweise alle 6.000 Codezeilen (LOC) verwendet, während Inline-Assembly Fragmente sogar nur alle 40.000 LOCs verwendet werden. Unsere Ergebnisse legen nahe, dass ProgrammiererInnen nur einen Teil der verfügbaren unstandardisierten Elemente verwenden. Insgesamt wurden nur ca. 3.000 verschiedene GCC-Builtins und ca. 200 verschiedene Inline-Assembly Fragmente verwendet. Darüber hinaus analysierten wir die historische Entwicklung der Projekte und stellten fest, dass sie meistens GCC-Builtins hinzufügten und selten entfernten. Dies legt nahe, dass unstandardisierte Elemente wahrscheinlich auch in zukünftigen Tools unterstützt werden müssen.

Werkzeug-Unterstützung. Um Werkzeug-Entwickler zu informieren, haben wir verschiedene Implementierungsstrategien evaluiert. Beispielsweise zeigt Abbildung 3 die Anzahl der GCC-Builtins, die implementiert werden müssen, um einen bestimmten Prozentsatz von Projekten zu unterstützen, wenn eine *gierige* Implementierungsreihenfolge angenommen wird. Die Anzahl der zu implementierenden Builtins steigt exponentiell an; Um die Hälfte der Projekte zu unterstützen, müssen 22 Builtins implementiert werden. Um 90% der Projekte zu unterstützen, müssen 106 Builtins implementiert werden und um 99% zu unterstützen, müssen 1.600 Builtins implementiert werden. Wir haben auch untersucht, inwieweit beliebte Werkzeuge Builtins unterstützen, indem wir eine Testsuite schrieben, mit der wir die Implementierung der am häufigsten verwendeten Builtins testeten. Während ausgereifte Compiler alle unsere Testfälle bestanden haben, konnten viele andere Tools aufgrund von falsch implementierten und fehlenden Builtin-Implementierungen nicht alle Testfälle erfolgreich ausführen. Interessanterweise fanden wir zwei fehlerhaft implementierte Builtins im formal verifizierten CompCert-Compiler, der hauptsächlich für sicherheitskritische Anwendungen verwendet wird, da seine Builtin-Implementierungen nicht verifiziert wurden.

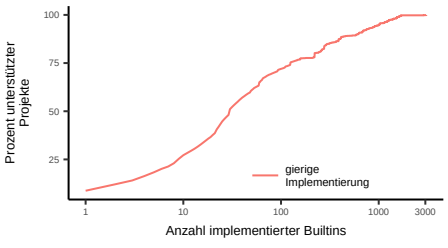

Figure 3: Implementierungsaufwand der GCC-Builtins; Beachten Sie die Exponentialachse.

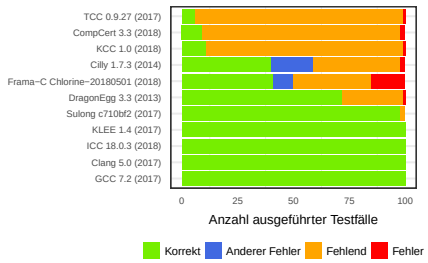

Figure 4: Evaluierung der Unterstützung von GCC-Builtins in verschiedenen Werkzeugen

List. 1: Robuste Implementierung von `strlen()`, die auch für unterminierte Strings funktioniert.

```
size_t strlen(const char *str) {
  size_t len = 0;
  while (size_right(str) > 0 && *str != '\0') {
    len++; str++;
  }
  return len;
}
```

7 Introspektion

In einigen Fällen können ProgrammiererInnen besser als automatische Ansätze wie Safe Sulong auf undefiniertes Verhalten reagieren, zum Beispiel indem sie diese Invarianten überprüfen und etwaige Fehler beheben. ProgrammiererInnen können jedoch keine Objekt-Metadaten wie z.B. Objektgrößen abfragen um illegale Zustände oder Parameter zu überprüfen, da der C-Sprache solche Mechanismen fehlen. Da immer mehr Tools zur Fehlererkennung und Fehlersuche Metadaten aufzeichen, um ihre Prüfungen zu implementieren, haben wir eine Introspektionsschnittstelle entwickelt, mit der ProgrammiererInnen diese Metadaten abfragen können, um undefiniertes Verhalten manuell anzugehen. Diese Introspektionsschnittstelle bietet beispielsweise die Funktion `size_right()` um die Grenze eines Objekts abzufragen. Wir haben die Introspektionsfunktionen in Safe Sulong implementiert und gezeigt, dass das Verfahren auch auf andere Fehlersuchwerkzeuge anwendbar ist, indem wir `size_right()` in ASan, SoftBound, und in MPX-basierter Grenzüberprüfung implementiert haben. Außerdem haben wir basierend auf Introspektion mit verschiedenen Anwendungsfällen experimentiert.

Context-aware Failure-oblivious Computing. Ein Anwendungsfall von Introspektion ist das Abschwächen von Fehlern, ohne die Ausführung des Programms zu beenden, was für Systeme mit hohen Verfügbarkeitsanforderungen (z.B. Servern) nützlich ist. Unsere Idee basiert auf dem *Failure-oblivious Computing* Ansatz, bei dem ungültige Schreib- und Lesevorgänge ignoriert werden [Ri04]. In unserem *Context-aware Failure-oblivious Computing* Ansatz implementieren ProgrammiererInnen manuell Code, der illegale Eingaben behandelt, wobei sie die Semantik der Funktion berücksichtigen, in der die Logik implementiert ist. Dies wird durch List. 1 demonstriert, in dem `strlen()`, eine Funktion die die Länge eines Strings berechnet, auch die Länge berechnen kann wenn das ‘\0’ Zeichen, das für gültige C Strings notwendig ist, fehlt. In einem solchen Fall nimmt die Funktion mittels `size_right()` die Länge des zugrunde liegenden Puffers als Stringlänge an. In vielen Fällen hilft dies dabei, die Ausführung ohne Folgefehler fortzusetzen. Wir haben auch verschiedene andere libc-Funktionen verbessert, um Fehler, die durch ungültige Argumente verursacht werden, abzuschwächen. Ein anderer Anwendungsfall, mit dem wir experimentiert haben, ist die Härtung der C-Standardbibliothek, indem Invarianten der übergebenen Argumente überprüft werden.

Wirksamkeit. Um die Anwendbarkeit von Context-aware Failure-oblivious Computing in realen Projekten zu demonstrieren, haben wir jüngste (d.h. weniger als ein Jahr alte) Pufferüberläufe in weit verbreiteter Software wie Dnsmasq, Libxml2 und GraphicsMagick betrachtet. Wir haben fünf geeignete Fehler in der *Common Vulnerabilities and Exposures*-Datenbank gefunden, bei denen in einer libc-Funktion ein Pufferüberlauf aufgetreten ist. Anschließend haben wir die Anwendungen mit den durch Introspektion erweiterten libc-Funktionen ausgeführt, mit Eingabeparametern die den Fehler auslösen. In vier Fällen konnte die Ausführung erfolgreich fortgesetzt werden. In einem Fall führte ein nachfolgender Pufferüberlauf in der Anwendung dazu, dass die Werkzeuge die Ausführung kontrolliert beendeten. Dies war zu erwarten, da unser Ansatz nur für Funktionen funktioniert, die mit Introspektionsfunktionen erweitert wurden.

Geschwindigkeit. Um die Geschwindigkeit des auf Introspektion basierenden Failure-Oblivious-Computing Ansatzes zu ermitteln, haben wir unterschiedliche Arbeitslasten betrachtet. Erstens haben wir Server in Betracht gezogen, weil sie eine hohe Angriffsoberfläche bieten und sie hochverfügbar sein müssen. Wir haben festgestellt, dass der Introspektions-Mehraufwand keinen Einfluss auf ihre Laufzeitleistung hat. Zweitens haben wir CPU-intensive Lasten betrachtet, nämlich die SPEC CPU 2016-Benchmarks. Bei MPX und SoftBound war der Mehraufwand meist gering und reichte bis zu 13%. Bei ASan war der Mehraufwand jedoch signifikant und reichte bis zu 130%. Dies war zu erwarten, da ASan die Grenzen eines Objektes nicht explizit speichert. Die Berechnung in `size_right()` hat daher eine Komplexität von *O(n)*.

8 Zusammenfassung

Programme, die in unsicheren Sprachen wie C/C++ geschrieben sind, können undefiniertes Verhalten verursachen, was problematisch für die korrekte Ausführung der Programme und die Sicherheit von Computersystemen darstellt. Um undefiniertes Verhalten in Angriff zu nehmen, hat die vorliegende Arbeit in drei Bereichen beigetragen.

Safe Sulong. Erstens haben wir einen Ansatz zur sicheren Ausführung unsicherer Sprachen entwickelt, bei dem ein Interpreter in einer sicheren Programmiersprache geschrieben ist. Wir haben diesen Ansatz als Werkzeug namens Safe Sulong implementiert. Unsere Evaluierung hat gezeigt, dass Safe Sulong Fehler in Sonderfällen erkennt, die andere Tools übersehen (z. B. weil Instrumentierung für Sonderfälle weggelassen wurde). Der Vorteil von Safe Sulong ist es, dass es auf automatischen Laufzeitprüfungen der zugrunde liegenden virtuellen Maschine basiert und einen optimierenden Compiler verwendet, der undefiniertes Verhalten nicht ausnutzt. Außerdem haben wir demonstriert, dass Programme, die von Safe Sulong ausgeführt laufen, in einigen Fällen in etwa gleich schnell ausgeführt werden wie Programme, die von unsicheren Compilern kompiliert wurden. Safe Sulong ist jedoch immer noch ein Forschungsprototyp und muss noch hinsichtlich seiner Vollständigkeit erweitert werden, um sein Verhalten bei großen Programmen zu evaluieren.

Empirische Studien. Zweitens haben wir die Verwendung von C-Spracherweiterungen, nämlich Inline-Assembly und GCC-Builtins, in C-Projekten analysiert. Wir haben festgestellt, dass diese unstandardisierte Elemente häufig verwendet werden, was Werkzeugautoren Anreize bietet, sie in vorhandenen Werkzeugen zur Fehlersuche zu unterstützen. Aufgrund des hohen Implementierungsaufwands ist eine vollständige Implementierung jedoch manchmal nicht realisierbar, da beispielsweise über 10.000 GCC-Builtins existieren. Unsere Ergebnisse deuten darauf hin, dass bereits durch die Implementierung einer kleinen Teilmenge von GCC-Builtins und Inline-Assembly eine große Anzahl von Projekten unterstützt werden kann, da die meisten Projekte eine gemeinsame Teilmenge dieser Elemente verwenden. Unsere Analyse der historischen Entwicklung von GCC-Builtins in Projekten legt nahe, dass es sich nicht um eine stagnierende Funktionalität handelt, so dass Werkzeuge sie wahrscheinlich auch in Zukunft unterstützen müssen.

Introspektion. Drittens haben wir einen Ansatz vorgeschlagen, um ProgrammierInnen Metadaten zur Verfügung zu stellen, die von vorhandenen dynamischen Fehlersuchwerkzeugen aufgezeichnet werden, damit diese die Metadaten verwenden können, um die Robustheit von Bibliotheken zu verbessern. Wir haben diesen Ansatz in verschiedenen Werkzeugen implementiert um zu zeigen, dass der Ansatz nicht nur mit Safe Sulong funktioniert. Wir haben Context-aware Failure-oblivious Computing als eine auf Introspektion basierende Technik vorgeschlagen, die es erlaubt, Programme auszuführen auch wenn Pufferüberläufe auftreten. Wir haben diesen Ansatz hinsichtlich der Wirksamkeit in einer Fallstudie mit realen Fehlern in gängigen Anwendungen und hinsichtlich der Geschwindigkeit evaluiert. Die Ergebnisse deuten an, dass Context-aware Failure-oblivious Computing dazu verwendet werden kann, um Fehler zu mindern und Programme weiter auszuführen. Bei Tools, bei denen der Introspektionszugriff effizient implementiert werden konnte,

waren die Geschwindigkeitseinbußen von Introspektion gering. Daher glauben wir, dass Context-aware Failure-oblivious Computing in Industrieszenarien eingesetzt werden könnte.

References

[Cu12] Cuoq, Pascal; Kirchner, Florent; Kosmatov, Nikolai; Prevosto, Virgile; Signoles, Julien; Yakobowski, Boris: Frama-C: A Software Analysis Perspective. In: Proceedings of SEFM'12. Springer-Verlag, Berlin, Heidelberg, pp. 233–247, 2012.

[Gr15] Grimmer, Matthias; Seaton, Chris; Würthinger, Thomas; Mössenböck, Hanspeter: Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In: Proceedings of MODULARITY 2015. ACM, New York, NY, USA, pp. 1–13, 2015.

[LA04] Lattner, Chris; Adve, Vikram: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of CGO '04. IEEE Computer Society, Washington, DC, USA, pp. 75–, 2004.

[Le09] Leroy, Xavier: Formal Verification of a Realistic Compiler. Commun. ACM, 52(7):107–115, July 2009.

[NS07] Nethercote, Nicholas; Seward, Julian: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: Proceedings of PLDI '07. ACM, New York, NY, USA, pp. 89–100, 2007.

[Ri04] Rinard, Martin; Cadar, Cristian; Dumitran, Daniel; Roy, Daniel M.; Leu, Tudor; Beebee, Jr., William S.: Enhancing Server Availability and Security Through Failure-oblivious Computing. In: Proceedings of OSDI'04. USENIX Association, Berkeley, CA, USA, pp. 21–21, 2004.

[RPM18] Rigger, Manuel; Pekarek, Daniel; Mössenböck, Hanspeter: Preventing Buffer Overflows by Context-aware Failure-oblivious Computing. In: Proceedings of NSS '18. 2018.

[Se12] Serebryany, Konstantin; Bruening, Derek; Potapenko, Alexander; Vyukov, Dmitry: AddressSanitizer: A Fast Address Sanity Checker. In: Proceedings of USENIX ATC'12. USENIX Association, Berkeley, CA, USA, pp. 28–28, 2012.

[Wa12] Wang, Xi; Chen, Haogang; Cheung, Alvin; Jia, Zhihao; Zeldovich, Nickolai; Kaashoek, M. Frans: Undefined Behavior: What Happened to My Code? In: Proceedings of APSYS '12. ACM, New York, NY, USA, pp. 9:1–9:7, 2012.

[Wu13] Wuerthinger, Thomas; Wimmer, Christian; Wöß, Andreas; Stadler, Lukas; Duboscq, Gilles; Humer, Christian; Richards, Gregor; Simon, Doug; Wolczko, Mario: One VM to Rule Them All. In: Proceedings of Onward! 2013. ACM, New York, NY, USA, pp. 187–204, 2013.

Manuel Rigger wurde am 13.09.1990 in Schwaz, Österreich geboren. Er begann seine Informatiker- und Forscher-Karriere mit einem Bachelorstudium an der *Johannes Kepler Universität (JKU) Linz*. Während seines *Software Engineering* Masterstudiums an der JKU absolvierte er ein Auslandssemester an der *National Taiwan University*. Bevor er sein Masterstudium abschloss verweilte er zwei Jahre lang in China, wo er an der *Xiamen University* ein Masterstudium in Chinesischer Philosophie absolvierte. Seine neu-erlangten Chinesischkenntnisse inspirierten den Namen seines Informatik-Dissertationsprojektes *Sulong*. In 2018 promovierte er an der JKU Linz unter der Betreuung von Prof. Dr. Dr. h.c. Hanspeter Mössenböck. Ab Februar 2019 wird er an der ETH Zürich als Postdoktorand unter der Betreuung von Prof. Dr. Zhendong Su arbeiten.