Elastic Program Transformations: Automatically Optimizing the Reliability/Performance Trade-off in Systems Software

THÈSE N^{O} 7745 (2017)

PRÉSENTÉE LE 8 JUIN 2017

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DES SYSTÈMES FIABLES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Jonas Benedict WAGNER

acceptée sur proposition du jury:

Prof. W. Zwaenepoel, président du jury
Prof. G. Candea, directeur de thèse
Prof. S. Nagarakatte, rapporteur
Prof. J. Regehr, rapporteur
Prof. J. Larus, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2017

Jonas Wagner: *Elastic Program Transformations: Automatically Optimizing the Reliability/Performance Trade-off in Systems Software*

Soli Deo gloria

ABSTRACT

Performance and reliability are important yet conflicting properties of systems software. Software today often crashes, has security vulnerabilities and data loss, while many techniques that could address such issues remain unused due to performance concerns.

This thesis presents *elastic program transformations*, a set of techniques to manage the trade-off between reliability and performance in an optimal way given the software's use case. Our work is based on the following insights:

- Program transformations can be used to tailor many software properties: they can make software easier to verify, safer against security attacks, and faster to test. Developers can write software once and use transformations to subsequently specialize it for different use cases and environments.

- Many classes of transformations are elastic: they can be applied selectively to parts of the software, and both their cost and effect scale accordingly.

- The trade-off is governed by the Pareto Principle: the right choice of program transformations yields 80% of the benefit for 20% of the cost.

This thesis makes four contributions that use these insights:

1. We developed -Overify, a strategy and technique for choosing compiler optimizations to make programs easier to verify, rather than faster to run. -Overify demonstrates that program transformations have the power to reduce verification time by up to 95×.

2. We show that known program transformations to detect memory errors and undefined behavior can be elastic. We developed ASAP, a technique and tool to apply these transformations selectively to make a program as secure as possible, while staying within a maximum overhead budget specified by the developer. For a maximum overhead of 5%, ASAP can protect 87% of the critical locations in CPU-intensive programs.

3. We improve the performance of fuzz testing tools that rely on program transformations to observe the software under test. We present FUSS, a technique and tool for focusing these transformations where they are needed, improving the time to find bugs by up to 3.2×.

4. We show that elasticity is a useful design principle for program transformations. In the BinKungfu project, we develop a novel elastic transformation to enforce control-flow integrity. Exploiting elasticity reduces the number of programs that violate performance constraints by 59%.

The essence of our work is identifying elasticity as a first-class property of program transformations. This thesis explains how elasticity manifests and how -Overify, ASAP, FUSS and BinKungfu exploit it automatically. We implemented each technique in a prototype, released it as open-source software, and evaluated it to show that it obtains more favorable trade-offs between reliability and performance than what was previously possible.

Keywords: systems software, program transformations, elasticity, compilers, instrumentation, profiling

ABSTRACT (DEUTSCH)

Geschwindigkeit und Zuverlässigkeit sind zwei wichtige, aber miteinander im Konflikt stehende Eigenschaften von Systemsoftware. Heutzutage sind Crashes, Sicherheitslücken und Datenverlust traurige Realitäten. Dennoch bleiben Techniken gegen solche Probleme häufig ungenutzt, weil sie die Performance zu stark beeinträchtigen.

Diese Dissertation präsentiert *elastische Programmtransformationen*: Techniken, um für jede Einsatzumgebung eines Programms die optimale Balance zwischen Zuverlässigkeit und Performance zu finden. Unsere Arbeit basiert auf folgenden Erkenntissen:

- Programmtransformationen beeinflussen viele der Eigenschaften von Software; mit den richtigen Transformationen wird Software einfacher zu verifizieren, sicherer gegen Attacken, und schneller zu testen. Entwickler schreiben bloss eine Version ihrer Software, und spezialisieren sie dann mittels Transformationen für verschiedene Umgebungen und Anwendungsbereiche.
- Viele Klassen von Programtransformationen sind elastisch. Sie können gezielt auf Teile eines Programms angewandt werden, was sowohl die Kosten als auch den Nutzen reduziert.
- Das Pareto-Prinzip ermöglicht vorteilhafte Trade-offs: eine sorgfältige Auswahl der Transformationen erreicht 80% des Nutzens mit bloss 20% der Kosten.

Diese Arbeit enthält vier Forschungsbeiträge, die sich diese Erkenntnisse zu Nutze machen:

1. Wir entwickelten -Overify: eine Strategie und Technik zur Auswahl von Compileroptimierungen mit dem Ziel, Programme einfacher zu verifizieren statt schneller auszuführen. -Overify beweist, dass Programmtransformationen die Macht haben, die zur Softwareverifizierung benötigte Zeit um bis zu 95× zu verringern.
2. Wir zeigen, dass Transformationen zum Entdecken von Speicherzugriffsfehlern und undefinierten Verhalten elastisch sein können. Wir entwickelten ASAP, eine Technik zur gezielten Anwendung solcher Transformationen. ASAP macht Programme so sicher, wie es für ein bestimmtes Overheadbudget möglich ist. Für ein Overheadlimit von 5% kann ASAP 87% aller kritischen Stellen in CPU-intensiver Software absichern.

3. Wir erhöhen die Leistung von Fuzz Testing. Diese Testmethode verwendet Programmtransformationen, um den Fortschritt des Testvorgangs laufend zu analysieren. Unsere Technik FUSS fokussiert diese Transformationen genau dort, wo sie nötig sind, und reduziert dadurch die Zeit zur Fehlersuche um bis zu 3.2×.

4. Wir zeigen, dass Elastizität ein nützliches Designprinzip für Programmtransformationen ist. In unserem BinKungfu-Projekt entwickelten wir eine neuartige, elastische Transformation zum Schutz der Control-Flow-Integrity eines Programms. Dank ihrer Elastizität konnten wir die Anzahl Programme mit Performanceproblemen um 59% verringern.

Der Kern unserer Arbeit ist die Erkenntnis, dass Elastizität eine zentrale Eigenschaft von Programmtransformationen ist. Diese Dissertation erklärt, wie Elastizität auftritt und wie -Overify, ASAP, FUSS und BinKungfu sie automatisch nutzen. Wir implementierten jede dieser Techniken und veröffentlichten unsere Prototypen als freie Software. Unsere Auswertung zeigt, dass sie eine bessere Kombination von Geschwindigkeit und Zuverlässigkeit erreichen, als bisher möglich war.

Schlagwörter: Systemsoftware, Programmtransformationen, Elastizität, Compiler, Instrumentation, Profiling

PUBLICATIONS

Some of the ideas presented in this dissertation first appeared in the following publications:

- Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. "High System-Code Security with Low Overhead." In: *IEEE Symp. on Security and Privacy*. 2015
- Jonas Wagner, Volodymyr Kuznetsov, and George Candea. "-OVERIFY: Optimizing Programs for Fast Verification." In: *Workshop on Hot Topics in Operating Systems*. 2013

ACKNOWLEDGMENTS

Now this is the Law of the Jungle—
as old and as true as the sky;
And the Wolf that shall keep it may prosper,
but the Wolf that shall break it must die.

As the creeper that girdles the tree-trunk
the Law runneth forward and back—
For the strength of the Pack is the Wolf,
and the strength of the Wolf is the Pack.

Rudyard Kipling, *The Second Jungle Book*

My thanks go to the many people who journeyed with me through this PhD. It is an amazing feeling to remember the deep conversations, activities, tough research discussions, fine dinners, and interesting insights that I have shared with all of you. In fact, these memories made the acknowledgement section the easiest part of the thesis to write, and so this is where I started.[1]

First I thank George Candea, my adviser who taught me not just computer science, but how to communicate skillfully, to think clearly, to strive for excellence, to manage time and priorities, to be a leader, to not fear feedback, and much more. Thank you.

I am grateful to James Larus, Santosh Nagarakatte, John Regehr, and Willy Zwaenepoel, who generously agreed to form my thesis committee.

I had the chance to work in a lab with very smart, hard-working labmates who were also fun to be with. Because it is impossible to reduce the great impact you had into a few words, I will just list your names. In alphabetical order: Adrian Herrera, Alexandre Bique, Alexandru Copoţ, Amer Chamseddine, Ana Sima, Arseniy Zaostrovnykh, Baris Kasikci, Benjamin Schubert, Bin Zhang, Cristi Zamfir, Daniel Mahu, Francesco Fucci, Georg Schmid, João Carreira, Loïc Gardiol, Lucian Mogoşanu, Petr Zankov, Radu Banabic, Roger Michoud, Silviu Andrica, Stefan Bucur, Tong Che, Vitaly Chipounov, Vladimir Diaconescu, and Vova Kuznetsov.

Special thanks are due to those who have worked with me on the same projects. Vova Kuznetsov, who understands the most complex topics and put them for me in simple terms. Johannes Kinder: I am very grateful for your advice and your support, even if that meant you had to stay up until 3am for a submission deadline. Radu Banabic, who kept his wit and sense of humor even through periods of intense work.

[1] Credit for the write-the-easiest-part-first technique goes to Thomas Rizzo.

These projects gave me a chance to be advised by, and to discuss with, great people. Thank you, Rachid Guerraoui, Ed Bugnion, Viktor Kunčak, and Răzvan Deaconescu. I also collaborated with several students: thanks to Florian Vessaz, Quentin Cosendey, Azqa Nadeem and Alexandra Sandulescu for joining me in my research.

I would like to particularly thank Nicoletta Isaac. You made life at DSLab so much nicer. I apologize for any stress my very spontaneous internships may have caused ;-) Also, thanks to Céline Brzak and Beary for making our lab a friendlier place.

It is impossible to name all the friends that I've made at EPFL during my PhD. I would just like to say thanks to Peter Perešíni, Dimitri Melissovas, Călin Iorgulescu, Bogdan Alexandru Stoica, and David Teksen Aksun for dances, music and discussions.

I am very grateful for the chance to do two internships during my PhD. During Summer 2015, I entered into the fascinating world of Google, a world bursting with things to learn and power to do impactful work. I want to thank Gogul Balakrishnan and Domagoj Babic, as well as the LASER team and my fellow interns, for that amazing experience.

A year later, I visited Prof. Dawn Song's group at the University of California at Berkeley. I received there a knowledge of X86 security, an appreciation for Risotto, an intense training in writing working code fast, a bit of understanding of distributed failure-tolerant systems, and a taste of hip, bustling Berkeley. Thanks to Dawn Song, Luca Guerra, Mauro Matteo Cascella, Chao Zhang, Riccardo Schirone, Gilad Katz, Jože Rožanec, Aristide Fattori, Rundong Zhou, Jinghan Wang, and Heng Yin.

During my PhD, there have been many moments of intense joy thanks to the associations I was part of. It is an interesting fact that the only *nuit blanche* I pulled during the PhD was spent organizing the Helvetic Coding Contest rather than submitting a paper. PolyProg has been a great source of friendships and learning. Cevi gave me the possibility to escape from computers into a world of campfires, hikes and laughter. I'm deeply thankful for the support, joy, and love I received from Westlake Church, the GBU, Peninsula Bible Church, and the Young Disciples at Berkeley.

Und i bi unändlech dankbar für d'Moni, und für mini Familie. Dir syt di Beschte!

CONTENTS

Part I

SETTING THE STAGE

Where we explain why we want systems that are both fast and reliable, why such systems are hard to build, and how program transformations can achieve a decent trade-off between speed and reliability.

We also look thoroughly at related work in program transformations, performance, reliability, and verification.

1 INTRODUCTION

> Not getting lost in the complexities of our own making, [. . .] that is the key challenge computing science has to meet.
>
> E. W. Dijkstra, 1984 [44]

1.1 PROBLEM: BUILDING SYSTEMS THAT ARE BOTH FAST AND RELIABLE IS HARD

Software is arguably the most complex type of artifact that humans have ever built. Its ability to solve complex problems makes software so valuable. Thus, businesses, industry, governments and individuals delegate more and more tasks to software and trust it to a great extent. To be worthy of this trust, software must be reliable.

Software complexity tends to reduce reliability. Complex software is often too large to be fully understood, and each additional 1000 lines of code introduce up to 50 new bugs [93]. Complexity also manifests in the large number of possible inputs and the many possible interactions between components. These factors lead to unforeseen emerging behavior that can cause reliability problems.

The desire for performance further increases the difficulty of making software reliable. To make systems fast, their authors use advanced algorithms, add a variety of caches and batch processing techniques, and resort to low-level languages that make efficient use of the hardware. Each of these techniques increases complexity and can introduce new reliability problems.

The trade-off between reliability and performance is particularly hard to resolve for systems software such as operating systems, web browsers and libraries. This type of software is used by higher-level applications, and thus its speed and reliability influences the entire software stack.

1.1.1 *Consequences of unreliable software*

Unreliable software can cause great harm. In 2002, NIST published a study that estimated the cost of bugs to be 0.6% of the US GDP. More precisely, the study quantifies costs due to lost revenue, software updates, data loss, etc., that could be saved if better tools were available to prevent bugs.

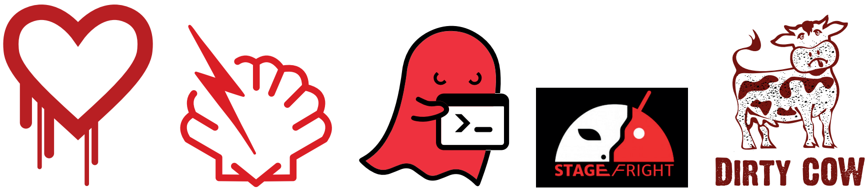


Figure 1: Prominent bugs discovered between 2011 and 2016

While performing the work presented in this thesis, we witnessed the discovery of Heartbleed, Shellshock, Ghost, Stagefright, and Dirty COW (Figure 1). Each of these is a software bug with sufficient impact to receive a catchy name and a logo. These bugs are located in widely deployed systems software, and thus affected a large number of users. For example, a survey by Netcraft found half a million web sites that were vulnerable to the Heartbleed vulnerability at the time it was disclosed [101] Similarly, all versions of Android published between 2010 and 2015 contained the vulnerable Stagefright code, affecting nearly one billion devices [92]. Fixing these bugs required a software update that typically caused service downtime.

In the case of Stagefright, performing a software update required substantial effort. The vulnerability revealed a fractured Android ecosystem, where software updates require collaboration between Android developers, cell phone carriers, system-on-chip manufacturers, and phone manufacturers. As a result, the Android team at Google designed a new update process with monthly security fixes for select phones [5].

A more positive consequence of software bugs is the research into tools and techniques for increasing reliability, which we will look at in the next section.

1.1.2 *Existing tools do not solve the problem*

Over time, software developers have built numerous tools to make it easier to create reliable software. These tools include compilers and their warnings, linters, static analyzers, dynamic analyzers, model checkers, automatic testcase generators, hardware mechanisms to isolate and protect programs, replication for fault tolerance, randomization, instrumentation techniques, and many more.

Tools do have the potential to make systems software more reliable. For example, automated fuzz testing tools discovered variants of the Shellshock bug that had remained after the original bug was fixed. Developers used memory safety checkers to scrutinize OpenSSL's source code after the Heartbleed bug had been discovered, and found more bugs. Tool-related publications usually contain impressive lists of trophies (e.g., [118, 141]), i.e., bugs found using the tool.

As a result of their successes, the use of tools for reliability is becoming more wide-spread. In the last years, an increasing number of research tools were transformed into commercial products (e.g., [128, 118, 98]. Competitions like the Cyber Grand Challenge [40] raised public awareness of software security issues. Bug bounty programs offer increasing rewards for security issues, indicating that they become harder to find [48]. These trends show that tools do contribute to more reliable software.

Yet many tools live in small niches. Why is this so? What is the difference between an ubiquitous technique and one that is rarely used? In their survey of solutions for memory safety [124], Szekeres et al. identify *protection*, *cost* and *compatibility* as the three criteria that influence whether a technique is adopted.

Szekeres and his co-authors found that successful techniques have a low performance cost and high compatibility. The protection offered by successful techniques is often weak or probabilistic. Yet this does not hinder their adoption, indicating that developers and system builders are in principle willing to use any tool that helps. However, performance is crucial: the study showed that no tool with more than 10% runtime overhead found widespread adoption in production systems. Compatibility is also a major concern. If a tool produces false alarms during normal operation, or cannot work with legacy systems, or limits how developers design their systems, this tool will remain a niche tool.

Interestingly, existing tools *can* be classified based on their overhead, protection, and compatibility. These values are fixed for a given tool. Developers choose between using a tool and paying *all* its cost, or not using the tool and getting none of its benefits. It is a binary choice. This need not be the case, as we show in the next section.

1.2 THE ELASTICITY PRINCIPLE

The heart of this dissertation is the formulation of the Elasticity Principle:

Elasticity Principle. *The cost and benefits of program transformations can be flexibly traded against each other, according to needs and use case.*

The Elasticity Principle allows developers to use program transformations selectively. It postulates that program transformations are not monolithic, but rather the sum of numerous small changes. By selecting a subset of these changes, developers can often achieve most of the desired effect of a transformation while keeping its performance impact low.

We call the Elasticity Principle a "principle" because we have observed it for a large number of program transformations and in many contexts. We have made use of it to reduce the overhead of run-time checking tools, to improve the overall performance of fuzz testers, and to design a better protection mechanism against return-oriented programming.

This thesis describes the insights behind the Elasticity Principle and the applications that build on it, but also explores its limits. We must keep in mind that systems are diverse and that making them reliable is a difficult problem. Program transformations offer a solution for parts of this problem, particularly when unreliability is due to low-level bugs. Selective instrumentation techniques based on the Elasticity Principle are useful in a subset of these cases, when the performance impact of program transformations is too high to use them in their original form. When selective instrumentation techniques apply, developers can trade a small amount of performance for a large part of the desired effect.

The formulation of the Elasticity Principle is novel, and this thesis presents the first techniques that apply the principle to off-the-shelf program transformations.

Two factors enabled our work: first, the recent rise in popularity of program transformations and related techniques, and second, the integration of ideas from different fields. Around the time our work started, mainstream compilers gained support for program transformations to detect memory errors, data races, and undefined behavior [118, 119, 36]. Their integration into compilers revealed a common structure and allowed us to build generic tools that would work with multiple types of program transformations. Our work combines these compiler techniques with ideas from security and operating systems, fields where our research group had prior experience. Overall, this put us in the right place at the right time to discover the Elasticity Principle.

1.2.1 *Insights*

In the course of our research, we found program transformations to be widely applicable and powerful. We explored several techniques to increase their effectiveness through elasticity. In that process, we discovered the following three insights into the origin, the nature, and the applicability of elastic program transformations.

1.2.1.1 *Program transformations can specialize one program for many environments*

Since the days of FORTRAN, developers have been using optimizing compilers. These employ program transformations that make programs faster. The transformations range from localized simplifications (e.g., transforming `x / 2` into `x >> 1`) to structural changes (e.g., swapping nested loops to take advantage of processor caches). Compiler optimizations were essential for the acceptance of FORTRAN.

Because of their success, developers are accustomed to the fact that transformations can make a program faster.

What surprised us is that the right program transformations can also make software *easier to verify*, *faster to test*, and *safer*. Figure 2 shows an example. We adapted the example from Ball et al.'s paper on the SLAM verifier [12], where it demonstrates that verifiers need to carefully reason about control flow. The verifier tries to prove that the program correctly pairs calls to `lock` and `unlock`. In the original program in Figure 2a, this is indeed difficult: there exists a static path through the program that calls `unlock` at lines 7 and 12 without an intervening call to `lock`. The verifier needs to analyze the loop condition to prove that this cannot happen.

The equivalent transformed program from Figure 2b is much easier to verify. Its control flow has been simplified by a transformation called *jump threading*. Modern compilers can perform jump threading, but they use it sparingly because it duplicates code (snippet (1) in this case) and increases program size.

The example shows how different environments and use cases call for specialized versions of a program. We prefer version 2a in a space-constrained environment (e.g., a CPU with a small instruction cache), but prefer 2b for verification.

The specialization is best done *after* the code has been written, using automatic transformations. This means that developers only have to write a single version of their code. In fact, at that time developers might not even know all the environments where their program will be used.

While the program versions 2a and 2b are equivalent, in general program transformations can make arbitrary semantic changes. For example, transformations can add code to the program to abort it when bad behavior is detected. This flexibility makes program transformations general, and applicable in more settings than just optimization.

1.2.1.2 *Many program transformations are elastic: costs and effects scale*

We call a program transformation *elastic* if we can apply it selectively to a program and obtain some of its benefit at reduced cost. Elasticity means that a program transformation is scalable. Like scaling an elastic cloud service by adding more servers, engineers can scale a program's reliability or performance by controlling the amount of program transformations used.

Our insight is that many classes of program transformations are elastic because they consist of many small and independent changes. We call these *transformation atoms.* For example, a transformation that protects programs from memory errors would modify all program instructions that access memory, inserting thousands of little checks to ensure the addresses being accessed are valid. Because these checks

```
void foo() {
  do {
    lock();
    n_old = n;
    /* (1) use shared data ... */
    if (/* (2) some condition */) {
      unlock();
      n += 1;
      /* (3) thread-local work ... */
    }
  } while (n_old == n);
  unlock();
}
```

(a) A program for which we would like to verify that calls to `lock` and `unlock` are properly paired. This is challenging because the verifier has to find the relationship between condition (2) and `n_old == n`.

```
void foo_opt() {
  lock();
  /* (1a) use shared data ... */
  while (/* (2) some condition */) {
    unlock();
    /* (3) thread-local work ... */
    lock();
    /* (1b) use shared data ... */
  }
  unlock();
}
```

(b) The same program after *jump threading* and *dead code elimination* transformations. This is easier to verify, because all static paths through the program call `lock` and `unlock` in pairs.

Figure 2: Program transformations can make programs easier to verify.

are numerous and independent, choosing any subset leads to a valid, selectively transformed program that obtains part of the benefit of the transformation and the transformation can be elastic.

Traditionally, program transformations were thought to be monolithic. The choice of enabling or disabling a transformation was a binary choice that came with a fixed performance impact. In some cases, users had manual control, e.g., by using a blacklist of functions that should not be transformed; but systems that controlled transformations automatically, on a fine-grained level, were rare.

To be fair, elasticity is not universal. Even transformations that are elastic may come with some fixed costs. For example, transformations to protect memory *accesses* might depend on instrumenting all memory *allocations* without exception, which introduces extra costs that do not scale with the number of memory access checks. Yet there are a number of findings that make us think of elasticity as a first-class concept:

- Costs scale: Particularly for CPU-bound applications, it is possible to reduce overheads to below 5% by exploiting elasticity.
- Effects scale: We found many instances where selective transformations preserve utility. For example, selective transformations are sufficient to guide fuzzers and protect against security vulnerabilities in cold code.
- Elasticity is a useful design principle: Designing for elasticity can sometimes eliminate fixed costs. We give an example of a binary hardening transformation that benefited from such a design in Section 4.4 of this thesis.

1.2.1.3 *Elasticity enables exploitation of the Pareto Principle to achieve better trade-offs*

What happens when one scales down an elastic program transformation to obtain 80% of its effect? In our experience, its performance cost reduces to 20% of the original cost. This disproportionality appears so consistently in many domains that it received a name: the Pareto Principle.

The Pareto Principle states that in a population that contributes to some common effect, most of that effect comes from a “vital few” participants. Joseph Juran wrote about the universality of this principle, and named it after Vilfredo Pareto, who had observed that the 20% richest land owners owned 80% of the land in Italy [73]. Empirically, many other phenomena follow similar distributions, also in software systems:

For example, the Pareto Principle applies to software reliability: Steve Ballmer said that data from Microsoft Error Reporting revealed that 80% of all errors are caused by 20% of software bugs [15]. In

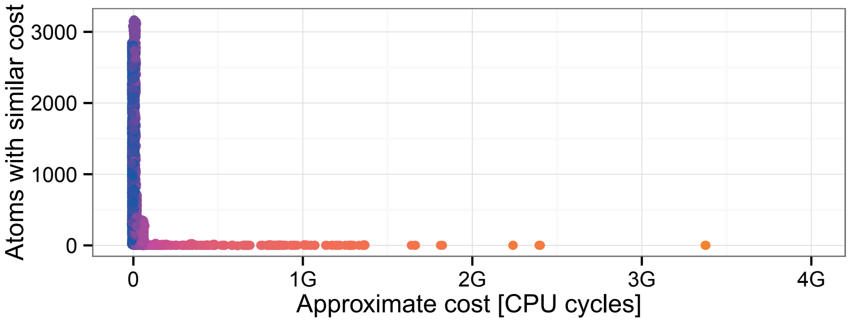


Figure 3: The 3,864 transformation atoms in the `bzip2` benchmark. A dot's color and position on the x-axis corresponds to the corresponding atom's cost, i.e., the number of CPU cycles it consumed in the experiment. The single most expensive atom is as expensive as the 3,414 cheapest atoms together.

performance engineering, 20% of a system's code uses 80% of CPU time [41].

An important caveat is that the choice of causes matters: 20% of all bugs sampled at random would cause only 20% of errors. It is the fact that Windows Error Reporting can identify the *most severe* bugs that makes it so powerful. Similarly, to exploit the Pareto Principle, tools need to identify those transformation atoms that have the biggest effect.

To give an intuition for how these "vital few" atoms could be identified, Figure 3 plots the transformation atoms obtained when transforming the `bzip2` program with AddressSanitizer [118]. The effect in this case is the atom's performance impact, shown on the x-axis and measured in billions of CPU cycles. We make three observations:

- Expensive atoms consume billions of CPU cycles. This makes it possible to identify them using various profiling mechanisms.
- The effect distribution is highly skewed, with many cheap atoms and few extremely expensive ones. This is the type of power-law distribution that enables exploitation of Pareto's Principle.
- Even a small program like `bzip2` has thousands of atoms, so that developers can choose many points in the trade-off between performance and (in this case) memory safety.

1.2.2 *Contributions*

This thesis makes four contributions, which we will now introduce. Each contribution uses the Elasticity Principle in a different way, and demonstrates its applicability in a different setting.

1.2.2.1 *Program transformations for verification*

We present -OVERIFY, a technique for compilers to transform programs for easier verification. This addresses the problem that code optimized for fast execution can be hard to digest for verification tools. Compilers can use -OVERIFY to identify those optimizations that are beneficial for verification. With the right choice of optimizations, symbolic execution of the Coreutils UNIX tools completes up to 95× faster.

More importantly, -OVERIFY demonstrates that program transformations can have an orders-of-magnitude impact in a use case where they have not traditionally been applied. Software has to perform in a number of secondary environments besides production machines. We identified the compiler as a sweet spot for optimizing these secondary use cases, because it has both access to the software in its richest form, as well as knowledge of the target environment.

-OVERIFY adapts the optimization process through the use of a cost model. This model informs the compiler whether or not a transformation is beneficial, and thus controls how often any class of transformations are applied. In retrospect, when developing -OVERIFY we became aware that transformations are trade-offs, and using them selectively is beneficial.

We describe and evaluate -OVERIFY in Chapter 3.

1.2.2.2 *Selective sanity checks with high coverage and low overhead*

We developed ASAP, a technique to manage the overhead of instrumentation code such as memory safety checks. ASAP applies this instrumentation selectively to generate a program that is *as secure as possible* while staying within a configurable overhead budget. This addresses the problem that developers often cannot protect production software with instrumentation because the instrumentation has a high, fixed, overhead.

The main contribution of the ASAP project is the concept of *elastic instrumentation.* ASAP works with a variety of instrumentation mechanisms—AddressSanitizer, SoftBound, UndefinedBehaviorSanitizer, ThreadSanitizer, assertions—and shows that most of their overhead is elastic. Empirically, this overhead is due to a few "hot" instrumentation atoms, and so ASAP benefits from exploiting the Pareto Principle. Moreover, we studied known bugs and vulnerabilities that are typically detected using instrumentation, and found that ASAP can estimate how many of these would be detected in a selectively instrumented program. This means that developers can make informed choices about the best points in the security/overhead trade-off space.

We explain the semantics of elastic instrumentation in Section 4.1. We present ASAP itself in Section 4.2 and evaluate it, together with other use cases of elastic instrumentation, in Section 4.5.

1.2.2.3 *Focused instrumentation for fast fuzzing*

We developed FUSS, a technique that reduces the amount of program instrumentation needed for coverage-guided fuzz testing. FUSS solves the problem that this instrumentation is expensive: coverage-guided fuzzers spend on average 54% of their time in instrumentation code. By exploiting the elasticity of instrumentation, FUSS reduces this time while preserving the instrumentation's effect, which enables fuzzers to find bugs up to 3.2× faster.

FUSS is an attractive technique for two reasons. First, fuzz testing is a CPU-intensive process that is often parallelized on large clusters of machines, and so there is a big potential for cost and energy savings through more efficient instrumentation. Second, by observing the fuzzing process FUSS can predict precisely where in the program instrumentation yields useful information and where it just slows things down. Thus, FUSS can potentially achieve efficiency gains without reducing the instrumentation's effect at all. The main contribution of FUSS is identifying and selecting the subset of instrumentation that is beneficial for fuzzing.

We present the design of FUSS in Section 4.3, and its evaluation in Section 4.5.

1.2.2.4 *Elastic binary hardening*

We enhanced BinKungfu, a tool that protects binaries against control-flow hijack attacks, using elastic program transformations. This addresses the problem that traditional control-flow enforcement has fixed overheads, which exceed the limits of BinKungfu's use case. With elastic transformations, BinKungfu was able to add some (albeit partial) protection to programs while respecting the performance bounds.

Our work introduces a novel check to enforce that functions return to a valid call site. Unlike previous work, our check does not modify call sites. All changes are local to the return instructions that are to be protected. This gives BinKungfu the flexibility to add as many checks as constraints permit, irrespective of the number of call sites.

This work gives an example where we make an existing transformation more elastic, and thereby achieve higher protection. In Section 4.4, we present details of the check and discuss the trade-offs between the elastic and fixed-cost version. We evaluate the performance effects in Section 4.5.

1.3 SUMMARY

Software should be reliable, particularly systems software that is the foundation for everything we do with computers. In this work, we are interested in program transformations that make existing software more reliable. These transformations make software resilient against bugs, faster to test, and easier to verify, at the cost of reduced performance.

We formulate the Elasticity Principle, which postulates that we can flexibly trade some of the benefit of program transformations for increased performance. We present four techniques that make use of the principle to apply transformations selectively, with precise control over their costs and effects. These techniques break transformations down into small atomic parts, and identify those transformation atoms that are cheapest and most useful for the given use case. By applying just the beneficial atoms, our techniques achieve favorable trade-offs between speed and reliability.

2 BACKGROUND AND RELATED WORK

This chapter lays the groundwork for the rest of the thesis. In Section 2.1, we define terms like systems software and reliability, and explain how the C and C++ programming languages set the bar for the performance and reliability of systems software. We then review work that has influenced this thesis, grouped in two categories:

- Techniques to improve performance, e.g., optimizations that reduce CPU and memory usage, in Section 2.2
- Techniques to improve reliability and security, including runtime checks, testing and verification techniques, and approaches for fault tolerance, in Section 2.3

Throughout this chapter, we highlight program transformations that can move software within the performance/reliability space. We will pay particular attention to trade-offs made by such transformations.

2.1 DEFINITIONS AND BACKGROUND

This section defines systems software and reliability, and explains the context in which systems software is built.

2.1.1 *Systems software*

In this thesis, we are concerned with *systems software:* foundational software such as operating systems, compilers, servers, web browsers, data bases, runtime libraries for higher level languages, hypervisors, etc. We now describe systems software more precisely by looking at what these examples have in common.

Systems software is often the basis for higher-level software. For example, consider an application written in a high-level language like Javascript. It relies on three layers of systems software: the Javascript virtual machine, the web browser wherein the virtual machine runs, and the operating system that executes the browser.

Systems software provides mechanisms to ensure the security and reliability of the entire software stack. For example, isolation and access control are crucial to contain malicious websites and prevent them from harming users of a web browser. Browsers are designed to enforce these properties, but also rely on sandboxes and isolation at the operating system level to provide multiple layers of defenses.

This architecture means that the reliability of systems software itself is particularly important, since everything else builds on it.

Systems software requires precise control over hardware resources. At the lowest layers, systems software such as drivers talks to hardware directly, and therefore needs means to communicate at a low level. Further up in the stack, precise control over resources is needed for efficiency. Systems software needs to be fast [83] to be a versatile layer on which higher-level applications can be built.

To summarize, systems software is an important class of software that builds the foundation for other applications, and requires particular focus on reliability and performance.

2.1.2 *Reliability*

The term *reliability* encompasses notions of availability, dependability, trustworthiness, safety and security. We would like systems that we can always depend on and entrust with our valuable data. Systems that will not harm us and are resilient against attacks.

To make these definitions more precise and practical, Section 2.3 will explain properties of reliable systems that are both precisely defined and enforceable. These properties ensure the absence of certain classes of bad behavior, e.g., the absence of data leaks due to invalid memory accesses.

2.1.3 *The C/C++ systems programming languages*

The C and C++ programming languages are the dominant languages used to implement systems software. All major operating systems and web browsers use them; they are the world's second and third most popular programming languages according to the TIOBE index [129]. The design of these languages influences both the performance and the reliability of systems.

C and C++ offer abstractions that give developers both high performance and many responsibilities. Table 1 illustrates this for three language features: memory management, concurrency, and undefined behavior.

These features are at the heart of the performance/reliability tradeoffs. In the coming sections, we will review techniques that ease the programmer's burden and help detect memory errors, concurrency errors, or undefined behavior. The performance of these techniques will be measured against the performance of "raw" C/C++, and is a crucial factor in deciding whether a given technique will find widespread use [124].

Feature	Performance	Requirements
Memory management	Full control over object size, field order, arrays, and object lifetime	Avoid out-of bounds accesses and use-after-free
Concurrency	Full flexibility for sharing data and synchronization	Avoid data races and deadlocks
Undefined behavior	No unnecessary runtime checks	Ensure all operations are valid

Table 1: Language features of C/C++, with their implications for performance and their requirements on developers.

2.1.3.1 *Memory management*

Systems languages give developers precise control over the memory use of their programs. This allows developers to achieve high data locality and reduce indirection, and leads to a small and predictable memory footprint. In exchange, developers must allocate the right amounts of memory for each object, respect object bounds, and deallocate memory when (and only when) it is no longer needed. This is hard: the ANSI C standard [69] rules for memory management are subtle, complicated, and often interpreted differently by compiler vendors and developers [95]. As a result, memory errors are the #1 cause of software vulnerabilities, according to the National Vulnerabilities Database [105].

2.1.3.2 *Concurrency*

With the prevalence of many-core machines, systems software increasingly needs to make use of concurrency. Originally, the C and C++ languages had no notion of concurrency [20]. They gained the capability to use multiple threads through operating system specific libraries like PThreads [24]. These multithreading libraries were not integrated into the language and the compilers, and so they came with rules that programmers had to follow in order to not break assumptions that compilers were making. Recent standards like C11 and C++11 [69, 68] incorporate concurrency and formally specify the result of concurrent accesses to memory. However, the burden to follow synchronization rules remains with the programmers. If programmers violate synchronization rules, programs can behave differently from what the programmer intended [19].

Concurrency thus brings new reliability problems. Not only are concurrent programs harder to reason about, but programmers also need to keep all synchronization rules. This is yet another trade-off between performance and reliability.

2.1.3.3 *Undefined behavior*

Memory safety violations and data races are instances of a more general concept called *undefined behavior* [36, 136, 116, 59]. The idea is that compilers may assume that certain behaviors (such as out-of-bounds memory accesses, data races, divisions by zero, signed integer overflows, etc.) never happen. In other words, compilers trust programmers to use the language correctly.

Undefined behavior enables more efficient code generation. Compilers can generate code to handle the good case only, and do not need to build superfluous checks into the program to handle bad cases such as overflows. Undefined behavior also enables more optimizations. For example, a C compiler may assume that `x+1 > x` is true if `x` is a signed integer, because the statement can only be false if the addition operation causes a signed integer overflow, which is an undefined behavior in C.

Undefined behavior creates yet another tension between performance and reliability, because understanding and avoiding undefined behaviors is hard.

2.2 TECHNIQUES FOR IMPROVING THE PERFORMANCE OF SYSTEMS

In this section, we look at techniques that help systems use hardware most efficiently. For the systems we are concerned with in this thesis—foundational software such as operating systems, compilers, servers, web browsers—efficiency is key. We look at how efficiency can be improved through performance optimizations done by compilers in Section 2.2.1, with a deeper focus on optimizations powered by profiling in Section 2.2.2.

2.2.1 *Optimizing compilers*

The first complete compiler for a high-level language, FORTRAN I, already implemented many transformations to make programs fast. It featured optimizations such as common subexpression elimination, hoisting computations out of loops, and automatic register allocation [11].

Modern compilers perform even more analysis and optimizations. A landmark in this area is LLVM [84]. This compiler framework introduced three ideas. First, LLVM defined an intermediate language to represent programs. Language-specific frontends would translate the source language into LLVM intermediate representation (IR). All optimizations then happen at the IR level, and can thus benefit all languages for which there is a frontend. Second, LLVM was designed to support program transformations at multiple stages in the translation.

In addition to compile-time optimizations, it also allows transformations to happen at link time, when all the components of the program are known and more information on the target architecture is available. Third, LLVM sees the compiler as a framework and makes its transformations easily accessible to third parties. This made it much easier to create tools that analyze and transform programs, and so LLVM was quickly adopted both in academia and industry.

The availability of compilers and program transformations is crucial to this thesis. Three out of the four projects presented here are built on the LLVM compiler and make use of its optimizations.

2.2.2 *Profile-guided optimization*

Modern compilers can use data obtained through profiling to guide their program optimizations [31, 58]. This data contains information about hot (i.e., frequently executed) vs. cold parts of the program, relative frequencies of branches, etc. Compilers use this data to optimize the program layout and improve branch prediction [113], as well as for profitability analysis: Profiling data helps compilers estimate the gains obtained by a program transformation and focus optimizations where they matter. For example, compilers can choose to inline only hot functions, and to unroll or vectorize only hot loops.

Compilers obtain profiling data through instrumentation or sampling. GCov [49] integrates data collection into the program itself, by means of instrumentation code that it adds to each location of interest. This code updates a counter whenever it is executed. When the program finished running, GCov collects the counter values and derives the number of times each location in the program has been executed. Ball and Larus [14] achieve this with lower overhead, by computing a minimal set of counters for a given control-flow graph, and inferring values for those locations without counters. Statistical profilers [94, 86] don't modify a program at all. Instead, they take samples of CPU activity while the program is running, and record which locations are currently being executed. This allows them to reconstruct a statistical distribution of CPU time over locations in the program.

Profile-guided optimization relates to this thesis in two ways. First, its use by compilers shows that many program transformations are trade-offs. For example, inlining, loop unrolling, and loop vectorization all trade speed for increased code size. Compilers need extra information to decide whether this trade-off is worth being made. The same is true for our techniques, which use program transformations in an elastic way that balances speed and reliability. Second, we found that profiling data is a good way to estimate the cost and benefit of a transformation, and so we use profilers in two of our projects.

The idea to focus efforts where they are most needed, and to identify these areas through profiling, appears frequently in the literature. Donald Knuth [79] estimates that 3% of a system really matter for performance, and recommends to optimize just these hot parts. Conversely, reliability issues tend to be located in cold parts of a system. Yuan et al. [140] analyzed catastrophic failures in distributed systems, and found that 30% of them were caused by wrong error-handling code, which is by definition cold. Developers can use methods such as bias-free sampling [74] or adaptive statistical profiling [33] to focus debugging efforts and program analysis on cold code.

2.3 TECHNIQUES FOR RELIABILITY AND SECURITY OF SYSTEMS

In this section, we turn our focus to the reliability of systems. We first divide reliability into concrete properties that we would like our software systems to have in Section 2.3.1. Then, we discuss strategies against reliability problems: detecting errors and preventing their consequences using program instrumentation (Section 2.3.2), and eliminating errors through systematic testing (Section 2.3.3). We also touch on whole-system approaches to achieve reliability in Section 2.3.4.

2.3.1 *Properties of reliable and secure systems*

Reliable systems are those that can be trusted to perform their work correctly. But what does this mean precisely? In the ideal case, the system comes with a specification that formally describes what it should do. In this case, a correct system is one that implements its specification.

In practice, the properties that we expect from systems are often weak, as weak as "it should not crash". Even such weak properties are valuable, because they are applicable to many systems and because existing systems do exhibit violations of these properties.

In-between these two extremes, there are a number of reliability properties. Each of these has proven useful because it offers a unique combination of strength, applicability, and ability to be automatically enforced.

Code contracts and *assertions* are light-weight forms of specifications that hold at specific points in the system. Code contracts [97] specify invariants that the program state must satisfy at interfaces, such as at the start and end of a function call. Assertions are flexible checks that developers can add anywhere in the program to verify that its state is consistent, and abort the program otherwise. These techniques go back to Alan Turing, who used assertions to simplify arguments about a program's correctness [130]. More recently in 2002, C.A.R. Hoare reported that the code for Microsoft Office contains over 250,000 assertions [64].

Type safety [114] is a property that specifies that all operations in a program are consistent with the type of objects they operate on. This is a strong property; it subsumes other properties mentioned below. For example, type safety prevents programs from accessing uninitialized or out-of-bounds memory, and from calling functions on the wrong type of object. A type-safe program will never misinterpret attacker-provided data as code or as internal objects.

The beauty of type safety is that it can be formalized in a type system and enforced. Strongly type-safe languages verify during compilation whether programs satisfy the type system, and reject all programs with risky operations that could compromise type safety. Languages like Java, C#, D, and Go achieve some of their type safety guarantees through automatic memory management. They rely on a garbage collector to keep track of used memory and decide when objects are safe to release. These languages have a larger run-time library and offer less control than languages with manual memory management. On the other hand, languages like Rust provide zero-cost abstractions for type safety, which gives developers a high degree of control but has a steep learning curve.

Memory safety [106, 18, 103, 62] is a subset of type safety ensuring that each operation only accesses memory that it is allowed to use. It can be enforced at a lower level, where one only considers operations that read or write memory. These memory accesses happen through pointers. Memory safety enforcement solutions assign to each pointer a memory region that it may access or point to, and enforce that the program never uses the pointer to touch other areas of memory.

Relaxed memory safety properties [43, 4, 3] specify a weaker form of memory safety that can be enforced more cheaply. One way to do this is to approximate the memory regions that a pointer may access. We give several examples later in Section 2.3.2, when we discuss tools that enforce memory safety. Another way is to enforce safety only for critical memory areas: *Code-pointer integrity* [82] ensures that pointers to executable code (i.e., function pointers and return addresses) cannot be overwritten by mistake. This splits memory into a data area and a control area, with the idea that protecting the control area will prevent catastrophic bugs that would allow an attacker to take complete control over the program.

Control-flow integrity [1, 143, 142, 135, 39, 127] thwarts attackers in a different way: instead of protecting operations that access memory, it protects operations that affect the program's control flow. These include calls to functions, returns from functions, and indirect branches. Control-flow integrity ensures that these operations always have a valid target, one that is allowed according to the static control-flow graph of the program. For example, this prevents calls to the middle of a function, or returns to a location other than the proper function call site. All these control-flow transfer operations are at risk because they use computed target addresses. While memory safety protects the way these addresses are computed, control-flow integrity just checks whether the result looks valid.

Freedom from undefined behavior [116, 59] means that programs follow the rules of their programming language, and avoid any operation that is undefined according to its semantics. Undefined behavior overlaps with many of the categories above (for example, an out-of-bounds memory access is undefined) but also includes other categories of problems, e.g., overflows in arithmetic operations [36] and data races [20].

2.3.2 *Run-time safety checks*

Many reliability properties from the previous section can be checked while the program is running. In fact, some of the properties, e.g., relaxed memory safety, have been invented because they offer a particularly sweet spot for run-time checks.

Run-time checking techniques face different trade-offs depending on their goals. One one hand, techniques for use during testing and bug finding focus on compatibility and the ability to detect as many problems as possible (Section 2.3.2.1). On the other hand, techniques to protect programs against the consequences of errors focus on low overheads (Section 2.3.2.2). In what follows, we review techniques from both categories.

We then look at automated ways to adapt a technique's cost and effect, in Section 2.3.2.3. These adaptation techniques are at the heart of all work related to this thesis. They share our goal of leveraging existing run-time checking tools, and find ways to do so in situations that require low overhead.

2.3.2.1 *Checks for bug finding and testing*

Run-time checks are popular for finding low-level reliability issues. Some of the earliest tools like BCC [76], `rtcc` [122], SafeC [8], and CCured [106] targeted memory safety errors. Compatibility was a major challenge for these tools, because they changed the representation of pointers in memory in order to track which memory area a pointer could access. Later tools like the Jones and Kelly bounds checker [72], Valgrind [120, 107], SoftBound CETS [104, 103] and Intel MPX [67, 110] store pointer metadata in a separate data structure, which improves compatibility but incurs higher overhead. Deputy [38] completely eliminates the need for metadata and instead relies on annotations provided by the developer. This achieves compatibility at high performance, but reduces the ease of use of the tool.

Besides memory safety, there are tools to automatically check all memory accesses for data races [119, 91, 75], and to check operations for overflows and other types of undefined behavior [36].

These tools share with our work the desire to improve the reliability of existing software. Our work, particularly ASAP, builds on such tools and adapts them, so that they can be used in more situations than just bug finding and testing.

2.3.2.2 *Checks as protection mechanisms*

To use run-time checks as protection mechanisms for production systems, the problem of overhead must be solved.

One approach to reduce overhead is to provide weaker guarantees: Baggy Bounds Checking [4] uses a memory allocator that rounds object sizes up to the next power of two. This allows for faster checks that require less metadata, but also means that overflows can only be detected if they exceed the object's padding. SAFECode [43] and WIT [3] statically partition program memory into sets of disjoint objects, and can detect erroneous memory accesses if the target is in the wrong set. In exchange for weaker guarantees, these approaches reduce overhead from 70%-116% (MPX, SoftBound) to below 10%.

AddressSanitizer [118] provides approximate memory safety by detecting invalid accesses to "poisoned" memory zones. Its allocator adds such zones between objects, and also marks recently freed memory areas as poisoned. This approach might miss some invalid accesses, but is highly compatible and has a lower time/memory overhead than complete memory safety solutions.

Another approach to achieve a better trade-off between performance and reliability consists in focusing protection where it matters most. Code-Pointer Integrity (CPI) [82] is a technique that provides memory safety, but only for memory locations that, directly or indirectly, lead to code pointers. These locations are particularly sensitive, because overwriting them can arbitrarily change a program's control-flow. Yet only about 6.5% of all memory accesses target these locations and need to be checked by CPI instrumentation. DataShield [29] uses a similar idea, but lets the programmer annotate sensitive data types in her programs, and limits protection to these types.

Focusing the scope further leads to even higher performance. The authors of CPI describe a variant called Code-Pointer Separation. It protects only code pointers themselves, but does not extend the protection to those locations that indirectly lead to code pointers, for a 4.3× reduction in overhead compared to CPI. The SafeStack variant further limits protection to the stack, where it separates safe and sensitive values from unsafe buffers. This protects against attacks like return-oriented programming at zero overhead, because the cost for maintaining the separation is offset by better data locality.

Another way to achieve protection at low overhead is to allow memory errors to happen, but prevent them from being exploited. Diehard [18] and ASLR [126] rely on randomization to prevent attackers from guessing object locations in memory. Control-Flow Integrity (CFI) [1, 143, 142, 135, 39, 127] prevents control-flow transfers if the target address has been corrupted due to a memory error.

The large number of techniques show that trade-offs are ubiquitous. Szekeres et al. [124] explore this trade-off and find that whether a technique gains adoption depends on its protection, compatibility, and overhead. The techniques in this section and our elastic program transformations share the goal of balancing protection and overhead. In contrast to the specialized solutions presented here, our work does so in a generic way, exploiting the elasticity that is present in many existing program transformations.

2.3.2.3 *Controlling the trade-offs for run-time checks*

We now describe manual and automated solutions to gain more control over the trade-offs faced by run-time checking techniques. These share the same goal as our elastic instrumentation techniques, and are closely related to our work.

Several of the tools presented so far allow developers to specify parts of the program that should not be instrumented, e.g., using a blacklist [118, 119, 36]. These approaches are relatively coarse-grained, and usually enable/disable instrumentation for entire files or functions.

Automatic approaches based on sampling are more fine-grained and allow for a variety of sampling strategies. Arnold and Ryder [6] switch between instrumented and non-instrumented versions of the same code such that each executed basic block has the same probability of running under instrumentation. Bursty Tracing and its extensions [63, 33] can instrument long execution traces and focus instrumentation on infrequently executed code. LiteRace [91] also focuses instrumentation in cold code, and like us exploits the Pareto Principle. Its authors find that monitoring 2% of all memory accesses is sufficient to detect 70% of the data races that could have been detected with full monitoring.

We know of one project that uses profiling to guide selective instrumentation. The Multicompiler [65] transforms programs to increase software diversity, and modifies only cold program areas that are not critical to performance.

All previous approaches reduce overhead by sacrificing instrumentation coverage. In contrast, RaceMob [75] achieves full coverage at low overhead through crowdsourcing. Each crowd participant receives a selectively instrumented program with sufficiently low overhead to enable in-production use.

This thesis presents several techniques that generalize these ideas and allow trade-offs to be optimized fully automatically, for off-the-shelf program transformation tools, in a fine-grained way, and informed by profiling data.

2.3.2.4 *Summary of run-time checking techniques*

Run-time safety checks can enforce a large number of reliability properties. Many techniques add such checks automatically to software. These techniques need to satisfy high demands for compatibility, performance, and safety guarantees, and no existing technique excels in all these areas. Compared to the number of techniques that add run-time checks, there is relatively little work to automatically adjust their compatibility, performance, and safety.

2.3.3 *Systematic testing and verification*

Techniques to systematically test and verify software aim to eliminate reliability problems before the software is being used in production. In our work, we find that program transformations can make these techniques faster. This is particularly true for automated approaches like fuzz testing and symbolic execution; those are CPU-intensive tasks, the performance of which depends on the structure of the program under test. In this section, we review such techniques and how our work relates to existing methods for faster testing and verification.

2.3.3.1 *Fuzz testing*

Fuzz testing is a systematic way to test programs, using quasi-random inputs. The origins of fuzzing go back to a 1988 idea of Miller. On a "dark and stormy night" when static discharges randomly flipped bits on a modem line, Miller observed that these random modifications caused UNIX utilities to crash. He proposed a student project to systematically explore random input mutations, and discovered several bugs in core systems programs [100, 99].

Nowadays, many fuzz testing techniques make heavy use of instrumentation to observe the software under test and generate high-quality inputs. The technique was first described by Ormandy [112]. He used coverage instrumentation to minimize a corpus of test inputs. He obtained these inputs from crawling the Internet, but kept only files that triggered previously unseen code. In a second stage, he generated test inputs by randomly mutating elements from this corpus.

This line of feedback-driven fuzzers continues today with tools such as AFL [141] and LibFuzzer [88]. Feedback-driven fuzzing is also frequently used to find security-relevant bugs in closed-source binaries. To apply coverage instrumentation, AFL-DynInst [109] and Murphy use static binary rewriting, whereas AFL-QEMU [57] and GRR [53] use dynamic binary translation.

Thanks to coverage instrumentation, fuzzers can find inputs that reach deep parts of the software under test. This comes at the expense of throughput, i.e., fuzzers can test fewer inputs. This kind of trade-off between speed and test quality is where elastic instrumentation shines; we explore this scenario in Section 4.3.

2.3.3.2 *Symbolic execution*

Symbolic execution [77, 50, 27, 26, 23, 34, 21, 30, 121] has the same goal as fuzz testing, namely to generate inputs which explore as much of a program under test as possible. However, symbolic execution analyzes the program much more deeply than fuzzers.

The idea is to track precisely how input flows through a program. Normal, non-symbolic, execution uses concrete data as input (e.g., `42`), processes it, and stores concrete computation results in memory (e.g., `y=21`). In symbolic execution, the program is interpreted by a symbolic execution engine that treats inputs as symbols (e.g., λ), and computation results as symbolic expressions (e.g., `y=`$\lambda/2$).

A symbolic execution engine can solve symbolic expressions to find interesting inputs. Consider a program statement like `if (y == 65) {...} else {...}`. A symbolic execution engine can generate the formula $\lambda/2 = 65$, and use a constraint solver to find inputs for each outcome of the branch.

By repeating this process for every branch, symbolic execution to enumerate all paths through the program, and explore each path in turn. In contrast to fuzz testing, where each randomly generated input has only a small chance to trigger new program behavior, each successful invocation of the constraint solver discovers a new path through the program.

The effectiveness of symbolic execution depends on the structure of the program, the number of paths through it, and the complexity of the expressions computed by the program. Program transformations affect all these aspects. In Chapter 3, we make use of program transformations to speed up symbolic execution.

2.3.3.3 *Static checking*

Static checking [37, 61, 12, 96, 45, 10], like symbolic execution, reasons symbolically about the computations performed by a program, the possible values of program variables, the reachability of a given program location, etc. The main difference between the approaches is that static checking does not reason about individual paths through a program, but considers all paths that could lead to a location of interest together.

Several approaches use aspects from both techniques. For example, Godefroid [51] uses function summaries, i.e., formulas that compactly represent all paths through a function, within a symbolic execution engine. Kuznetsov et al. [81] describe how symbolic execution engines can merge multiple paths together to analyze them as a single unit, and give a heuristic to decide when this would be profitable. Veritesting [9] detects during symbolic execution program parts that are easy to reason about as a whole, and switches from symbolic execution mode to static checking mode for these parts.

All these approaches try to reason about the program in an efficient way. Program transformations that affect program structure can make this easier or harder. Chapter 3 gives many examples of this. Similarly, adding or removing safety checks from a program affects how many reliability properties these approaches can verify, and how difficult that task is.

2.3.3.4 *Formal verification*

The strongest formally verified software is built from the ground up with verification in mind. Developers of projects like seL4 [78], CompCert [85], and IronClad [60] write specifications, code, and proofs that the code implements the specification. This approach achieves strong reliability guarantees and, in addition, makes formal statements about the functionality of systems. In contrast, our work targets existing software, but can at most verify the absence of certain classes of bugs or failures.

Verified software is proven to be correct at a given abstraction level, like source code or assembly code. Projects that verify source code (seL4, CompCert) then rely on the correctness of compilers to preserve the guarantees when transforming the program into an executable binary. Fortunately, compilers and their transformations are increasingly verified themselves: CompCert is formally proven to correctly translate a large subset of C into assembly code, and ALIVE [89] is a technique to prove the correctness of peephole transformations used by the LLVM compiler.

2.3.4 *Isolation, redundancy, defense in depth*

We have so far described reliability in terms of low-level properties, such as the absence of memory-related bugs. This is the kind of reliability most related to our work, because it can be increased through program transformations. Yet, when designing large software systems, there are many higher-level properties that influence reliability.

Consider modularity and isolation. These are primarily achieved through good design. Techniques from operating systems, compilers and program transformations can be building blocks for such a design. For example, web browsers use process-level isolation to separate website rendering from core browser tasks [17], the Singularity research operating system uses type systems to isolate processes [66], and program transformations can isolate different program parts within a process [134, 139]. Designers need to carefully choose how much of these techniques to use, because there is a trade-off between isolation granularity and performance. For example, performance is the reason why the Chromium browser shares a single renderer process among multiple websites [71].

Redundancy is another general approach to improve a system's reliability in the presence of unpredictable and uncorrelated failures. Like modularity, it is typically handled at a higher design level, and performed manually rather than using automatic transformations. Redundancy also faces trade-offs between reliability and performance, e.g., when choosing the number of replicas and the consistency model [42]. In addition to high-level redundancy, designers can use automatic program transformations to obtain redundancy at a low level, e.g., replicating individual assembly instructions to mask transient CPU errors [80].

Defense in depth means adding multiple layers of defense against reliability problems, in order to be protected even if one technique fails. For example, developers might harden a vulnerable server program using program transformations, *and* run it in a restricted sandbox. This is a good idea in general, because it increases the difficulty of compromising a system. For example, the exploit that controlled Chrome OS and won the Pwnium 4 competition required no less than five different bugs in various parts of the Chrome browser and operating system [35]. Defense in depth is also a good idea when program transformations are used as a defense layer, because program hardening mechanisms are often approximate due to overhead or compatibility concerns [124].

2.4 SUMMARY

This chapter started by defining systems software and providing background information on the program languages using which systems are built. We also defined reliability in terms of properties we would like our systems to have: correctness with respect to specifications, assertions, and code contracts, and freedom from memory errors, data races, illegal control flow and undefined behavior.

Throughout this chapter, the trade-off between reliability and performance appeared frequently. We saw it in systems programming languages, where features like low-level memory management and concurrency are necessary for performance but problematic for reliability. The trade-off also appeared when using run-time checks to increase reliability, where the techniques with the strongest guarantees have the highest overhead. We found the same trade-off when we presented fuzz testing, symbolic execution, and static checking: Those techniques that reveal the most reliability problems are also the most CPU-intensive.

We identified program transformations both for improving performance and reliability, as well as transformations used during testing and verification. So far, relatively little work exists that makes use of the elasticity of these transformations to tailor them for a given use-case. Existing techniques are based on either sampling or profiling. These techniques are the ones most closely related to the work in this thesis.

Part II

TRANSFORMING SOFTWARE FOR VERIFICATION, SPEED, AND RELIABILITY

Where we look at three use cases for automatic program transformations: making software systems easier to verify, protect systems from vulnerabilities while satisfying performance constraints, and smarter fuzz testing. In each case, we explain the trade-off at hand, present the design and implementation of program transformation to turn this trade-off in our favor, and evaluate its effect on the program.

3 -OVERIFY: SELECTING PROGRAM TRANSFORMATIONS FOR FAST VERIFICATION

3.1 DESIRED PROPERTY: FAST VERIFICATION

Automated program verification tools are essential to writing good quality software. They find bugs and are crucial in safety-critical workflows. For example, for a bounded input size, symbolic execution engines like KLEE [26] and Cloud9 [23] can verify the absence of bugs in systems software like Coreutils, Apache, and Memcached; the SAGE [21] tool found many security vulnerabilities in Microsoft Office and Windows. Unfortunately, verification tools are not used widely in practice, because they are considered too slow and imprecise.

Many verification tools, including the ones mentioned above, verify *compiled* programs. The observation underlying the work presented in this chapter is that verification complexity could be significantly reduced if only a program was compiled specifically for verification. Building on this point, we argue that compilers should have an -OVERIFY option, which optimizes for fast and precise verification of the compiled code.

In the following sections, we first show how the requirements of verification differ from those of fast execution on a CPU. The differences arise because verification is complete: verification tools consider all possible executions, all possible variable values, or all feasible paths through a program. For the purpose of this work, we consider tools that completely explore a program to be verification tools. This includes static analyzers, model checkers, tools based on abstract interpretation, and exhaustive symbolic execution (see Section 2.3.3 for a description of these techniques).

We then identify program transformations that reduce verification complexity, and others that increase it. We also present ideas for new transformations that, although not commonly performed by today's compilers, would be of great benefit to verification tools. These transformations have two types of effects: they simplify program operations, and they reduce various program features that cause verification complexity, such as function calls, loops, or branches. We believe that a wide range of verification tools can benefit from these transformations because they affect many aspects of a program.

Finally, we present the design and evaluation of our -OSYMBEX prototype. It generates programs optimized for a specific type of verifi-

```
int wc(unsigned char *str, int any) {
  int res = 0;
  int new_word = 1;

  for (unsigned char *p = str; *p; ++p) {
    if (isspace(*p) || (any && !isalpha(*p))) {
      new_word = 1;
    } else {
      if (new_word) {
        ++res;
        new_word = 0;
      }
    }
  }

  return res;
}
```

Figure 4: Count words in a string; they are separated by whitespace or, if `any != 0`, by non-alphabetic characters.

cation, namely symbolic execution, by applying those program transformations that we identified as profitable for this use case.

-Overify is a step toward enabling wide-spread use of verification tools: it gets developers the verification results faster, is easy to integrate into existing build chains and, since it uses time-tested compiler technology, it generates reliable verification results. Moreover, -Overify shows that program transformations are powerful tools that are applicable to more use cases than we perhaps thought.

3.1.1 *Motivating example*

The example in Figure 4, a function that counts words in a string, illustrates the effect of compiler transformations on program analysis.

This simple function is challenging to analyze for several reasons: It contains an input-dependent loop with an unpredictable number of iterations. Inside the loop, the control flow branches on the type of the current input character. This leads to an explosion in the number of possible paths: there are $O(3^{\text{length}(str)})$ paths through this function. On top of that, `wc` calls the external library functions `isspace` and `isalpha`; their implementations further complicate the analysis of `wc`.

We exhaustively tested all paths through `wc` for inputs up to 10 characters long using KLEE [26], and Table 2 shows the results for four different compiler settings. Aggressive optimizations can dramatically reduce verification time. At level -O2, the reduction comes mostly from algebraic simplifications and removal of redundant operations, which lead to a reduced number of instructions to interpret

Optimization	-O0	-O2	-O3	-OVERIFY
t_{verify} [msec]	13,126	8,079	**736**	**49**
t_{compile} [msec]	38	42	43	44
t_{run} [msec]	3,318	704	694	1,827
# instructions	896,853	480,229	37,829	312
# paths	30,537	30,537	2,045	11

Table 2: Using symbolic execution to exhaustively explore all paths in the code of Figure 4 for strings of up to 10 characters: time to verify (t_{verify}), time to compile (t_{compile}), and time to run on a text with 10^8 words (t_{run}).

```
int sp = isspace(*p) != 0;
sp |= (any != 0) & !isalpha(*p);
res += ~sp & new_word;
new_word = sp;
```

Figure 5: Branch-free version of wc's loop body.

in KLEE. The number of explored paths remains the same as for -O0, indicating that -O2 does not fundamentally change the program's structure.

At level -O3, the compiler unswitches the loop: The loop-invariant condition `any != 0` is moved out of the loop, and simplified copies of the loop body are emitted for `any == 0` and `any != 0`, respectively. This enables algebraic simplification of the branch condition, which in turn reduces the number of paths to $O(2^{\text{length}(str)})$ and thus reduces verification time. The price for this reduction is an increase in the size of the compiled program, due to the duplicated loop body.

Compiling the program using -OVERIFY goes beyond the optimizations performed by -O3, and results in the code in Figure 5, which contains no more branches. This reduces the number of paths to $O(\text{length}(str))$, and symbolic execution time decreases by 15×.

A traditional compiler would not perform this optimization, because the cost of executing a branch on a CPU is small. It is cheaper to perform a branch that skips some instructions than to unconditionally execute the entire loop body. Indeed, when actually executed, the branch-free version takes 2.5× as long as the -O3 branching version (Table 2). This illustrates the conflicting requirements of fast execution and fast verification.

3.1.2 *How verification differs from execution on a CPU*

In a nutshell, verification reasons about the semantics of *all* possible program executions, whereas a CPU is concerned about running typical executions as quickly as possible.

When transforming programs for fast execution, it makes sometimes sense to add complexity. For example, a program may distinguish fast from slow paths if this speeds up the average case. Similarly, re-ordering operations or re-arranging objects in memory can lead to better cache behavior.

For fast execution, it is OK to obfuscate semantics, drop high-level types, etc. For example, batching might combine unrelated memory operations into a single wide load or store. Similarly, using a precomputed table can replace a costly operation by a memory access, but this makes it harder to reason about that operation's semantics.

Verification is different. The *time* to verify a program is dominated by the number of branches it has, the overall number of loop iterations, memory accesses, and various arithmetic artifacts. The *precision* of the analysis can also depend on the program structure, e.g., on the number of control-flow join points or the kinds of operations the program performs. Verification tools can often exploit high-level knowledge of the program, like information about types of variables or about the program's use of the standard library.

Program transformations affect all these aspects. Thus, compilers can make programs more verification-friendly using program transformations described in Section 3.2.

3.2 PROGRAM TRANSFORMATIONS THAT AFFECT VERIFICATION

While a compiled program must be semantically equivalent to its source code representation, a compiler still has a lot of room for optimizations. In Table 3 we show the various options a typical compiler could offer today to -OVERIFY, as well as options it *does not* offer.

We now give a few examples of the large body of possible program transformations and illustrate the effect they have on verification complexity.

3.2.1 *Simplifying computation: arithmetic simplifications*

Standard simplifications, such as *copy propagation* and *constant folding*, are good for execution speed, but can be even better for verification. Consider a tool that reasons about value ranges for variables: When encountering code such as `x = input(); y = x; x -= y;` the tool might think that `x` could have any value. Yet standard simplifications can turn this code into the equivalent but easier version `input(); x = 0`.

Transformation	Verification	Execution	Available
Arithmetic simplifications	+	+	✓
Remove/split memory accesses	+	+	✓
Simplify control flow	+	+/−	✓
Function inlining, loop unrolling	+/−	+/−	✓
Improve cache behavior etc.	−	+	✓
Program annotations	+	−	few
Generate runtime checks	+	−	some

Table 3: Compiler optimizations and their impact (positive +, or negative −) on *Verification* time and/or *Execution* time. The last column shows to what degree the optimizations are readily available in today's compilers.

3.2.2 *Simplifying memory accesses*

Memory accesses complicate the data-flow graph of a program, requiring verification tools to analyze which accesses may correspond to the same memory location vs. which cannot (alias analysis). The complexity of this analysis typically grows exponentially with the number of related memory accesses.

A compiler can easily help by converting values that reside in memory to register values, and by splitting large objects into independent smaller objects, thereby reducing the opportunities for memory access aliasing. On the other hand, transformations that group unrelated accesses (e.g., packing a structure into a single wide integer) may hide semantics and introduce additional dependencies that make programs harder to analyze.

3.2.3 *Simplifying control-flow: if-conversion, loop unswitching, function inlining*

Program verification often becomes drastically easier if the program's control flow is simplified by a compiler. An optimization called *jump threading* checks whether a conditional branch jumps to a location where another condition is subsumed by the first one; if yes, the first branch is redirected correspondingly, turning two jumps into one. Another example is *loop unswitching*, as seen in Section 3.1.1.

These are especially important for verification tools that reason about multiple execution paths through a program (either individually or grouped in some way). For such tools, the complexity of verifying a program depends on the number of possible execution paths through it, which in turn grows exponentially with the number of conditional branches and the number of possible loop iterations.

As a result, branches and loops have a much higher cost for verification than for normal execution.

Control flow can be further simplified by transforming conditionally executed side-effect-free statements into speculative branch-free versions. This transformation is called *if-conversion*. Compilers perform it when saving one branch instruction outweighs the cost of speculation (e.g., GCC converts `if (test) x = 0;` into `x &= --(test == 0);`). When using -Overify, this simplification is pursued more aggressively, because the cost of a branch is higher.

Simplifying control-flow can increase the precision of verification tools based on abstract interpretation. These tools reason about programs at the granularity of program locations. If multiple paths lead to one location, merging the path information can lose precision. Unswitching a loop can improve precision, because such information merging now occurs only once, after the loop, instead of after every iteration.

Transformations that re-structure the program often open up additional opportunities for simplification. For example, *function inlining* replaces a function invocation by a copy of that function's body. The compiler can now specialize this copy for the particular context where the function was inlined. *Loop unrolling* has a similar effect.

Function inlining can also increase verification precision in another way. It adds context-sensitivity to verification tools that might otherwise be intraprocedural. Similarly, both loop unrolling and function inlining increase the number of program locations. That way, verification tools can analyze individual loop iterations or function instances separately, with increased precision.

3.2.4 *Caching and CPU-specific optimizations*

To generate code that executes fast, compilers must optimize for the target CPU's cache structure and pipeline: keep loops small (to avoid instruction cache misses), pad objects (to keep them aligned in memory), and reorder instructions (to reduce pipeline stalls and to improve branch prediction).

Modern CPUs offer vectorized instructions, which can apply an operation to multiple data values in parallel. Exploiting these instructions speeds up programs but complicates program structure. For example, when the compiler vectorizes a loop, it usually creates two sub-loops: one that processes data in vector-sized chunks and one that handles the remainder of the data when the data size is not a multiple of the CPU's vector size.

All these issues are irrelevant to many classes of verification tools, and some of these optimizations can even slow down verification. Thus, they are omitted under the -Overify switch. This offers the further benefit of considerably more freedom in generating code.

```
inline int isspace(int c) {
  return (__ctype_ptr__[c+1] & _S);
}
```

(a) A typical implementation of isspace function in a standard C library. __ctype_ptr__ is a constant array containing character type flags for each character.

```
inline int isspace(int c) {
  assert((unsigned)c < 256);
  return (c == ' ') | ((unsigned)(c - 0x09) < 5);
}
```

(b) isspace function in the C standard library for -Overify. The absence of memory accesses simplifies program analysis.

Figure 6: Two variants of isspace with different verification complexity

3.2.5 *Simplifying semantics: verification-friendly standard library functions*

For programs that use the C/C++ standard library, the analysis effort depends significantly on the complexity of library functions. This is why some tools, such as KLEE and KLOVER [87], ship with a custom version of the C/C++ standard library.

As part of -Overify, we sought to develop a version of libC that is tailored to the needs of program analysis in general, and thus reusable for many tools. This library provides versions of the standard functions designed to minimize analysis costs. These simplifications entail high-level reasoning and semantic understanding of the code that is beyond what modern compilers can do automatically.

For example, consider the isspace function shown in Figure 6, which checks whether its argument is a space-like character. The common implementation of that function uses a lookup table (shared between multiple similar functions). We found that in most cases, the version of isspace shown in Figure 6b is easier for symbolic execution tools to handle, as it avoids a (potentially) input-dependent memory access to a large array.

Functions in our verification-friendly C library contain run-time checks to verify their preconditions. Such checks are often absent or disabled in production code. For testing and verification, these checks improve the tools' ability to find bugs, and they also lead to better error reports, because bugs are found closer to their root cause.

3.2.6 *Preserving information: annotations and run-time checks*

The output of today's compilers does not preserve all information present in the source code of the program, such as high level types or the separation of a program into modules. Compilers also do not keep information computed during compilation, such as alias information, variable ranges, loop invariants, or trip counts. This information however is priceless for verification tools, and could be easily preserved in the form of program metadata. Some of these are available today, e.g., the Clang compiler [125] can annotate memory accesses with types.

Similarly, compilers know the exact semantics of the program in its source language. For example, an integer overflow is a perfectly well-defined operation in the X86 ISA. Whether it is well-defined in the original program depends on the data type of the overflowing variable.

Compilers can make that semantic information available to verification tools through run-time checks, which transform illegal behavior into crashes. Recent versions of Clang and GCC can emit run-time checks for various forms of illegal behavior, such as overflows, memory corruption, data races, or the use of uninitialized data. These checks empower verification tools and make verification simpler, as tools now only need to check for one type of failure (i.e., crashes).

3.3 DESIGN AND IMPLEMENTATION OF -OVERIFY

We designed -Overify as a compiler switch that enables program transformations for fast verification. -Overify modifies the compilation process in four complementary ways: (1) it selects a set of compiler passes suitable for verification tools, and it inhibits compiler passes that would increase verification complexity; (2) it adjusts cost values and parameters (such as the maximum size of a function to inline) to optimize compilation for fast verification, not fast execution; (3) it causes more metadata to be preserved in the program; and (4) it links the program with a specialized version of the C standard library optimized for verification.

We next motivate this design and discuss its trade-offs, show how developers use it, and present our prototype implementation.

3.3.1 -Overify *belongs in the compiler*

Why perform verification optimizations inside compilers? Why not let every verification tool transform source code using its own specialized transformations?

Making -Overify part of a compiler has three key advantages: First, compilers are in a unique position in the build chain. They have access to the most high-level form of a program (its source code) and control all steps until the program is in the low-level intermediate form that is to be analyzed.

Second, the program needs to be compiled anyway, and so -Overify gains access to a wealth of information, like call graphs and alias analysis results, at no additional cost.

Third, with -Overify, verification tools need not be aware of complex build chains, and need not re-implement common transformations.

We do not advocate a monolithic approach in which compilers and verifiers are tightly coupled. Instead, we want to put to more effective use the services that a compiler offers, and reduce the amount of information lost during compilation.

3.3.2 *Risks and generality of* -Overify

Verifying a verification-optimized version of the program and then shipping a performance-optimized version means that end-users get not exactly what was tested and verified. However, -Overify relies on compiler transformations that are anyway used (perhaps differently) in the rest of the build chain, and this offers confidence in the equivalence of the verified and shipped versions. Moreover, these transformations are themselves extensively tested [138] and even partially verified [89].

Programs that contain undefined behavior face the largest risk. For example, a function that returns the address of a local variable might behave differently when the function is inlined. Compilers can often, but not always, detect such cases and warn the developers.

One could argue that each verification tool requires its own specialized version of -Overify. Yet the *idea* behind -Overify generalizes, and we expect that developers of verification tools can readily decide—intuitively and based on our examples—which optimizations would be advantageous. Compilers can help them by providing access to built-in heuristics (e.g., to decide when a function should be inlined), as well as heuristics specialized for -Overify (e.g., heuristics for estimating when speculative execution would reduce analysis time [81]).

These considerations open new questions for the systems community, spanning the entire process from software development to verification: At what levels should verification be performed? What abstractions should compilers export, so that clients can express their needs and customize the build process? What data should be preserved across the different program transformation steps, and in what format? How can we guarantee the correctness of program transformations?

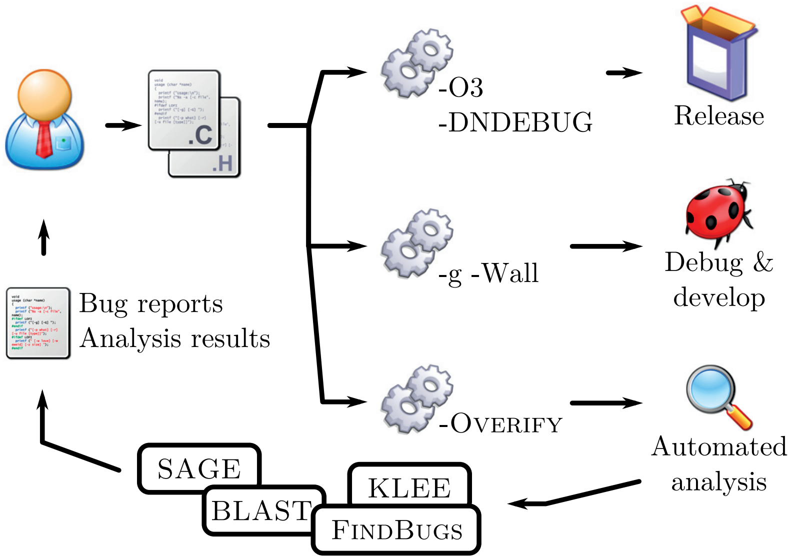


Figure 7: Adding -OVERIFY to an existing build chain, to enable fast automated program analysis and testing.

3.3.3 *Using* -OVERIFY *in practice*

Developers usually create different build configurations for software systems. During development, a program is compiled with debug information, assertions, and possibly reduced optimization to aid testing and debugging. At release time, the program is compiled with the highest optimization level. Our proposed -OVERIFY option adds a third build configuration, aimed at automated testing and verification. This process is illustrated in Figure 7.

An -OVERIFY-enabled compiler can be directly leveraged by a number of program analysis tools. We built a prototype that can generate special binaries optimized for analysis using the S2E system [34] or SAGE tool [52], which perform symbolic execution on x86 binaries. Alternatively, it can generate LLVM IR bitcode optimized for analysis by tools like KLEE and its descendants [26, 87, 23]. We expect our ideas to apply broadly to many other tools, such as FindBugs for Java [47], or Microsoft Pex [128] for .NET.

-OVERIFY makes it possible to use verification-specific optimizations with minimal changes to a build chain, lowering the entry barrier to the use of verification tools available today. We envision future project creation wizards and build systems creating -OVERIFY configurations by default, thus encouraging wide adoption of powerful verification tools, which in turn can help build better software and improve developer productivity.

3.3.4 *Prototype*

We implemented a prototype of -Overify, called -Osymbex, that makes verification easier for symbolic execution tools like SAGE, KLEE, and others. These tools follow the approach outlined in Section 2.3.3.2. They analyze programs one path at a time, treating program inputs as *symbolic*, i.e., they assume inputs can have any value (up to a bounded size). As it interprets the program, a symbolic execution tool keeps track of all the symbolic expressions computed by the program. At conditional branches, the tool invokes a constraint solver to check whether the symbolic condition could be true, false, or both. In the latter case, the tool explores both paths independently, adding the branch condition (or its negation, respectively) as a constraint on the inputs for the current path.

The performance of symbolic execution tools is determined by the number of paths to explore and by the complexity of input-dependent branch conditions. Our prototype -Osymbex reduces both, thereby improving the performance of symbolic execution tools without requiring the tools themselves to be modified.

We chose symbolic execution because of its importance and popularity, which means that tools and benchmarks to evaluate our prototype are readily available. A prototype for a different verification technique would need to choose a different set of program transformations, according to the factors that drive verification cost for the target technique. Note also that while -Osymbex speeds up symbolic execution in general, we may only consider a program to be soundly verified if symbolic execution terminates, i.e., exhaustively explores all the paths through the program. This is in general only possible for a bounded (and typically small) input size.

We built -Osymbex on top of the LLVM compiler infrastructure. Compared to -O3, -Osymbex: (1) considers the cost of a branch to be higher than on a CPU, to avoid branches through speculative execution and loop unswitching; (2) removes loops from the program whenever possible, even if this increases the program size; and (3) aggressively inlines functions in order to benefit from simplifications due to function specialization.

3.4 evaluation of -overify

We evaluated our prototype -Osymbex on real systems code: we ran it on the Coreutils 6.10 suite of UNIX utilities, in essence repeating the case-study from [26].

Optimization	-O0	-O3	-OSYMBEX
jumps threaded	0	7,678	65,618
loops unswitched	0	377	3,022
branches converted	0	959	5,405
functions inlined	0	7,746	16,505
loops unrolled	0	1,615	3,299

Table 4: Compiling Coreutils with different options.

3.4.1 *Static impact: how* -OVERIFY *affects program structure*

Table 4 shows how -OSYMBEX affects the number of program transformations performed by the compiler. We obtained this data by compiling Coreutils and counting the number of times each transformation was performed. The data shows that compilers do transform programs selectively. Even compared to the compiler's strongest optimization level, -OSYMBEX increases the amount of transformations that are beneficial for symbolic execution by almost an order of magnitude.

3.4.2 *Dynamic impact: how* -OVERIFY *affects verification times*

Figure 8 shows the effect -OSYMBEX has on the verification of Coreutils. For each of the 93 tested programs, we measured how long it takes to compile and analyze all paths with KLEE, using 2 to 10 bytes of symbolic input. We did this with -O0, -O3, and -OSYMBEX, respectively. We kept all experiments where KLEE terminates within one hour on at least one of the three versions. This set includes experiments from 79 out of 93 tested programs.

On average, -OSYMBEX reduces overall compilation and analysis time by 58% compared to -O3, and by 63% over -O0. The maximum benefit is a 95× reduction in total time (right side of Figure 8). The verification of 6 programs runs out of time with -O3 (and 11 with -O0), but completes with -OSYMBEX. In a few cases, -O3 outperforms -OSYMBEX, because it takes longer to compile with -OSYMBEX than -O3; this effect vanishes in longer experiments.

The reductions in symbolic execution time are due to exploring fewer paths and spending less time per path. This follows from our evaluation of KLEE statistics, for those experiments where both the -OSYMBEX and -O3 versions terminate; we could not obtain statistics for experiments that timed out. In total, -OSYMBEX reduces the number of explored paths by 57% compared to -O3.

Figure 9 plots the reduction in explored paths and symbolic execution time for each experiment individually. Each dot represents an

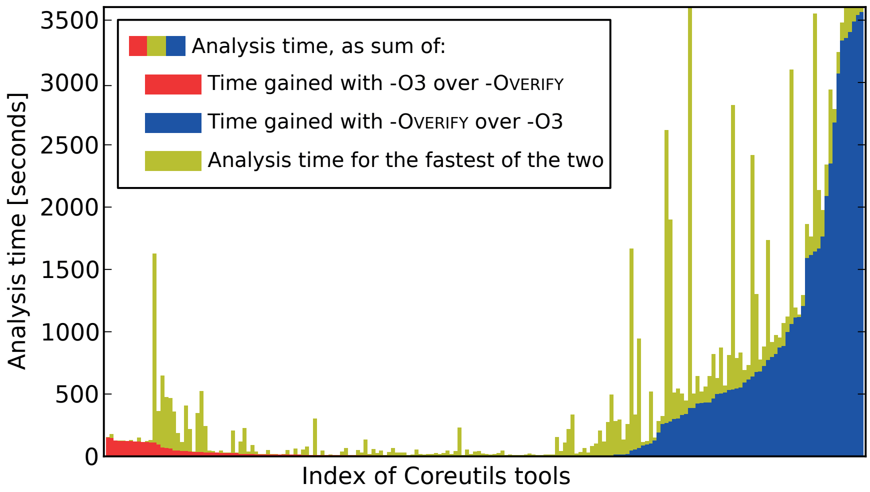


Figure 8: Time to compile and test Coreutils; each bar represents one experiment. Blue (sorted on the right) represents time gained by using -OVERIFY over -O3; red (on the left) shows when -O3 is faster than -OVERIFY; yellow shows the time of whichever one is fastest.

experiment and is positioned according to -OSYMBEX's effect on symbolic execution time and on the number of explored paths, relative to -O3. The points are close to the diagonal, indicating that the reduction in execution time is correlated to the reduction in explored paths. However, many long-running experiments are below the diagonal, which indicates that -OSYMBEX also reduces the time spent *per path*. For 41 out of 79 programs, -OSYMBEX does not affect the number of paths. These experiments correspond to the points at $x = 1.0$ in Figure 9. Some of these experiments nevertheless terminate faster with -OSYMBEX because the simplifications performed by -OSYMBEX reduce the amount of per-path work. There are also experiments for which -OSYMBEX increases symbolic execution time; most of these terminate quickly, e.g., due to small input sizes. Thus, their influence on the overall symbolic execution time is small.

We verified that indeed all bugs discovered by KLEE with -O0 and -O3 are also found with -OSYMBEX.

3.5 SUMMARY

-OVERIFY reveals the power of program transformations. It uses compiler optimizations to transform programs into a form that is semantically equivalent, but easier for verification tools to handle. -OVERIFY inhibits optimizations that make verification harder, adds novel transformations that make verification easier, and controls the amount of existing optimizations by modeling the cost of program features (e.g., branches). This reduces the time to symbolically execute the resulting programs by up to 95×, and 58% on average.

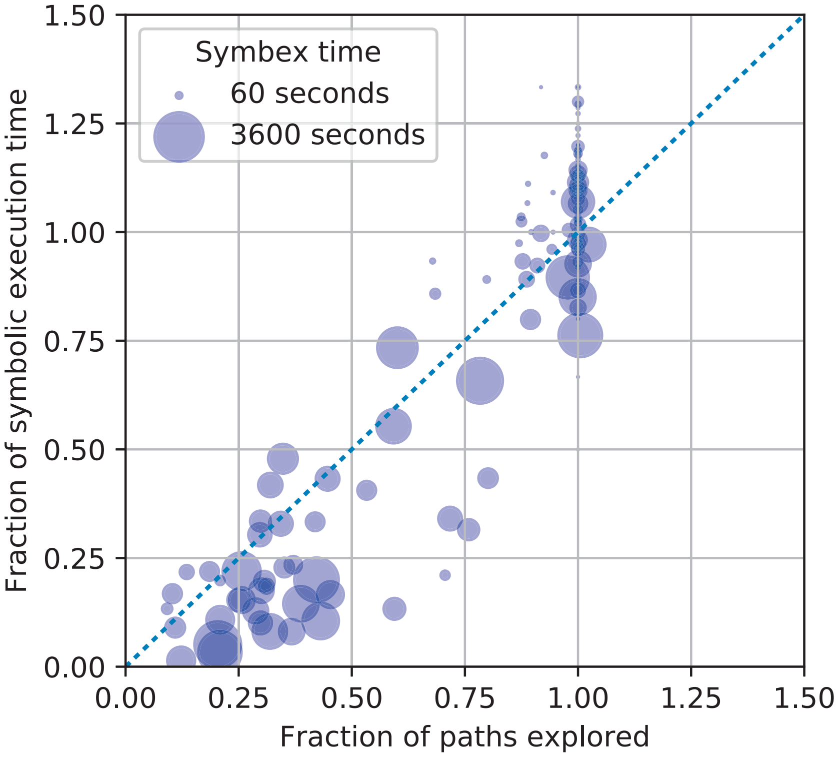


Figure 9: Effect of -OSYMBEX on symbolic execution time and the number of explored paths. Each dot represents an experiment. Its size corresponds to the duration of the experiment with -O3. Its position corresponds to the fraction of time needed when using -OSYMBEX relative to -O3, and the fraction of paths explored in this case. The reduction in explored paths correlates with reduction in time, but is not the only cause, as evidenced by the points below the diagonal.

4 ELASTIC INSTRUMENTATION

4.1 INSTRUMENTATION AND THE PARETO PRINCIPLE

In this chapter of the thesis, we are interested in *instrumentation.* This is a common form of program transformation, whereby a human or tool adds extra *instrumentation code* to software to enhance its orthogonal properties: robustness, reliability, security, the ability to trace and understand a program, etc.

To start, we define instrumentation code and give examples how it is used (Section 4.1.1). Then, we introduce elasticity, i.e., the observation that instrumentation can scale its cost and effects (Section 4.1.2). Elasticity is a useful concept because it enables exploitation of the Pareto Principle (Section 4.1.3), and thus obtain most of the benefits of instrumentation at low overhead.

4.1.1 *Semantics of instrumentation code*

Software needs to be functional, but this is not enough. We care about many other properties that are orthogonal to the core functionality of software. Software should:

- be safe in unforeseen circumstances,
- detect problems quickly,
- provide developers with logs, traces or other information for troubleshooting and reporting,
- be easily testable,
- protect its integrity and security when under attack.

These properties are *orthogonal* in the sense that the software would still work without them. In fact, developers typically have multiple build configurations where they enable or disable different transformations to choose the best trade-off between orthogonal properties and speed. These configurations do not affect the correctness of the software because orthogonal properties are not part of the software's specification; they add a different kind of value.

Instrumentation code is code that has been added to the program solely to enhance orthogonal properties. It can be added manually by developers, or automatically by a tool. Instrumentation differs from other program transformations, like those seen in Chapter 3, in that it is purely additive.

Examples of instrumentation code:

- *Assertions* are developer-written checks that verify the consistency of a program's state. They check for "impossible" conditions that should never happen, except due to software bugs. Because assertions are orthogonal, they are often compiled conditionally and removed from release builds where performance matters and bugs are presumably rare.

- *Logging code* records information about the program's execution for auditing and understanding the program. Like assertions, logging code is typically added by developers and can be disabled at run-time.

- *Safety checks* detect many forms of illegal behavior in the program, such as memory safety violations, deviations from regular control flow, integer overflows, or data races. We describe a number of automated tools to add safety checks to programs in Section 2.3.2. Automated tools can add these checks exhaustively to all program locations where a given type of illegal behavior could occur, thus ruling out entire classes or problems.

- *Coverage measurement code* detects which parts of a program have been executed. This information is useful during manual and automated testing, where it provides feedback on the quality of the tests.

4.1.2 *Elasticity and why it matters*

Program instrumentation is always a trade-off, because instrumentation code comes at a cost. This cost comes in the form of increased CPU usage, increased memory usage, increased binary size, disk space required to store logging/profiling data, etc. This cost is a severe problem that limits the applicability of instrumentation. In the sections to follow, we will see cases where instrumentation cost is a bottlenecks for overall system performance, and cases where developers forgo the use of instrumentation altogether because its overhead exceeds what they are willing to pay.

Elasticity is the characteristic of instrumentation that determines whether we can control this trade-off. We define elasticity as the ability to scale the costs and effects of instrumentation. The term is used similarly for example in cloud computing, where an "elastic cloud" allows developers to scale computing resources according to demand. Elasticity enables selective instrumentation: it allows developers to add instrumentation code to selected parts of a program only, in order to gain partial protection at a reduced overhead.

Elasticity of instrumentation combines two properties: First, it captures the *granularity* at which the amount of instrumentation can be adjusted. The best case is when instrumentation consists of many small independent pieces, because these can be enabled individually at fine granularity.

Second, elasticity captures the *cost-proportionality* of instrumentation. For truly elastic instrumentation, the cost is solely a function of the amount of instrumentation that is enabled ("pay for what you use") with little or no fixed costs.

Our first observation is that instrumentation, whether added by humans or by tools, is indeed often elastic. Instrumentation has a fine granularity it consists of many parts that are small, independent, and localized. For example, instrumenting the FFMPEG benchmark with AddressSanitizer adds almost 400,000 memory safety checks to the program. Each of these protects its own memory access instruction. We call these independent parts *instrumentation atoms.*

However, some types of instrumentation are only partially cost-proportional. For example, memory safety checks typically perform book-keeping to keep track of which memory addresses the program may legally address. Even a partially-instrumented program must perform this book-keeping completely, so that checks are accurate. The overheads due to such book-keeping limit the speedups gained using selective instrumentation techniques, a limitation which we discuss in Section 4.6.5.

Our second observation is that selective instrumentation yields useful effects. At first, one could argue that partial protection is useless. For example, could not a single unprotected buffer overflow be enough for an attacker to take control over a program? We find time and again that this is not so. Our evaluation will show, for example, that selective use of memory safety checks prevents vulnerabilities, and that partial coverage instrumentation produces enough information to guide coverage-driven fuzz testing.

Our third observation is that different instrumentation atoms often have very different characteristics. This enables exploitation of the Pareto Principle, as described in the next section.

Elasticity matters for two reasons. First, it enables selective instrumentation. This gives developers the possibility to control the cost of instrumentation, rather than the binary choice of using instrumentation at its full cost or forgoing it altogether. As a result, developers can use instrumentation tools in situations where it would be impossible without elasticity.

Second, elasticity is a design principle that provides benefits in its own right. In Section 4.4, we analyze an example of this in detail. We replace a check for control-flow-integrity by a new form that is fully cost-proportional. This alone reduces overhead by 71%, and enables further benefits due to selective instrumentation.

4.1.3 *The Pareto Principle for instrumentation code*

Thanks to elasticity, we can think of instrumentation as a set of atoms, with each atom having a cost and an effect. Further analysis shows that atom costs tend to vary greatly between atoms. A few atoms are expensive and a large majority of atoms are cheap. In one of our experiments, where we used CPU overhead as a cost metric, we found that almost half of the overhead in compute-intensive benchmarks comes from only 1% of all atoms (Section 4.5.1.3).

This observation that a small number of causes contribute most of the effects is called the Pareto Principle. Another name for the principle is the 80/20 Rule, because typically, 20% of all causes account for about 80% of the effects.

When applied to instrumentation code, the Pareto Principle implies that the trade-off achieved by selective instrumentation is favorable. By identifying the atoms with the highest benefit/cost ratio, we can quickly obtain 80% of the total benefit while paying only 20% of its cost. Adding more instrumentation on top of that yields diminishing returns, because the remaining instrumentation atoms are increasingly expensive.

Interestingly, the Pareto Principle applies even in situations when we cannot precisely quantify the benefit of instrumentation atoms. The reason is that the cost distribution is so highly skewed. When comparing two atoms, the difference in their cost can be several orders of magnitude, and this tends to dominate the difference in their benefit. Thus, instead of considering the benefit/cost ratio to select the best atoms, using the approximation 1/cost works just as well.

That said, we have found cases where cheap atoms have particularly high benefits. For example, many security vulnerabilities seem to be in rarely-executed code, where they can be prevented by cheap memory safety checks (Section 4.5.1). Similarly, fuzzers that use cheap coverage instrumentation to guide them toward new program parts are effective at finding bugs (Section 4.5.2). This makes the trade-off achieved using selective instrumentation even more favorable.

At this point, a word of caution is due: even the most favorable trade-off remains a trade-off. Applying instrumentation partially does reduce its effects on the program. We have seen cases where this did harm. For example, our evaluation of FUSS contains one benchmark where aggressive selective instrumentation prevents a fuzzer from finding a bug. Thus, in the absence of cost constraints, using full instrumentation is the best choice.

However, if there are cost constraints and if a trade-off needs to be made, then elasticity implies that there are many points in the trade-off space to choose from, and the Pareto Principle implies that there exist favorable points that obtain high benefit and large cost savings.

4.2 USE CASE: OPTIMIZING SYSTEM-CODE SECURITY VS. OVERHEAD

This section of the thesis is about a specific type of program transformation: instrumenting a program using sanity checks to detect bugs and vulnerabilities. In the following pages, we:

- motivate the use of sanity checks and introduce the notion of "overhead budget" to control their overhead;
- explain the semantics of sanity check instrumentation and why they belong to the class of elastic program transformations;
- present ASAP, a tool to automatically tailor sanity checks to achieve a good trade-off between system-code security and overhead.

4.2.1 *Desired property: high system-code security subject to a cost budget*

Security is undeniably important. To focus the present work, we consider security with three restrictions. We target systems software, achieve security through sanity checks, and require that we can control the overhead of these checks. Here is why:

Systems software is particularly subjected to conflicting demands of security, productivity, and performance. A lot of systems code is written in unsafe languages, like C/C++, because they have low runtime overhead, they enable low-level access to hardware, and because they are sometimes the only way to use legacy libraries or tool chains. The drawback is that unsafe languages burden the programmer with managing memory and avoiding the many behaviors left undefined by the language specifications. This makes it especially hard to write secure software; but the security of the entire software stack depends on input parsers, language runtimes, cryptographic routines, web browsers, OS kernels, and hypervisors written in these languages. Even extensive test suites and the use of tools like Valgrind still leave holes in the code. It is thus not surprising that buffer overflows are still the #1 vulnerability exploited by attackers [105] and that new ones have been revealed to take control of browsers and OSs in every edition of the Pwn2Own contest [115] since 2007.

Developers do have techniques available for "retrofitting" security and safety into their software. Tools like AddressSanitizer [118], UndefinedBehaviorSanitizer [36], ThreadSanitizer [119] etc. insert *sanity checks* into the code to verify at run-time that desired safety properties hold. These checks might verify that array indices are in bounds, that arithmetic operations do not overflow, or that no two threads write to a variable concurrently. If a sanity check fails, it typically is unrecoverable, and the program is aborted. Other than that, sanity checks do not affect the program state.

Unfortunately, such approaches are hardly ever used in production because of their overhead. The introduced sanity checks slow down the program. The strongest forms of protection introduce over 100% overhead, which is often more than developers are willing to pay in production environments.

It seems that the conflict between security, productivity and performance is inescapable. With existing solutions, developers can choose at most two of the three properties.

We propose to abolish the binary choice between high and low performance, and instead let developers specify precisely how much overhead they are willing to pay for security. We then generate a selectively instrumented program containing a subset of the available sanity checks that satisfies the developers' constraints. We measure our success by computing the *sanity level*, i.e., the fraction of sanity checks that we have preserved.

The elasticity of sanity checks suggests that this may be possible. The Pareto principle gives hope that this technique may even achieve high security at low overhead. Moreover, we will show that it can be completely automated and used with a wide range of sanity check instrumentation tools, so that developer also retain productivity.

The next sections will introduce ASAP, the first fully-automated approach for instrumenting programs subject to performance constraints. It allows developers to specify an *overhead budget*, and then automatically profiles and selects checks such as to build a program that is *as secure as possible* for the given budget.

We will first explain the semantics of sanity check instrumentation that the technique relies on (Section 4.2.2), and then describe how we designed our system to reason about the cost and effect of checks and optimize them in a principled way (Section 4.2.3).

4.2.2 *Semantics of program transformations that affect security*

Our technique ASAP works with program transformations that make software more secure by adding *sanity checks* to critical operations. These checks verify that the preconditions of an operation are satisfied. For example, sanity checks can guard memory accesses to ensure that the target location is valid, or guard arithmetic operations to ensure that the operands cannot cause an overflow. If the preconditions hold, the operation proceeds, otherwise the check aborts the program.

A range of properties can be enforced using sanity checks. ASAP works with checks for *memory safety*, checks that detect *undefined behavior,* and checks that detect *data races.* Support for other types of checks can be easily added.

Developer can choose from a range of tools to add such checks to programs automatically. Automation is useful because such tools can

Type	Tool	Overhead	ASAP
Memory	WIT	7%	
	CPI	8%	
	SAFECode	10%	
	BaggyBoundsChecking	60%	
	AddressSanitizer	73%	✓
	SoftBound/CETS	116%	✓
UB	UndefinedBehaviorSanitizer	71%	✓
Data races	ThreadSanitizer	400%	✓

Table 5: Automatic solutions to enforce program safety. Overheads are those reported in the corresponding publications. A check mark in the last column indicates that our ASAP prototype includes support for the tool.

exhaustively protect operations of a certain type, e.g., memory loads and stores. This adds a large number of checks but guarantees that the corresponding reliability property cannot be violated. Table 5 lists popular tools along with their safety property and CPU overhead. We reviewed these tools in more detail in Section 2.3.2.

To understand how ASAP works and what it assumes, we define a sanity check to be a piece of code that tests a safety condition and has two properties: (1) a passing check is *free of side-effects*, and (2) a failing check *aborts* the program. This characterization of sanity checks has important implications: First, ASAP can automatically recognize sanity checks in compiled code. Second, removing a sanity check is guaranteed to preserve the behavior of the program, unless the check would have failed.

The sanity checks seen by ASAP do not necessarily correspond exactly to operations in the program source, since sanity check interact with other transformations and optimizations performed by the compiler. ASAP benefits from its tight integration with the compiler. Depending on their type, the compiler may be able to eliminate certain sanity checks on its own when they are impossible to fail. Other transformations such as function inlining can duplicate static sanity checks in the compiled program. This refines the granularity of ASAP: if there are multiple copies of the same function, ASAP can distinguish the individual call sites and may choose to optimize only some of them.

Some types of sanity checks depend on additional program transformations. Memory safety checks work in tandem with changes to the memory allocator. The allocator performs extra work to record which regions in memory are currently accessible to the program, or to store metadata for pointers to keep track of the range of addresses they may access. Some techniques also change the layout of objects in memory: WIT [3] and AddressSanitizer [118] insert "poisoned" bytes in-between objects and report an error when the program accesses them. Baggy Bounds Checking [4] pads object sizes to the next power of two, because this enables more efficient lookup of metadata. These changes are not part of sanity checks by our definition. The overhead they introduce remains as residual overhead and is unaffected by ASAP.

Dynamic data race detectors, like ThreadSanitizer [119], combine checks and metadata management as follows: ThreadSanitizer adds a check to each memory access to determine whether another thread has accessed the same location concurrently without synchronization. It also logs the access, thereby gathering the metadata needed for future checks. In addition, ThreadSanitizer monitors all synchronization operations to determine cases where accesses by multiple threads are legal.

When using ThreadSanitizer with ASAP, it is worth relaxing the definition of sanity check to increase elasticity. We can consider the tracking of memory accesses to be an elastic operation, because a tool that tracks only some memory accesses has false negatives but never suffers from false positives. In other words, the consequence of removing tracking is the same as the consequence of removing checks: some data races may no longer be detected. The monitoring of synchronization operations however needs to be complete, otherwise the data race detector can report false positives.

4.2.3 *Design and implementation of ASAP*

We now present ASAP, a technique to automatically adapt the overhead of sanity check instrumentation. ASAP takes as input a software system and a workload, as well as one or several instrumentation tools. It then applies these tools to obtain a full set of available sanity checks. After estimating the cost of checks by profiling, ASAP then selects and applies a maximal subset of checks, such that the combined overhead is within budget. Figure 10 illustrates this workflow and shows the inputs and outputs of each step.

ASAP is the first technique to automatically exploit elasticity to enhance existing instrumentation tools. As such, it is the ancestor of FUSS and BinKungfu, which we present later in this thesis. ASAP shares with these techniques parts of the overall workflow and optimization process. We next describe this workflow in more detail, and then present the ASAP prototype in Section 4.2.3.2. We defer a discussion of other design points to Section 4.6, where we compare all three techniques.

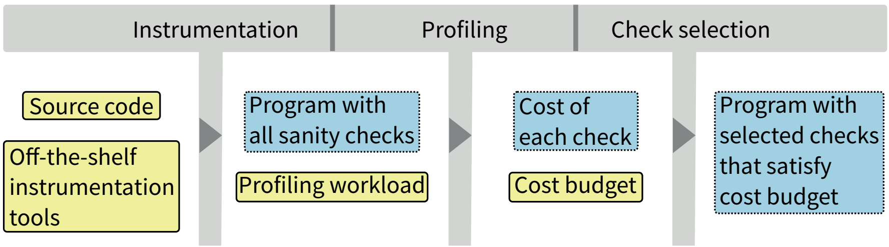

Instrumentation
Profiling
Check selection
Source code
Off-the-shelf instrumentation tools
Program with all sanity checks
Profiling workload
Cost of each check
Cost budget
Program with selected checks that satisfy cost budget

Figure 10: The three steps of ASAP's workflow. Yellow boxes represent inputs provided by the user, whereas blue dotted boxes are artifacts and results computed by ASAP.

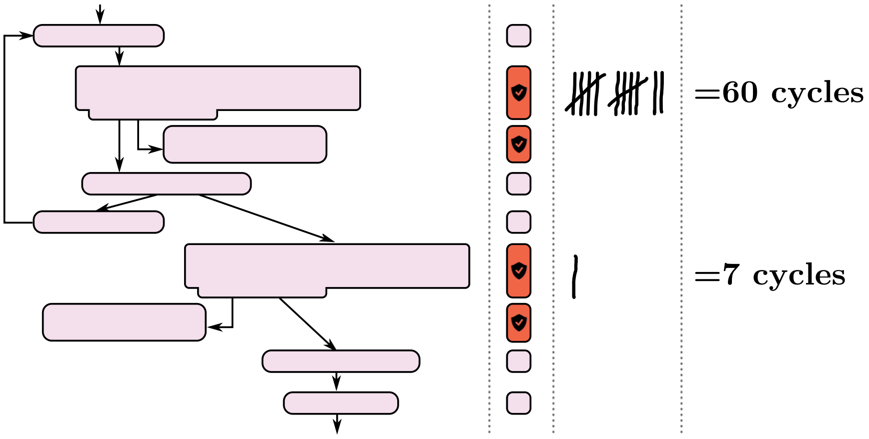


Figure 11: Recognizing sanity checks and measuring their cost. The figure show an example control-flow graph fragment of an instrumented program. ASAP first recognizes all the sanity checks (shown in red) by their structure. During profiling, ASAP counts how often each instruction in these checks is executed. It then uses these counts to estimate the amount of time spent due to each check.

4.2.3.1 *The ASAP workflow*

A user of ASAP starts with a software system that is to be protected using one or several instrumentation tools. We designed ASAP to be part of the software's compilation process, just like the instrumentation tools that it works with. Compilation using ASAP consists of three steps: *instrumentation*, *profiling*, and *check selection*.

INSTRUMENTATION The user starts by compiling the target program with full instrumentation enabled. This step depends on the specific instrumentation tool, but can be as simple as adding an additional compilation flag (e.g. `-fsanitize=address` for AddressSanitizer). This leads to a binary (or several) that is protected, but too slow to run in production.

ASAP can recognize sanity checks in the instrumented program by looking for code that aborts the program under some condition, and is free of side-effects otherwise. The left half of Figure 11 illustrates an instrumented program. A red annotation in the second column shows which parts ASAP recognizes as sanity checks.

PROFILING The second step consists of profiling the application against a suitable workload and computing the cost of each check. To obtain profiling data, ASAP further instruments the program from step 1 with profiling counters. Similar to GCOV [49], it adds one counter per edge between basic blocks. At each branch, ASAP inserts an increment of the corresponding counter.

Once the profiling run finishes, ASAP computes from the counter values the number of times any given instruction in a sanity check has been executed. By multiplying this value with a static estimate of the CPU cycles required to execute that instruction, it computes the accumulated cost for that instruction. The total cost in CPU cycles of a given sanity check is then the sum of the costs of the instructions inserted by that check. The sum of the costs of all sanity checks in the program gives the total number of cycles spent in checks while executing the profiling workload with the fully instrumented program. This is shown in the right half of Figure 11.

Knowing the total amount of CPU cycles spent by checks, ASAP can now compute what fraction of the cost it can preserve to achieve a given target overhead. We call this number the *cost level* c. For this, ASAP needs two additional pieces of data, namely the maximum overhead o_{max} and the residual overhead o_{min}. It can obtain these by measuring the software with full instrumentation and with all checks removed, respectively. The overhead o is a linear function of c, and so ASAP uses the following formula to compute the cost level for a given overhead budget:

$$o = o_{min} + c \cdot (o_{max} - o_{min}) \Rightarrow c = \frac{o - o_{min}}{o_{max} - o_{min}}$$

CHECK SELECTION Knowing the cost of each check and the target cost level, ASAP now uses a simple greedy algorithm to compute a maximal set of checks to preserve, while staying within the overhead budget. It orders checks by cost and preserves them starting with the cheapest check, as long as the fraction of the total check cost allowed by the cost level c is not exceeded. Because the distribution of check cost is highly skewed, it is possible to preserve a fraction of checks that is much larger than the fraction c of the total cost.

ASAP eliminates all checks that have not been preserved by removing them from the instrumented program generated in step 1. It then re-optimizes the program using standard compiler optimizations. This ensures that all data computed solely for use by those sanity checks is also removed from the program. The result is an optimized, production-ready executable.

When production workloads have significantly changed from what was used during profiling, steps 2 and 3 can be repeated with an updated workload to re-estimate the performance trade-off and produce a newly adapted binary.

4.2.3.2 *Implementation*

This section presents the architecture of ASAP, and its core algorithms for detecting sanity checks, estimating their cost, and removing expensive ones from programs.

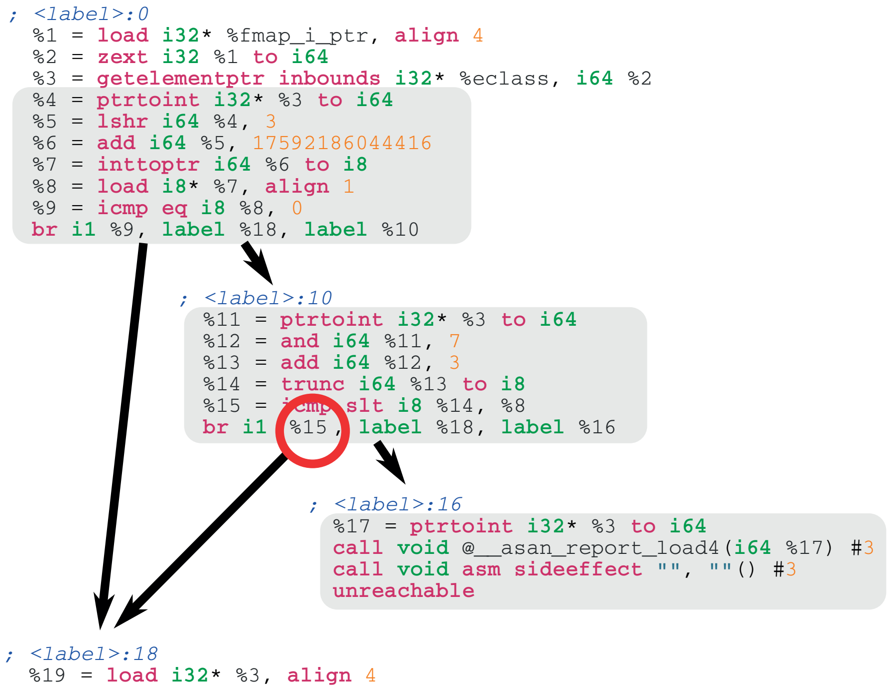
ASAP is based on the LLVM compiler framework and manipulates programs in the form of LLVM bitcode, a typed assembly-like language specifically designed for program transformations. It supports source-based instrumentation tools and those that have themselves been built on LLVM, which covers the majority of modern static instrumentation tools for C/C++/Objective C.

Users use ASAP through a wrapper script, which they invoke instead of the default compiler. In addition to producing a compiled object file, this wrapper also stores a copy of the LLVM bitcode for each compilation unit. This copy is used during subsequent stages to produce variants of the object with profiling code, or variants instrumented for a particular overhead budget.

ASAP works on programs one compilation unit at a time. It keeps no global state (except check data described later) and does not require optimizations at link-time. This is important for supporting large software systems that rely on separate and parallel compilation. The only phase in the workflow that requires a global view is the check selection phase, where ASAP computes a list of all sanity checks in the software system and their cost. This phase uses an efficient greedy selection algorithm described in Section 4.2.3.1 and has little impact on compilation time.

ASAP automatically recognizes sanity checks. Recall from Section 4.2.2 that a sanity check verifies a safety property, aborts the program if the property does not hold, and is otherwise side-effect-free. ASAP searches for sanity checks by first looking at places where the program aborts. These are recognizable either by the special LLVM `unreachable` instruction, or using a list of known sanity check handler functions. The sanity checks themselves are the branches that jump to these aborting program locations. Figure 12 shows an example, a memory safety check generated by the AddressSanitizer tool.

The listing is shown in the LLVM intermediate language, which uses static single assignment form (SSA); each line corresponds to one operation that computes a result and stores it in a virtual register, numbered sequentially from `%1` to `%19`. The sanity check in the listing protects a load from the address stored in register `%3`. It computes the metadata address (`%7`), loads shadow memory (`%8`) and performs both a fast-path check (the access is allowed if the metadata is zero) and



```
; <label>:0
  %1 = load i32* %fmap_i_ptr, align 4
  %2 = zext i32 %1 to i64
  %3 = getelementptr inbounds i32* %eclass, i64 %2
  %4 = ptrtoint i32* %3 to i64
  %5 = lshr i64 %4, 3
  %6 = add i64 %5, 17592186044416
  %7 = inttoptr i64 %6 to i8
  %8 = load i8* %7, align 1
  %9 = icmp eq i8 %8, 0
  br i1 %9, label %18, label %10

; <label>:10
  %11 = ptrtoint i32* %3 to i64
  %12 = and i64 %11, 7
  %13 = add i64 %12, 3
  %14 = trunc i64 %13 to i8
  %15 = icmp slt i8 %14, %8
  br i1 %15, label %18, label %16

; <label>:16
  %17 = ptrtoint i32* %3 to i64
  call void @__asan_report_load4(i64 %17) #3
  call void asm sideeffect "", ""() #3
  unreachable

; <label>:18
  %19 = load i32* %3, align 4
```

Figure 12: A sanity check inserted by AddressSanitizer, in the LLVM intermediate language. The corresponding C code is `cc1 = eclass[fmap[i]]` and is found in `blocksort.c` in the bzip2 SPEC benchmark. Instructions belonging to the check are shaded. The red circle marks the branch condition which, when set to `true`, will cause the check to be eliminated.

a slow-path check (the access is also allowed if the last accessed byte offset is smaller than the metadata). If both checks fail, the program is aborted using a call to `__asan_report_load4`.

ASAP computes the set I_c of instructions belonging to the check starting with the aborting function (`__asan_report_load4` in our example). It then recursively adds all operands of instructions in I_c to the set, unless they are also used elsewhere in the program. It also adds to I_c all branch instructions for which the target basic block is in I_c. This is repeated until I_c reaches a fixpoint. In Figure 12, a shaded background indicates which instructions belong to I_c.

The instructions in I_c are used for computing check costs as described in Section 4.2.3.1.

A number of different profiling mechanisms can be used to measure instruction cost. Our choice fell on GCOV-style profiling counters, where the profiler uses one counter per basic block in the program and adds a counter increment before every branch instruction. Profiling thus determines the number of times each instruction was executed; we obtain an estimate of the actual cost by applying the static cost model for instructions that is built into LLVM's code generator. The advantage of this approach is that it is robust and yields cost estimates at instruction granularity that are unaffected by the profiling instrumentation itself.

ASAP removes checks that are too costly from the program by altering their branch condition. In our example in Figure 12, it replaces the branch condition `%15`, circled in red, by the constant `true`, so that the check can never fail. The rest of the work is done by LLVM's dead code elimination pass. It recognizes that all shaded instructions are now unused or unreachable and removes them.

All steps ASAP performs are generic and do not depend on any particular instrumentation. In fact, the ASAP prototype works for AddressSanitizer, SoftBound, UndefinedBehaviorSanitizer, ThreadSanitizer, and programmer-written assertions. It contains exactly five lines of tool-specific code, namely the expressions to recognize handler functions such as `__asan_report_*`. This makes it straightforward to add support for other software protection mechanisms. Also, we did not need to alter the instrumentation tools themselves in any way.

We mention one extra feature of ASAP that helped us significantly during development: ASAP can emit a list of checks it removes in a format recognized by popular IDEs. This makes it easy to highlight all source code locations where ASAP optimized a check. Developers can use this to gain confidence that no security-critical check is affected.

ASAP is freely available for download at `http://dslab.epfl.ch/proj/asap`.

4.3 USE CASE: OPTIMIZING FUZZING EFFICIENCY

4.3.1 *Desired property: efficient fuzzing*

Fuzz testing, or testing a program with quasi-random inputs, is an effective way to find crashes and security vulnerabilities in software. Some of the most notorious security vulnerabilities were found in this way [141], and fuzzing has become a standard fixture in most commercial software testing strategies [98, 2].

Key to the effectiveness of fuzzing is test *quantity*, i.e., the ability to generate and try out new inputs at high rate. Large-scale fuzzing efforts (such as ClusterFuzz [7] that tests the Chromium web browser) test software round the clock, using as many machines as are available. The number of bugs found is limited by the number of CPU resources given to the fuzzer—intuition suggests that the more tests the fuzzer gets to run, the more likely it is to find bugs.

The other key ingredient is test *quality*. First, testing many random inputs in a blackbox fashion can at best discover shallow bugs, whereas picking inputs smartly can penetrate deeper into the program code and reduce the number of executions needed to find a bug. To improve the quality of generated inputs, modern fuzzers use feedback from prior executions to steer input generation toward those inputs that are more likely to uncover bugs. Second, detecting anomalous behaviors automatically during the test runs increases the chances of detecting the manifestation of a bug. Therefore, fuzzers check for a wide range of "illegal behaviors," with memory safety violations being the most popular. The premise is that higher-quality test are more likely to find bugs.

Figure 13 illustrates the workflow typical of such a "smart" fuzzer, like AFL [141] or LibFuzzer [88]. The objective is to quickly execute as much different code as possible, in order to reveal bugs that lurk within. The fuzzer repeatedly generates new inputs and subjects

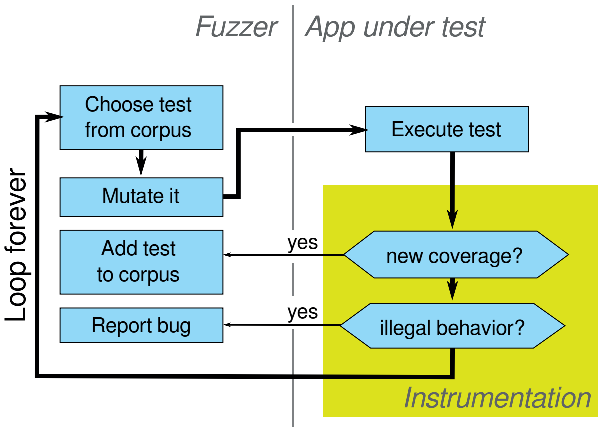


Figure 13: The workflow of a coverage-driven fuzzer

them to the program under test. If feedback from the program determines that the input is of particularly high quality, the fuzzer adds it to its corpus, where it becomes a starting point for future inputs.

Ideally, a fuzzer should simultaneously have high test throughput and high test quality, but unfortunately these two requirements conflict: obtaining good feedback comes at the cost of throughput. Both detecting program misbehavior and collecting code coverage information is done using program instrumentation: Fuzzers like AFL and LibFuzzer insert code into the target program to track coverage of each basic block or control-flow edge of the program, and they instrument the program with sanitizers that check for misbehavior (e.g., AddressSanitizer [118] detects illegal memory accesses like buffer overflows, and ThreadSanitizer [119] detects data races and deadlocks). This instrumentation code uses precious CPU cycles: in our measurements, we see on average 54% of all CPU cycles spent in instrumentation code, which means that the fuzzer can perform 2.2× fewer iterations of the fuzzing loop than without instrumentation. Said differently, fuzzers invest less than half of their resources into executing code of the target program, and spend the rest on improving test quality.

In this part of the thesis, our goal is to obtain high *fuzzing effectiveness*, i.e., to have both high test quality and quantity. An effective fuzzer is one that quickly and fully explores a program under test. We measure this by analyzing how much of the program is executed with fuzzer-generated inputs, and at what rate the fuzzer increases test coverage. Ultimately, faster program exploration enables the fuzzer to find bugs more quickly. To measure these end-to-end gains, we apply the fuzzer to test programs with known bugs, and measure the time it takes to reveal these bugs.

Our technique achieves higher fuzzing effectiveness using the Elasticity Principle. We observed that most of the benefit that the fuzzer obtains from instrumentation can be obtained by few CPU cycles invested into the most productive instrumentation atoms. We start by understanding how a fuzzer uses instrumentation (Section 4.3.2). This allows us to identify those instrumentation atoms that do provide useful information to the fuzzer, taking its current progress into account. We then re-instrument the program, keeping only the useful instrumentation. We describe our technique to classify and select instrumentation atoms in Section 4.3.3.

We call our technique FUSS, or "Fuzzing on a Shoestring", because it invests only few carefully selected resources into test quality.

Our evaluation in Section 4.5.2 shows that this approach obtains high test quality using only 12% of the CPU cycles needed for full instrumentation. This improves test quantity by 1.5× on average. For some benchmarks, this translates into a reduced time to find bugs. FUSS is up to 3.2× faster than the state of the art in these cases.

4.3.2 *Semantics of program transformations that affect fuzzers*

Fuzzers rely on instrumentation to detect program features triggered by tests, e.g., the execution of a basic block or a memory write past the end of a buffer. Whenever a feature is seen for the first time, the fuzzer reacts: it adds the test to its corpus of interesting test cases, or reports a bug. This is a type of genetic algorithm, where test cases that trigger previously unseen features survive and have offspring.

Each of the features that the fuzzer is interested in corresponds to a single instrumentation atom. Conversely, each atom detects a single or a small number of features.

Instrumentation atoms have a lifetime: once the fuzzer has explored all features that an instrumentation atom can detect, that atom can never again influence the fuzzer's behavior. A tool like FUSS could remove that atom without reducing the amount of feedback that the fuzzer obtains.

Different types of instrumentation can detect many features of interest.

Coverage bits detect when a particular edge in the control-flow graph of the program is executed. When a coverage bit fires, the fuzzer knows that it has found new code. A *Coverage counter* similarly detects how often an edge has been executed, and can signal to the fuzzer that it made progress when exploring a loop.

Coverage instrumentation atoms are the sensors through which the fuzzer detects interesting test cases, and their sensitivity influences how the fuzzer spends its time. The fuzzer will explore each interesting test case and invest time to mutate it. The ideal coverage instrumentation is just sensitive enough so that the fuzzer can detect each test case that triggers new behavior. However, it must not be too sensitive, otherwise the fuzzer spends excessive time exploring almost identical behaviors.

To illustrate the sensitivity trade-off, consider how AFL and LibFuzzer handle loops. They consider a test case that causes a loop to exit after a single iteration to be different from a test case that triggers two loop iterations. On the other hand, two test cases that trigger 100 and 101 iterations, respectively, are considered to be similar, and the fuzzers keep only one of them. AFL and LibFuzzer keep up to eight test cases to explore a loop, one each for iteration counts of 1, 2, 3, 4-7, 8-15, 16-31, 32-128, and >128. Note that they are more sensitive for small iteration counts than for large ones.

This sensitivity is useful, for example, to find inputs that contain keywords. A program that compares its input against a constant keyword, e.g., using `strcmp(input, "key")`, contains an internal loop that examines the input character by character. The fuzzer will find that the test cases `"k.."` and `"ke."` trigger a different number of loop iterations. It will thus further explore `"ke."`, and eventually find a

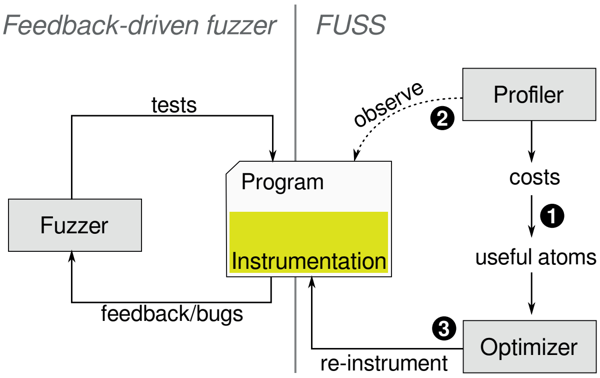


Figure 14: A feedback-driven fuzzer boosted with FUSS. During fuzzing, FUSS gathers profiling data to determine the cost of each instrumentation atom. From this, FUSS identifies those atoms that generate useful feedback cheaply, and re-instruments the program accordingly.

random mutation that transforms it into `"key"`. Without feedback from instrumentation, the fuzzer would not have found the prefixes `"k"` and `"ke"`, and would have been unlikely to generate the entire keyword through a single random mutation.

Safety checks detect abnormal conditions, alerting the fuzzer that a bug has been found. Such checks can be added by developers in the form of assertions, or automatically by tools. Both AFL and LibFuzzer recommend to use instrumentation tools for a variety of error conditions: Examples are UndefinedBehaviorSanitizer, ThreadSanitizer, AddressSanitizer, `FORTIFY_SOURCE`, AFL's libdislocator, and stack-protector [36, 119, 118, 70, 46].

Using safety checks increases the quality of tests, and thus the number of bugs that fuzzers can detect. Without them, developers need to hope that illegal behavior leads to a segmentation fault or other exception. This does not always happen, particularly for tricky cases such as use-after-free or buffer over-read bugs.

4.3.3 *Design and implementation of FUSS*

Figure 14 shows an overview of our design for FUSS. In this section, we explain the following points (numbered in the Figure). (1) Section 4.3.3.1 explains how FUSS classifies instrumentation atoms as useful or not. (2) FUSS obtains cost data for this classification using a statistical profiler, which we describe in Section 4.3.3.2. (3) We explain how FUSS re-instruments the program under test in Section 4.3.3.3.

4.3.3.1 *How FUSS identifies useful atoms*

When analyzing the types of instrumentation used by fuzzers, we find a pattern: an atom's usefulness decreases with the number of executions. This is most obvious for coverage bits, the lifetime of which ends immediately after it is executed for the first time. Similarly, the more powerful coverage counters count their number of executions and expire when that number lies within a certain range. Safety checks also tend to detect bugs during the first few executions, rather than when they are already well exercised.

This relationship between execution count and usefulness exists because the execution count summarizes three aspects of an atom: location, lifetime, and cost.

Location: Atoms in unexplored area are more useful than those in well-explored core locations of the program under test. Program areas that the fuzzer has explored often, in many different contexts, become less likely to reveal new behavior or bugs that instrumentation can detect. This is precisely why successful fuzzers guide their exploration to newly discovered areas, and why instrumentation there is more valuable.

Lifetime: Atoms at the beginning of their lifetime are more useful than others. In our analysis of different instrumentation types in Section 4.3.2, we saw that each atom "fires" a small number of times, and then expires. Each atom provides only a limited amount of information to the fuzzer. Once the fuzzer made use of all the information, the atom no longer adds value.

Cost: Cheap atoms are more useful than those with high cost. The argument is simply that, if we are on a budget, it is better to enable ten cheap atoms than one expensive atom. Because effective fuzzers need both high test quantity and test quality, it can be beneficial to impose a budget, and reduce instrumentation in exchange for a higher throughput. In this case, removing the most expensive atoms yields the biggest throughput increase.

An atom's execution count embodies all these characteristics. It is a simple metric that can be conveniently measured via profiling, and works well in practice. For these reasons, FUSS identifies atoms as useful or not based on their execution count. It derives this count from the number of CPU cycles spent on a particular atom, as described in the next section.

4.3.3.2 *How FUSS estimates cost*

The goal of FUSS's profiler is to obtain precise data about the cost of each instrumentation atom. The quality of this data determines how accurately FUSS classifies atoms.

Our design uses a statistical sampling profiler, Linux Perf [94]. The profiler examines the CPU state frequently (e.g., 4,000 times per second) while the fuzzer is running, and records the current program counter and recently executed program locations found in the CPU's last branch record (LBR). This gives FUSS a statistical distribution

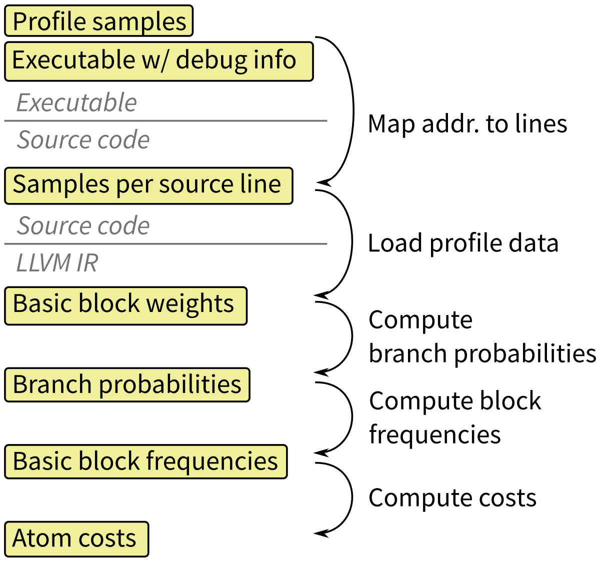


Figure 15: Steps to compute atom costs from data generated by a sampling profiler.

of CPU time over assembly instructions in the executable, including those corresponding to the atoms.

FUSS's optimizations work not on assembly code but on the LLVM Intermediate Representation (IR). To map profiling data to instructions in LLVM IR, FUSS uses a sequence of transformations shown in Figure 15. In a first step, FUSS uses the AutoFDO tool [32] to map profiling samples to source code lines, using debugging information embedded in the profiled program. This results in a distribution of execution time over the lines in the program, which the compiler can load along with the source code.

Once loaded, the LLVM compiler uses profiling data first to compute block weights, and then derives branch probabilities from block weights: each branch target receives a probability that indicates how frequently it is taken relative to other targets of the same branch. Branch probabilities are LLVM's preferred form of profiling metadata, because they can be kept consistent across program transformations. For example, inlining a function changes the weights of the function's blocks, but usually preserves the probabilities of branches.

Toward the end of the compilation process, FUSS computes atom costs from branch probabilities as follows. First, FUSS uses LLVM's built-in BlockFrequencyInfo pass to convert branch probabilities back to block frequencies. These block frequencies are relative to the function that contains the block; to obtain a frequency relative to the entire program, FUSS multiplies block frequency with the weight of the block's function. A function's weight is the number of samples in its entry block, and is proportional to how often the function was called during profiling. Finally, FUSS computes the cost of each instruction in an atom, by multiplying the instruction's latency (in CPU cycles) with the frequency of the block containing the instruction. The cost of an atom is the sum of the costs of its instructions.

The resulting atom cost is an approximation, both because the data gathered by the profiler is imprecise and because each transformation step further reduces precision. We discuss the consequences of these imprecisions when we analyze the limitations of FUSS in Section 4.6.3.

The advantage of this design is that it can obtain profiling data from an unmodified, running fuzzer, without restrictions on the type of fuzzer or the type of instrumentation used. In contrast, alternative profilers such as GCov (as used by ASAP) require modifying the program under test, and introduce an overhead of their own.

4.3.3.3 *How FUSS winnows instrumentation*

When FUSS has gathered information about the cost of instrumentation atoms, it generates an optimized version of the program.

We implement this re-instrumentation as a plugin for the LLVM compiler [84]. It receives the instrumented program in the compiler's intermediate language, LLVM IR. Our plugin scans the program for instrumentation atoms, obtains their cost, and elides those for which this cost is above a threshold.

Our design builds on ASAP. FUSS re-uses ASAP's concept of instrumentation atom and the ability to prioritize and select atoms. Beyond that, FUSS extends ASAP in two main ways.

First, FUSS recognizes any type of instrumentation, including coverage instrumentation used by fuzzers. ASAP itself recognizes sanity checks only; it identifies them by scanning the program for conditions that lead to aborts, as explained in Section 4.2.3.2.

FUSS more generally recognizes a "root" for every atom, which is an instruction that unambiguously identifies it. This can be the operation that increments a coverage counter, for example, or a function call into the instrumentation's run-time library. Starting from the root, FUSS finds the other instructions in the atom. These are the instructions that are used, directly or indirectly, by the root but not by any other parts of the program.

This technique can in principle recognize any type of atoms. A bit of care is needed for atoms that contain instructions with side effects. FUSS would normally consider the side effect to be part of the program's functionality, and thus would not add such instructions to an atom. However, it is possible to include side effects in atoms by making them part of the atom's root. This is how FUSS handles coverage instrumentation, where the side effect is a write to an array of coverage counters.

With this infrastructure in place, recognizing new types of instrumentation is as easy as adding a pattern to recognize a root instruction. This makes FUSS easily applicable to any type of instrumentation.

Second, FUSS extends ASAP with the ability to compute costs based on data from a sampling profiler. The choice of using a sampling profiler simplifies the workflow compared to ASAP. In Section 4.2.3.1, we discussed how ASAP creates three version of an executable: an initial version to serve as basis for future steps, a version for profiling, and a final optimized version. ASAP uses a compiler wrapper script to modify the build process and ensure that the right version is used as input and output of each stage of the workflow. In contrast, FUSS can simply observe the fuzzer while it is running, and does not need to perform any setup for profiling.

To perform the actual re-instrumentation, FUSS recompiles the program with an extra compiler flag that triggers the FUSS compiler pass, and another flag that points the compiler to the file containing profiling data. The process is simpler than for ASAP, because there is no need for compiler wrapper scripts or state from previous steps (other than the profiling data file).

During re-instrumentation, FUSS removes from the program those atoms for which the cost exceeds an empirically determined threshold. The value of this threshold strikes a balance between test quantity and quality: The threshold must be low enough to eliminate most of the overhead due to instrumentation, and yet high enough that the preserved instrumentation is sufficient to guide the fuzzer. Unfortunately, we did not find a good automated way to determine this threshold. We simply tested multiple values, and chose one that worked well.

4.3.3.4 *Prototype*

We have implemented a prototype of FUSS that developers can use to fuzz-test arbitrary software. Our prototype uses the LibFuzzer fuzzing engine, and also includes preliminary support for AFL. Its input is the source code of the software to test, along with commands to compile it and to invoke the fuzzer. From this, our prototype produces an optimized executable that can be fuzzed with high throughput. Our prototype performs five steps:

Compile: FUSS starts by compiling the application using the user-provided command. Users can choose the compilation flags to enable instrumentation, safety checks, assertions etc. The result of this step is a valid, but slow, executable to be fuzzed.

Warm-up: FUSS then fuzzes the application under test. This step is identical to how the fuzzer is normally used. The warm-up period is short (60 seconds were sufficient in our experiments). It serves to explore the core parts of the program, and results in an initial corpus of testcases.

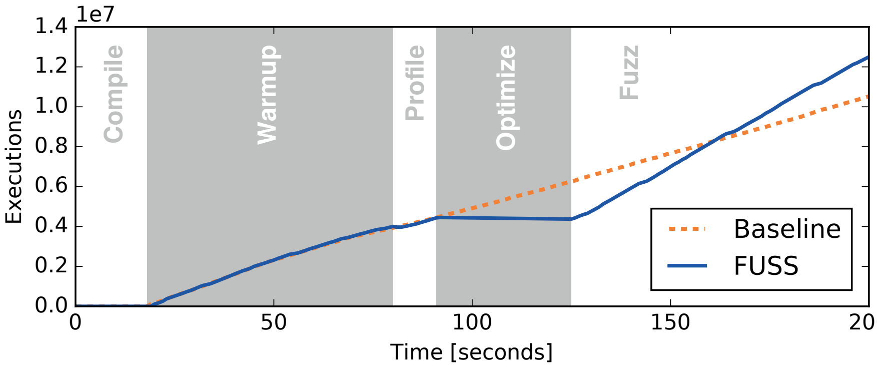


Figure 16: Number of test executions when fuzzing the Harfbuzz benchmark. In this figure, the five steps performed by FUSS are highlighted with alternating shades of gray. Warm-up starts after the benchmark is compiled, at 20 seconds. After a 60-second warm-up time, FUSS performs 10 seconds of profiling, and then recompiles the benchmark to optimize it. During recompilation, no fuzzing progress is made because the CPU is dedicated to the compilation. Afterwards, fuzzing continues with a higher throughput than before.

Profile: At this point, FUSS attaches the Perf profiler to the running fuzzer. It gathers data that serves to identify the well-explored core parts of the program. The output of this step is a statistical distribution of execution time over program locations.

Optimize: FUSS obtains a cost estimate for each instrumentation atom from the profiling data, and removes those atoms that cost the most CPU cycles. FUSS performs this step inside the compiler. It activates the necessary compiler passes by adding FUSS-specific flags to the compilation command, and re-compiling the application under test. During re-compilation, we stop the fuzzer to make the CPU available for the compiler. The result is an executable with instrumentation specifically tailored for fuzzing.

Fuzz: After optimization, fuzzing can continue on the optimized binary. From the fuzzer's perspective, nothing has changed, except that the program under test now runs faster.

Figure 16 illustrates the four steps performed by FUSS. We plot the evolution of executions over time, for FUSS and a traditional fuzzer. We use the Harfbuzz benchmark as example; other benchmark from our collection would show similar trends. FUSS initially invests time to gather profiling data and optimize the program, so its progress falls behind the baseline. However, the optimized fuzzer very quickly amortizes the invested time by virtue of its superior speed. Note that we cap Figure 16 at 200 seconds to highlight the steps performed by our prototype. A typical fuzzing session would last much longer.

While fuzzing, the fuzzer saves all interesting testcases in its corpus. This corpus contains all the fuzzer's progress. When the fuzzer is interrupted (e.g., after FUSS recompiles the program under test), it reloads the corpus from disk and, within few seconds, resumes fuzzing at the point where it stopped.

The FUSS prototype will be made available as open-source software, at `http://dslab.epfl.ch/proj/fuss`.

4.4 USE CASE: BINARY HARDENING

This section describes how elastic instrumentation principles apply to the task of hardening software binaries. We compare two transformations that enforce a security property called control-flow integrity (CFI): an implementation of classic ID-based CFI [1] and a novel implementation using elastic checks. Our elastic implementation can be applied selectively in situations where full protection has prohibitive overhead.

We implemented the two CFI-transformations in a binary hardening tool called BinKungfu tool, and evaluated it using benchmarks from the Cyber Grand Challenge (CGC) [40]. We found that selective instrumentation was useful to protect programs in the presence of tight overhead limits. For example, classic CFI fails to meet a 5% overhead target for 17 benchmarks from the CGC set. In 10 of these 17 cases, BinKungfu can use elastic checks to obtain partial protection, which is often good enough to thwart attacks.

4.4.1 *Desired property: high protection and small, fast code*

In this section, we explain what it means to protect a program against attacks, and how we quantify the security and overhead of a protection mechanism. We discuss these terms and metrics in the framework established by the DARPA Cyber Grand Challenge (CGC), because its formalization is useful and because we evaluated our prototype, BinKungfu, with benchmarks from CGC.

4.4.1.1 *Quantifying protection*

We desire to protect programs to prevent attackers from gaining control over our machines by exploiting the program's vulnerabilities. Vulnerabilities arise from *bugs* in the software: unintended program behaviors that can be triggered by specific inputs. When a bug allows attackers to gain control of the machine or read secret data, we call the bug a *security vulnerability*. We call the particular input that an attacker uses to trigger the vulnerability an *exploit*.

The organizers of CGC precisely defined what it means for an exploit to be successful. They distinguish two types of exploits: exploits of type I hijack control-flow, and type-II-exploits leak data. A control-flow hijack exploit is successful if the attacker can set the processor's program counter to an arbitrary value, and control the contents of at least one processor register. In most real-world settings, this is sufficient to execute arbitrary code on the machine where the program runs. Alternatively, a data leak exploit is considered successful if the attacker can obtain the contents of a designated memory area that the program tries to keep secret. Again, this capability is usually equivalent to reading arbitrary values from the program's memory.

These definitions are useful from an attacker's point of view, but difficult to use when defending programs against attacks. In principle, a program is defended when no successful exploit exists. Alas, the presence of exploits or the number of vulnerabilities in a program is in general unknown. Our inability to find an exploit is by no means a proof that the program is safe.

Thus, defenders usually reason about security in terms of security properties enforced by the program. These properties are satisfied by all possible program executions and rule out specific classes of vulnerabilities. A program is then considered "safe" if its security properties prevent all known exploit methods.

For example, the *control-flow integrity* property (CFI) states that the program will only ever execute code that is part of the program's static control-flow graph (CFG). The property means that all changes to the processor's program counter correspond to valid transitions in the CFG. This prevents a large class of attacks from succeeding. In particular, attackers can now no longer take control of the program by setting the program counter to arbitrary values. On the other hand, different types of attack (such as leaking secret values from the program's memory) might still be possible.

To enforce control-flow integrity, defenders need to ensure that each program instruction that changes the program counter does so according to the valid CFG transitions. For most instructions, the effect on the program counter is fixed, and its validity can be statically checked. In the X86 architecture, the exceptions are indirect jumps, indirect function calls, and function returns. For these instructions, the target address is computed at runtime and could depend on attacker-controlled input. CFI defenses add a check between the address computation and the jump to ensure the validity of the target address.

CFI defenses guarantee that a program is *secure* (i.e., has the CFI property) if and only if all dynamic control-flow transfer instructions are preceded by a check. Even a single unprotected transfer could allow a powerful attacker to gain control.

Yet this black-and-white approach to quantify security is limited. In this thesis, we instead consider the fraction of protected control-flow transfer instructions to be an indicator for the resilience of a program against attacks. In situations where overhead constraints limit the amount of extra checks that can be added to a program, we think it is better to protect the program partially than to leave it completely unprotected.

The justification for our metric comes from the observation that bugs often have a localized effect. For example, a buffer overflow might allow an attacker to only overwrite a single return address. In this case, only one control-flow transfer instruction is affected, namely the return instruction that uses the attacker-controlled address. If this instruction happens to be protected, the program is safe.

We do not attempt to formalize the amount of control that a bug gives to an attacker, or make assumptions about the number or distribution of bugs in a program. As such, the fraction of protected program instructions does not quantify safety or correspond to a probability of being protected against a vulnerability. Our work on ASAP did find correlations between the fraction of safety checks in a program and the number of detected bugs (Section 4.5.1); but all we claim here is that more safety checks are better than fewer safety checks.

4.4.1.2 *Quantifying overhead*

In contrast to security, *overhead* is relatively easy to specify and measure. In the context of CGC where we evaluate our techniques, there are three types of overhead:

- *Time overhead,* which is the increase in CPU time needed by the protected program compared to the original program. This overhead comes directly from executing safety checks, but also indirectly from their effect on caches, program layout, branch predictors, etc.
- *Memory overhead,* being the increase in memory consumption of the protected program compared to the original. This overhead can come from increased code size, but also be due to memory used by the protection mechanisms.
- *File size overhead,* i.e., the increase in program size due to the protection mechanisms.

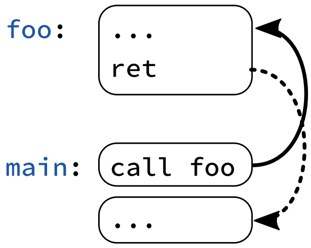
CGC further defined rules for weighting different types of overhead against each other. For the kind of instrumentation performed by BinKungfu, the dominant factor was memory overhead, and the rules meant that protection mechanisms may increase memory consumption by at most 5%. The instrumentation that BinKungfu performs increases memory consumption through its size. The instrumentation instructions occupy space in the program's code segment, which gets loaded in memory when the program executes. As a result, BinKungfu tries to limit the amount of instrumentation code that it adds to a program.

Based on this trade-off between security and overhead, the goal for BinKungfu is thus: maximize the fraction of program locations protected with CFI checks, while keeping code size low to maintain memory overhead below 5%.

```c
void foo() {
  char buffer[3];
  strcpy(buffer, "much data!");
}

int main() {
  foo();
  return 0;
}
```

(a) Source code in the C language



(b) Control-flow graph

Figure 17: A minimal program and its CFG. The return from `foo` is an indirect transfer, which fails due to a buffer overflow.

4.4.2 *Semantics of program transformations for hardening*

We now look in detail at the program transformations available for program hardening. We limit ourselves to those transformations that enforce control-flow integrity (CFI). CFI is a popular security property; we contrasted it with other security properties and described some implementations in Section 2.3.2, and will now explain its semantics. Figure 17 shows an example program along with its control-flow graph (CFG).

The CFG is derived from the rules of the programming language, in this case C. The rules specify, e.g., how a function call transfers control to the callee, and how control returns to the caller afterwards.

According to this specification, compilers transform the program into machine code where these control-flow transfers take the form of direct and indirect jumps. Compilers use direct jumps if the target address is known and static, as is the case for the function call in our example. They use indirect jump if the target address is variable or computed. The return instruction in our example is an indirect jump, because the function `foo` could be called in several places, and so there are multiple potential target addresses.

Indirect jumps can go wrong due to bugs in the program. In our example, the programmer mistakenly copies too much data into a buffer. The extraneous data overrides other values on the program's stack, including the return address of the function `foo`, so that `foo`'s return instruction will jump to an invalid location.

Control-flow integrity transforms all indirect jump instructions to prevent this case from happening. The semantics of this transformation are as follows:

Completeness: instrument all indirect control flow transfers. The transformation adds a check to each such instruction, at the point just after the target address has been computed and just before the

```
call foo
...
```

(a) Original call site

```
          call foo
          jmp after_id
          .dword id
after_id: ...
```

(b) Call site with ID

```
ret
```

(c) Original return instruction

```
       push edi
       mov edi, [esp + 4]
       cmp [edi + 2], id
       je id_ok
       call abort
id_ok: pop edi
       ret
```

(d) Return with check

Figure 18: An ID-based CFI check to verify that a return instruction jumps to a valid call site.

jump is performed. The check verifies whether the target address is valid according to the CFG, and aborts the program otherwise.

Soundness: checks accept all valid targets. Failure to do so might cause false positives, i.e., situations where a CFI check aborts the program even though nothing went wrong.

The classic CFI check relies on IDs embedded in the program code. These are byte sequences that identify target locations. Completeness requires that these IDs appear only at valid target locations. Soundness requires that all target locations are marked. Figure 18 gives an example of an ID-based check as implemented in BinKungfu.

The top half of Figure 18 shows a call site instrumented with an ID. The ID is a four-byte constant value. It is preceded by a two-byte short jump instruction, which causes the processor to skip over the ID when executing the program. Together, the ID and jump form a six-byte sequence with no effect on the program behavior.

The bottom half of the figure shows a return instruction preceded by a check. The check first loads the return address from the stack into the `edi` register, and then compares the four bytes at `[edi + 2]` to the ID. If the return address is valid, this location will indeed contain the ID, and so the check succeeds.

Note that BinKungfu uses an approximation of the program's CFG for ID-based CFI. In particular, each return instruction is allowed to target any call site. We chose this design because obtaining a precise CFG is difficult when analyzing X86 executables. It simplifies return checks a bit, because all checks can use a single shared ID.

Instrumentation	Bytes	Avg. Count	Avg. Overhead
`ret` check	20	49	980
id at call site	6	160	960

Table 6: Overhead introduced by CFI instrumentation for return instructions. Numbers are averaged over all benchmarks from the CGC qualification.

4.4.3 *Design and implementation of elastic binary hardening*

In our elastic instrumentation framework, we forgo the completeness requirement of full CFI instrumentation, and instead maximize the number of checks we can add to the program. In exchange, we want to achieve lower overhead.

Alas, ID-based CFI checks are only partially elastic. Due to the soundness requirement, we cannot add checks unless all valid target locations are marked with an ID.

In the case of BinKungfu using ID-based return checks from Figure 18, overhead breaks down as in Table 6. The table measures overhead as the increase in code size due to each type of instrumentation. Our benchmarks have 49 return instructions (roughly one per function) and 160 call sites on average. The large number of call sites means that 49% of the overhead comes from marking call sites with IDs.

We next describe how to avoid this up-front cost using a new elastic CFI check.

4.4.3.1 *An elastic CFI check*

We designed a new CFI check for return instructions that avoids overhead from instrumenting call sites. The insight is that we can often recognize a valid return target by examining the code around the target address. If we find that this is the case for all call sites of a given function, then we use a specialized check and do not modify the call sites.

Our check is displayed in Figure 19. It verifies that the target address is preceded by a direct call to the function to which the return instruction belongs. In X86 assembly, direct call instructions are of the form `0xe8 + offset`, where `0xe8` is the instruction opcode and `offset` is the offset between the current program counter (i.e., the location right after the call instruction) and the start of the function to be called.

On a function call, the processor performs the following actions: It first increments the program counter, so that it points to the instruction following the call instruction. Then, the value of the program counter is pushed on the stack, to serve as the return address for the

```
ret
```

(a) Original return instruction

```
            push ecx
            mov  ecx, [esp + 4]
            add  ecx, [ecx - 4]
            cmp  ecx, f.addr
            je   offset_ok
            call abort
offset_ok:  pop  ecx
            ret
```

(b) Return with check

Figure 19: Specialized check for directly-called functions. The check ensures that the instruction just before the return site looks like a call to the function at `f.addr`.

function call. Finally, the processor adds the `offset` to the program counter, and starts executing the function.

The check reproduces these actions in order to compute the expected offset in the `ecx` register. It first saves `ecx` on the stack. This places the old value of `ecx` at `[esp]`, and the return address at `[esp + 4]`. The instruction `mov ecx, [esp + 4]` loads the return address from the stack into the `ecx` register. `add ecx, [ecx - 4]` adds to this the four bytes preceding the return location, which—assuming the return location is valid—correspond to the `offset` part of the call instruction. Here, `ecx` is used both as destination and source operand of the addition, in order to clobber as few registers as possible. If the return address is legitimate, the result should be equal to the address of the callee function. In this case, the check restores `ecx` and `esp`, and allows the return to happen.

We name this check `retdirect`, because it verifies `ret` instructions for directly-called functions.

For efficiency, the check does not actually verify the opcode. Enforcing a correct offset is strong enough to eliminate most invalid target locations.

A return check for directly-called functions is fully elastic. At 22 bytes, the check itself has 10% more memory overhead than an ID-based check. Yet this is more than compensated by savings on IDs and by selective instrumentation, as our evaluation in Section 4.5.3 will show.

An ID-less `retdirect` check requires more information about the program's control-flow graph than ID-based checks. For the latter, it is enough to identify function call and return instructions. For `retdirect` checks, BinKungfu needs to identify function addresses and determine for each function whether it could ever be called indirectly. Compiler optimizations such as tail-call optimizations can make this difficult. In our evaluation, we found that BinKungfu's disassembler handled all benchmarks correctly, so that `retdirect` checks did not introduce functionality problems.

Finally, note that `retdirect` checks are more precise than the ID-based checks used by BinKungfu. The ID-based checks share the same ID across all call sites and returns. This approximation allows return instructions to target any call site, including those from different functions. In contrast, `retdirect` checks only allow returns to call sites of the right function.

4.4.3.2 *Selective instrumentation*

Elastic CFI checks enable selective instrumentation, but selectively instrumenting a program brings its own set of challenges. We will look at two challenges in particular: First, how to select a subset of checks with high expected protection? This task is difficult because it is unknown where in a program a control-flow hijack could occur, and thus which checks are needed. Second, how to make full use of the available overhead budget, without exceeding it? The main difficulty for this task is computing the overhead created by a given subset of checks.

SELECTING CHECKS We would like to prioritize checks to protect the program where it most needs protection. Whether a check is useful depends on whether it can catch the effects of a bug. Existing tools often use heuristics to estimate this. For example, GCC's stack protector [123] gives users the choice to either protect all functions or only those that use buffers (and thus are more prone to buffer overflows that the stack protector detects).

BinKungfu uses a rather simple heuristic to prioritize checks, based only on the check's type. BinKungfu first adds `retdirect` checks to return instructions in directly-called functions. If this does not use all the overhead budget, it also adds classic ID-based checks to all other functions. Finally, it adds CFI checks for other control transfer instructions, namely indirect calls.

This heuristic is based on the average cost of the checks, and on the observation that return address overwrites have historically been the most exploited type of control-flow hijack vulnerability.

CONTROLLING OVERHEAD The second challenge in selective instrumentation is to fully use the available overhead budget. This requires predicting the overhead due to a subset of changes, in order to determine it fits in the overhead budget. Such a prediction is difficult because the overhead for a set of changes is not simply the sum of overheads of its components. Non-linearity arises because each change interacts with other changes and can influence the layout of the entire program. We next explain how this phenomenon occurs in BinKungfu's approach to patching.

When BinKungfu patches a program, it leaves the binary mostly intact, and only modifies parts (i.e., basic blocks in the CFG) that need

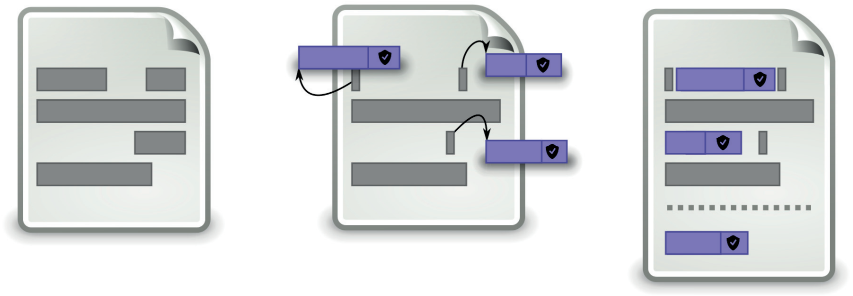


Figure 20: Instrumenting a program with BinKungfu. On the left is the original program. Grey blocks represent active code, with free spaces in-between, e.g., due to unused functions. The middle panel shows the program's state after BinKungfu replaced some blocks with jumps to instrumented code. The instrumented code is initially "floating", until BinKungfu assigns an address to each block (right panel). If not all blocks fit into unused space, BinKungfu allocates a new code section.

protection. This approach increases compatibility, because the functionality of unmodified program parts is preserved, even if BinKungfu does not perfectly understand those parts or did not perfectly reconstruct the CFG. However, this approach introduces constraints on the type of modifications that can be performed; in particular, BinKungfu does not move arbitrary program parts around. This preserves jumps, code pointers, and other functionality which expects program code to be in a fixed location.

When BinKungfu does decide to change a basic block, it generates a copy with equivalent functionality and added protection. Then, BinKungfu replaces the original block with a jump to the new code. The left two panels of Figure 20 illustrate this patching process.

The changed parts of the CFG are initially "floating", i.e., they don't have an assigned address. Further changes to these blocks do not introduce more jumps, because floating blocks are not subjected to the same constraints as regular blocks

Once BinKungfu has applied all changes to the CFG, it tries to pack the floating changes into the available space, as shown in the right panel of Figure 20. BinKungfu aims to minimize fragmentation, i.e., use the available space efficiently by matching floating blocks to available spaces of the appropriate size. It uses heuristics do so, because computing an optimal assignment is computationally hard. Only once this heuristic bin-packing completes does BinKungfu know the amount of overhead induced by the changes.

BinKungfu thus performs instrumentation in two distinct phases. During the first phase, the program consists only of original code and floating changes. In this phase, BinKungfu can modify the program, either by introducing a new floating change or modifying an existing one. In the second phase, BinKungfu establishes the layout of the instrumented program and commits the set of changes to its final form. At this point, modifications are no longer safe and efficient. Figure 21a contains pseudocode for this two-phase instrumentation process.

For selective instrumentation, BinKungfu needs to know the overhead caused by a subset of the available changes, in order to generate a program that satisfies the desired overhead limit. This means that BinKungfu needs to interleave the change generation and address assignment phases. We implemented this on top of BinKungfu's existing functionality; Figure 21b shows our implementation in pseudocode, and Figure 21c illustrates the difference compared to the original version.

We modified BinKungfu's original algorithm in two ways. First, we generate changes to the CFG in order of decreasing priority. Second, we run the address assignment after each step. If this assignment is successful, BinKungfu snapshots the resulting program executable, and then undoes the address assignment so that the CFG can be further modified.

Running and undoing the address assignment turned out to be surprisingly tricky. This highlights the trade-offs faced by software engineers between ease of implementation and extensibility, and between imperative and functional styles.

Originally, BinKungfu exploited the strict separation of change generation and address assignment. The change generation phase made assumptions about the structure of the program, and did not handle code that normal compilers are unlikely to create. For example, BinKungfu's CFG data structure could not represent IDs (like those introduced by the CFI protection mechanism) unless they were part of a floating change. These assumptions no longer held after the second phase, when the floating changes had been integrated into the program.

We explored several solutions to interleave the change generation and address assignment phases: (a) copying the CFG data structure and running the address assignment phase on a copy, or (b) predicting space usage without actually assigning addresses, or (c) implementing an undo functionality for address assignment. Choice (a) seemed simplest conceptually, but we had to rule it out because much code in BinKungfu depended on having a single CFG. Choice (b) required us to solve a difficult algorithmic problem. This left us with choice (c), which was conceptually rather ugly and inefficient. However, it worked, and working code is better than efficient or elegant code.

4.4.3.3 *Summary of elastic binary hardening*

In this section, we described how we used elastic instrumentation techniques with the BinKungfu binary hardening engine. The first

```
# Step 1: generate floating changes
while has_change(cfg):
  c = next_change(cfg)
  apply_change(cfg, c)

# Step 2: assign addresses and generate patched executable
assign_addresses(cfg)
result = finalize(cfg)
```

(a) BinKungfu's original instrumentation loop.

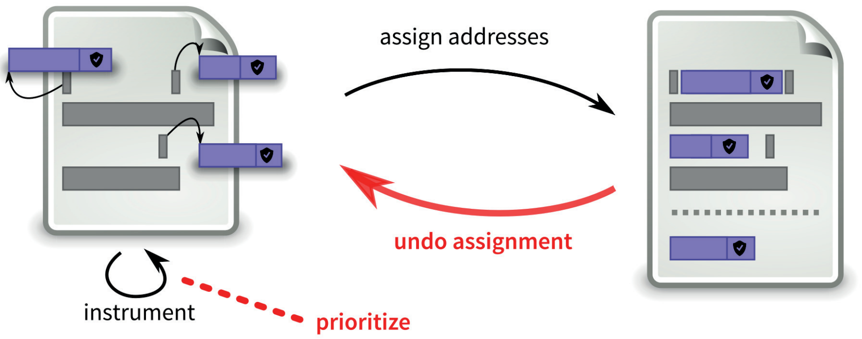
```
result = original_program

# Generate floating changes, ordered by priority
while has_change(cfg):
  c = next_change(cfg)
  apply_change(cfg, c)

  # Tentatively assign addresses and create snapshot
  assign_addresses(cfg)
  if fits_in_overhead_budget(cfg):
    result = finalize(cfg)

  # Make changes floating for further modifications
  undo_assign_addresses(cfg)
```

(b) BinKungfu with support for selective instrumentation.



(c) Summary of the steps performed by BinKungfu. Extra steps for selective instrumentation are drawn in bold red.

Figure 21: Selective instrumentation in BinKungfu.

component of our technique is an elastic check that can verify the integrity of returns from directly called functions. The novelty of this check is that it does not require any modifications to the function's call sites. The second component is an algorithm to apply checks selectively and determine, after each change to the program, whether the result satisfies the desired overhead limits.

We will show in Section 4.5.3 how each of these components contributes to instrumenting programs with tight overhead limits.

BinKungfu will be released as open-source software, available at `http://codejitsu.org/`.

4.5 EVALUATION OF ELASTIC INSTRUMENTATION

In this section, we evaluate ASAP, FUSS, and BinKungfu, the three techniques that use elastic instrumentation. There are common questions that we answer for each technique, although the exact methodology necessarily differs. In particular, we first look at how much elasticity there is to exploit, and whether there are fixed costs that limit elasticity. Next, we examine the impact of elastic instrumentation on raw performance, e.g., on instrumentation overhead for ASAP or test throughput for FUSS. Finally, we report end-to-end benefits. These are specific to each technique. We evaluate security subject to overhead constraints for ASAP, the time to find bugs for FUSS, and the number of programs that BinKungfu can instrument with limited code size.

4.5.1 *Results for ASAP*

In our evaluation, we want to know both how fast and how secure instrumented programs optimized by ASAP are. Any software protection mechanism needs to quantify its overhead and security. More specifically in the case of ASAP, we ask:

- *Effectiveness:* Can ASAP achieve high security for a given, low overhead budget? We show that ASAP, using existing instrumentation tools, can meet low overhead requirements, while retaining most security offered by those tools.
- *Performance:* How much can ASAP reduce the overhead of instrumentation on any given program? Does it correctly recognize and remove the expensive checks? What effect does the profiling workload have on performance?
- *Security:* Does ASAP in practice preserve the protection gained by instrumenting software? How many sanity checks can it safely remove without compromising security? We also analyze the distribution of both sanity checks and security vulnerabilities in software systems, and draw conclusions on the resulting security of instrumented programs.

4.5.1.1 *Metrics*

We quantify performance by measuring the runtime of both the instrumented program and an uninstrumented baseline and computing the *overhead*. Overhead is the additional runtime added by instrumentation, in percent of the baseline runtime.

To quantify the security of an instrumented program, we measure the *detection rate*, i.e., the fraction of all bugs and vulnerabilities that have been detected through instrumentation. The detection rate is relative to a known reference set of bugs and vulnerabilities (e.g., those detected by a fully instrumented program), because all bugs or vulnerabilities present in a particular software cannot be known in general.

In addition to these end-to-end metrics, we report the *sanity level* and *cost level* of optimized programs. The sanity level is the fraction of checks remaining in the program, i.e., the fraction of critical instructions that are protected. The cost level measures the fraction of instrumentation cost that ASAP preserves.

4.5.1.2 *Benchmarks and methodology*

We evaluated ASAP's performance and security on programs from the Phoronix and SPEC CPU2006 benchmarks, the OpenSSL cryptographic library, the Python 2.7 and 3.4 interpreters, and the FFMPEG media decoder.

For instrumenting the target programs, we used AddressSanitizer (ASan) and UndefinedBehaviorSanitizer (UBSan). Both are widely applicable. We also evaluated ASAP with ThreadSanitizer and SoftBound, but do not report these results because those instrumentation tools are applicable to fewer benchmarks. SoftBound is a research tool that unfortunately fails to compile many programs, and ThreadSanitizer is only meaningful for multithreaded applications.

We use a collection of real and synthetic bugs and vulnerabilities to quantify ASAP's effect on security. For Python and FFMPEG, we reintroduced existing bugs from previous versions of the software. We also measured ASAP's effect on detecting bugs in the RIPE benchmark (a program containing 850 variants of buffer overflow exploits), and we analyzed the entries in the CVE vulnerability database for the year 2014.

The paragraphs below give more information about each benchmark. However, we omit details on our hardware, compilation options etc; the interested reader who would like to reproduce our experiments can find all scripts in the ASAP source code, and detailed information in the ASAP conference paper [132].

SPEC CPU2006 BENCHMARKS The SPEC CPU2006 suite is a set of 19 benchmark programs written in C/C++. Each program comes with a *training workload* that we used for profiling, and a *reference workload*, approximately 10× larger, used for measuring overhead.

We used ASAP to create optimized executables for cost level 0.01, i.e., removing 99% of the estimated cost, to generate data in Figure 22a. We also generated executables for sanity levels between 80% and 100%, i.e., preserving that amount of checks. These results are shown in Figure 23.

Unfortunately, not all benchmarks are compatible with our instrumentation tools. We could run 14 out of 19 benchmarks for ASan and 12 out of 19 for UBSan. In most cases, this is because the benchmarks rely on undefined or implementation-specific features of the C/C++ language. Two cases, `Xalancbmk` and `dealII`, fail due to a bug in LLVM's cost model used by ASAP.

OPENSSL We compiled OpenSSL with ASan to protect it from the Heartbleed vulnerability. We use its test suite as profiling workload for ASAP. We report its overhead by measuring the throughput of OpenSSL's built-in web server when serving a 3 kB static HTML file over an encrypted connection.

PYTHON We compiled the Python 3.4 interpreter with ASan and UBSan instrumentation. We obtained profiling data by by running Python's unit test suite. We evaluated performance using the default benchmarks from the Grand Unified Python Benchmark Suite [55].

FFMPEG We test FFMPEG with ASan. We use the FATE test suite, a collection of media files in hundreds of formats, as profiling workload. We use a benchmark from OpenBenchmarking.org [111] to measure the performance of instrumented FFMPEG binaries. It measures the time taken to convert a video file (1:04 minutes, 290 MB, not part of FATE) from the h264 format to dvvideo.

4.5.1.3 *Performance results*

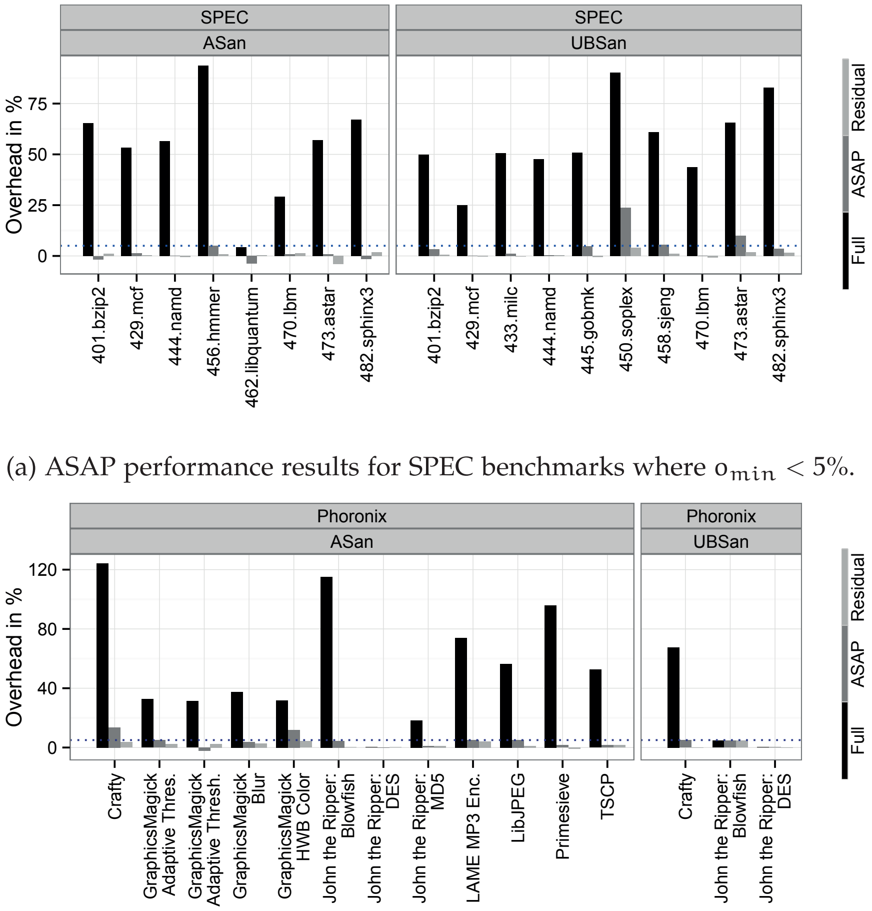
We report the cost of security, with and without ASAP, in Figure 22. For each benchmark, we display three values: The overhead of full instrumentation (leftmost, dark bars), the overhead with ASAP at cost level 0.01 (gray bar, center), and the residual overhead (light bars, right). This data reveals a number of results:

Full instrumentation is expensive. On SPEC, both AddressSanitizer and UndefinedBehaviorSanitizer typically cause above 50% overhead.

ASAP often reduces overhead to acceptable levels. For eight out of 14 SPEC benchmark, ASAP reduces ASan overhead to below 5%. This result is also achieved for seven out of twelve benchmarks with UBSan. For three UBSan benchmarks, the overhead at cost level 0.01 is slightly larger than 5%.

For the remaining benchmarks, ASAP gives no security benefits because they are not elastic: their residual overhead is larger than 5%. In this case, ASAP can only satisfy the overhead budget by producing an uninstrumented program.

We also report overheads for the benchmarks used in our security evaluation, in Table 7. None of these benchmarks is sufficiently elastic to have less than 5% residual overhead. From this, we have to conclude that there are many applications that are not sufficiently elastic for ASAP to be effective.



(a) ASAP performance results for SPEC benchmarks where $o_{min} < 5\%$.

(b) ASAP performance results for Phoronix benchmarks where $o_{min} < 5\%$.

Figure 22: Summary of ASAP performance results. For each benchmark, we show three values: The darkest bar represents overhead for full instrumentation. The next bar shows overhead with ASAP at cost level 0.01. The lightest bar show the residual overhead, i.e., overhead that is due to other factors than sanity checks. Only elastic benchmarks (with residual overhead <5%) are shown. ASAP brings the overhead of instrumentation close to the minimum overhead, while preserving a high level of security. For the benchmarks shown here, ASAP removes 95% of the overhead due to checks, and obtains an average sanity level of 87%.

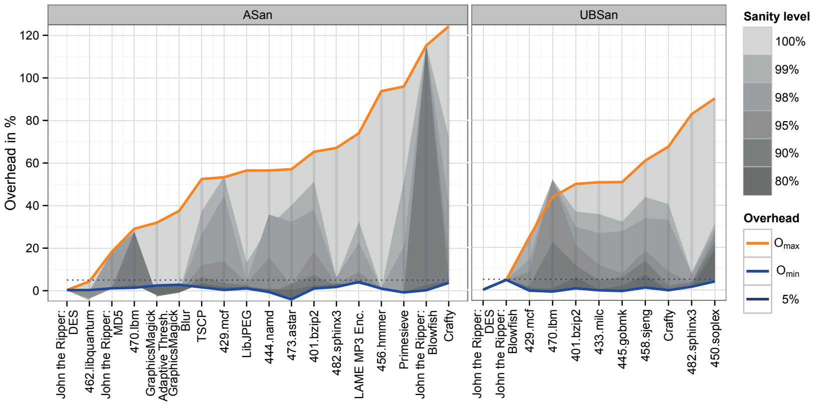


Figure 23: This graph shows the space of possible performance-security trade-offs. The orange line shows overhead of existing instrumentation tools; it averages at 54% for ASan and 45% for UBSan. ASAP can reduce this overhead down to the blue minimal overhead line. The shade of the area corresponds to the sanity level (darker = fewer checks). Reducing the sanity level by a small value has a large impact on overhead; for example, reducing the sanity level to 99% reduces overhead by 47% on average. There are a few cases where programs with more sanity checks are slightly faster than programs with fewer checks (e.g., libquantum with ASan or lbm with UBSan). This is due to the sometimes unpredictable effects of checks on caches, compiler heuristics, optimizations etc.

ASAP eliminates most overhead due to checks. In all cases except for `soplex`, the overhead at cost level 0.01 is very close to the residual overhead. Although many checks remain in the programs (87% on average for the benchmarks in Figure 22, generally more for larger programs such as Python), they do not influence performance much, because they are in cold code.

These results show that ASAP correctly identifies and removes the hot checks. We conclude that *ASAP's profiling is sufficiently good* to eliminate overhead. However, ASAP does not achieve a particular overhead budget very precisely. For the benchmarks in Figure 22, it removes 95% of the overhead due to checks, whereas it was expected to remove 99% with a cost level of 0.01. Users who need to precisely achieve a target overhead need to run ASAP iteratively to fine-tune the cost level.

Even small reductions in security lead to large performance gains. In Figure 23, we show the speedups obtained when reducing the sanity level step by step. The gray area corresponds to the entire security-performance space that ASAP can navigate. The lightest gray area, or 47% of the total overhead, can be eliminated by removing just 1% of the sanity checks. This shows how additional cycles invested into

	cost level		overhead [%]	
	c_{safe}	c_{rec}	full	at c_{rec}
OpenSSL	0.0080	0.01	16	7
Python 3.4 (ASan)	0.0050	0.01	172	81
Python 3.4 (UBSan)	—	0.01	66	7
FFMPEG	0.0850	0.01	62	14
RIPE	0.0004	0.01	—	—

Table 7: A summary of our security experiments. At cost level c_{safe}, ASAP prevents all known vulnerabilities. In addition, we report overheads for full instrumentation and for our recommended cost level c_{rec}, which provides a good trade-off between security and performance.

security give diminishing returns, and confirms that indeed only few checks are hot.

4.5.1.4 *Security evaluation*

Developers and operators who use ASAP need to know how safe the resulting programs are. In particular, we measure how ASAP affects the detection rate of software instrumentation: what is the chance that a bug or vulnerability that was previously prevented by instrumentation is present in a ASAP-optimized program?

The detection rate depends primarily on the sanity level, i.e., the fraction of critical instructions that are protected with sanity checks. Since the sanity level is directly determined by the cost level, we can find an overall minimum cost level at which all known vulnerabilities would have been caught. The following paragraphs present our results of case studies on the OpenSSL Heartbleed vulnerability, Python bugs, FFMPEG vulnerabilities, and the RIPE benchmark. They demonstrate that a cost level of 0.01 would have been sufficient to prevent most, but not all, vulnerabilities studied.

We summarize all our security results in Table 7, and provide information about individual experiments next.

OPENSSL HEARTBLEED The OpenSSL Heartbleed vulnerability is caused by a bug in OpenSSL that manifests when processing heartbeat messages. Such messages have a length field and a payload, the size of which is expected to match the length field. Yet, attackers can send a heartbeat message with a length field larger than the payload size. When constructing the reply, the server would copy the request payload to the response, *plus whatever data followed it in memory*, up to the requested length. This allows the attacker to read the server's memory, including sensitive data like passwords.

When compiling OpenSSL with ASan, ASan adds a bounds check to prevent this buffer over-read. When we profiled OpenSSL using its test suite as profiling input, that critical check was never executed. This is because heartbeat messages are an optional and rarely used feature of OpenSSL, and the test suite does not cover them. This means that ASAP estimates the cost of the critical check to be zero and will never remove it, regardless of the target overhead specified.

We extended the test suite with a test case for heartbeat messages. Now the cost of the critical check is non-zero, but there are 15,000 other more expensive checks accounting for 99.99% of the total cost. We can further increase the check's cost by using larger payloads for the heartbeat messages we test. With a payload of 4KB, still 99.2% of the cost is spent in more expensive checks. Thus, ASAP will preserve this sanity check for all cost levels larger than 0.008.

PYTHON The interpreter of the widely used Python scripting language consists of about 350kLOC of C code, 1,900 of them assertions. When compiled with ASan instrumentation, the interpreter binary contains 76,000 checks.

We analyzed three bugs, which we re-introduced into the Python 3.4 interpreter's source code from earlier versions: #10829, a buffer overflow in `printf`-style string formatting that ASan detects; #15229, an assertion failure due to an uninitialized object; and #20500, an assertion failure when an error occurs during shutdown. Reports for these bugs are available on the Python bug tracker at http://bugs.python.org/. The number of analyzed bugs is unfortunately limited, because we only evaluated bugs that satisfied the following criteria: (1) the bugs must be real-world problems recently reported on the Python issue tracker, (2) they must be detectable using instrumentation or assertions, and (3) they must be deterministically reproducible.

ASAP preserves the sanity checks that detect these bugs for all cost levels larger than 0.005. Section 4.5.1.5 contains a more detailed evaluation of sanity checks and bugs in Python.

FFMPEG We reproduced twelve security vulnerabilities in FFMPEG from 2013 and 2014. In these two years, 18 vulnerabilities have been reported to the CVE database and fixed by the developers. However, we could not reproduce six of them because they were in code that handled obscure media formats, or did not lead to detectable memory corruption. Out of twelve reproduced vulnerabilities, ASan detects memory corruption in eleven cases. In the remaining case, the memory corruption is limited to a single C struct, which can only be detected using a stronger mechanism than ASan.

For FFMPEG, ASAP only preserves security for cost levels larger than 0.085. In other words, the amount of overhead it can safely remove is significantly less than for our other benchmarks. If users choose a lower cost level like 0.01, ASAP does remove almost all the elastic parts of the overhead, but also removes the checks necessary to detect two out of the eleven detectable bugs.

In Section 4.5.1.5 below, we discuss this risk more thoroughly, and contrast sanity checks in FFMPEG with those in Python.

RIPE BENCHMARKS The RIPE benchmark suite [137] is a set of exploits for synthetic buffer overflows. It consists of a vulnerable program that attacks itself. In total, it features 850 unique attacks that differ in five characteristics: (1) the location of the buffer, e.g., on the stack or inside a structure; (2) the code pointer being overwritten, e.g., a return address; (3) whether the target pointer is overwritten directly or indirectly; (4) the type of shellcode; and (5) the function where the overflow happens, e.g., `memcpy` or `sprintf`.

The RIPE benchmark is well-known in the security community and contains a large number of exploits. However, its synthetic nature makes it problematic for evaluating ASAP: First, the exploits are all very similar; they differ only in few aspects of their construction, so that the number of effectively different scenarios is much smaller than 850. In particular, there are only ten distinct program locations where a memory corruption happens, so that the security gained by instrumentation is based on only ten sanity checks. Second, RIPE is designed for the sole purpose of overflowing buffers. There is no relevant workload that could be used for profiling. For the lack of an alternative, we exercised all the different overflow mechanisms to obtain profiling data. Third, RIPE makes strong assumptions about the compiler and the operating systems. Many exploits depend on the order of objects in memory, or on particular pointer values. Small changes in compilation settings or even between different runs of a program can cause such assumptions to fail; this makes it difficult to compare benchmarks.

For these reasons, we do not evaluate individual exploits in detail, and solely measure the minimal cost level needed to preserve the protection against buffer overflows gained by ASan instrumentation. ASAP preserves all critical sanity checks inserted by ASan for cost levels larger than 0.0004. Furthermore, nine out of ten buffer overflows happen inside library functions such as `memcpy`, which ASan redirects to its safe runtime library. Checks in ASan's runtime library are part of the residual overhead that ASAP does not yet address. ASAP currently preserves these checks at all cost levels.

SECURITY EVALUATION SUMMARY In our case studies on OpenSSL, Python, FFMPEG, and RIPE, we determined the minimum cost level to protect against all known vulnerabilities to be 0.008, 0.005, 0.085, and 0.0004, respectively.

Our default cost level is 0.01, which provides a good trade-off between security and performance. It corresponds to a sanity level of

94% in OpenSSL, 92% in Python, and 93% in FFMPEG. This fraction of checks prevents all vulnerabilities in OpenSSL, Python and RIPE, but causes ASAP to miss two bugs in FFMPEG. This means that a cost level that works well for one type of software does unfortunately not generalize to other software. Users of ASAP should analyze the result, e.g., by examining the elided checks as described in Section 4.2.3.2.

4.5.1.5 *Discussion of sanity checks*

To understand the security effect of ASAP, it is helpful to analyze the properties of sanity checks that are removed and preserved, respectively.

We first consider the 100 most expensive sanity checks in the Python interpreter. These checks together account for 29% of the total cost. They are in hot core locations of the interpreter: 49 of them belong to core Python data structures such as maps or tuples; 23 are in the main interpreter loop; 22 are in reference counting and garbage collection code; and 6 in other parts of the interpreter. Any meaningful Python program exercises the code where these checks reside. A bug in these parts of the interpreter would likely affect many Python scripts and thus be immediately detected. Hence we are confident that removing these checks in production is safe. The Python developers seem to partially agree with this: six out of these 100 checks are assertions in code regions that are only compiled when `Py_DEBUG` is defined, i.e., only during development.

In contrast, the checks that guard real-world bugs are executed rarely. The bugs in our case study are executed only (i) when a format string contains the `"%%"` character sequence, (ii) when a Python script circumvents the usual constructors and directly executes `__new__`, or (iii) when an error is raised during interpreter shutdown. We did not select the bugs to be particularly rare—it just happens to be that many real-world bugs *are* tricky corner cases.

Figure 24 sheds further light on this issue. For this graph, we looked at the checks in Python 2.7, and differentiate between checks that are located in buggy code, and “normal” checks. We take as buggy code those parts of the source code that have received bug fixes between the time Python 2.7 was released, until the current version 2.7.8.

We find that checks in buggy code are executed less frequently than regular checks. This makes them less likely to be affected by ASAP. For example, at cost level 0.01, ASAP removes 8% of all checks, but only 5% of the checks in buggy code. If we assume that our notion of buggy code is representative, we can conclude that the sanity level as computed by ASAP (92% in this case, for a cost level of 0.01) is a lower bound on the fraction of bugs that are protected by checks (95% in this case). This follows from the fact that the dark line is always below the bright line in Figure 24.

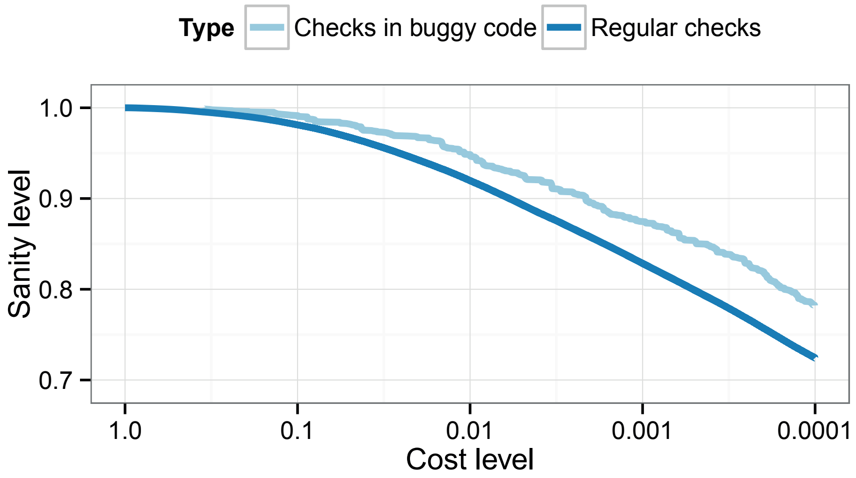


Figure 24: Fraction of checks preserved by ASAP in the Python interpreter, for various cost levels. The dark line corresponds to the sanity level as computed by ASAP. The bright line corresponds to the fraction of protected buggy code. Because checks in buggy code have a lower cost on average than regular checks, they are more likely to be preserved.

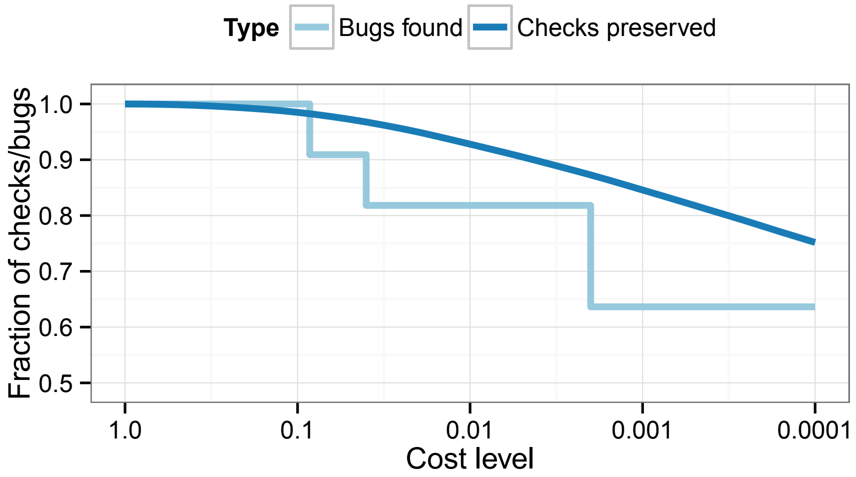


Figure 25: Fraction of preserved checks and bugs found in FFMPEG, for various cost levels. For FFMPEG, the actual effectiveness of instrumentation (bugs caught) drops below the expected effectiveness (the sanity level). Our intuition that bugs are in cold code only does not hold in this case.

Contrast this with Figure 25, where we plot the effectiveness of selective instrumentation and the sanity level for FFMPEG. In this case, we find that the sanity level does not provide a lower bound on the actual effectiveness. Indeed, two out of eleven vulnerabilities in FFMPEG can only be detected by checks that are relatively hot, and are only preserved when the cost level is >0.085.

We do not have a good solution for this, because bugs and vulnerabilities are by nature unpredictable. Our experiments show that there *are* a few bugs in hot code, so using ASAP does reduce security. The computed sanity level gives developers an estimate of this reduction. Although some uncertainty remains, this estimate allows them to make informed choices regarding the best trade-off between security and performance.

4.5.1.6 *CVE vulnerability survey*

We complete our security evaluation by studying known security vulnerabilities from the CVE database [105]. We focus on memory-related vulnerabilities because sanity checks are particularly promising for protecting against this category.

The CVE data set contains 879 memory-related vulnerabilities for the year 2014. For 180 of these, it was possible to obtain the source code and patch that fixed the vulnerability. From the source code and patch, we determined the location of the memory error itself. The error is not always located in the patched program part. For example, a common pattern is that developers add a missing check to reject invalid input. In this case, we searched for the location where the program accesses the illegal input and corrupts its memory. For 145 vulnerabilities, we could tell with sufficient certainty where the memory error happens.

We then manually analyzed the bugs to determine whether they lie in hot or cold parts of the program. We used four criteria to classify a code region as cold: (1) the code does not lie inside loops or recursively called functions, (2) the code is only run during initialization or shutdown, (3) comments indicate that the code is rarely used, and (4) the code is adjacent to much hotter regions which would dominate the overall runtime. In absence of these criteria, we classified a code region as hot.

Overall, we found 24 vulnerabilities that potentially lie in hot code regions. The other 121 (83%) lie in cold code where ASAP would not affect checks protecting against them. Because our criteria for cold code are strict, we think this is a conservative estimate. It provides further evidence that a large fraction of vulnerabilities could be prevented by applying instrumentation and sanity checks only to cold code areas.

The results of our CVE study are publicly available and can be accessed at `http://dslab.epfl.ch/proj/asap`.

4.5.2 *Results for FUSS*

In this section, we answer the following questions about FUSS:

- Does FUSS reduce the amount of instrumentation being executed? We count the number of instrumentation atoms executed per testcase, and show that FUSS reduces that count by 88% on average.
- How does FUSS affect test quantity? We compare number of executions per second of fuzzers and their optimized versions, and find that FUSS increases throughput by 1.5× on average.
- How does FUSS affect test quality? We find that tests generated by FUSS-optimized fuzzers match or exceed the coverage obtained by regular fuzzers. In addition, optimized fuzzers reach a given coverage target faster because they generate tests at a higher rate.
- Does FUSS help developers find bugs faster? We evaluate FUSS using 12 real-world bugs that were found through fuzzing. FUSS is up to 3.2× faster at finding them than the state of the art.

4.5.2.1 *Experimental setup*

We evaluate FUSS on 11 benchmarks taken from existing fuzzing projects. Our first source is Google's fuzzer-test-suite [54], a set of programs that have been subjected to fuzzing by Google and others. We take from this suite all benchmarks that have reproducible crashes (as of December 2016); there are nine benchmarks with a total of ten bugs. Many of them are security-critical libraries embedded into web browsers (e.g., the harfbuzz font rendering engine and the c-ares DNS library), servers (e.g., the OpenSSL cryptographic suite) and other software (e.g., the sqlite database engine).

We also add two benchmarks selected from the Fuzzing Project website [25]: http-parser and file. In the latter, we found two new bugs while testing FUSS. We include these bugs in our time-to-bug experiments. We also reported them to the developers of file, who promptly fixed the bugs.

We compile a our benchmarks using full optimization (-O3). We also enable debug information generation (-g compiler flag) because it is required for profiling and helps to identify bugs when a fuzzer finds a crash. This does not affect the program's performance, but may increase compilation and linking time.

All our experiments use LibFuzzer [88] as fuzzing engine. We slightly modified its output format to gather data for time and coverage plots, and added a benchmarking mode to measure test throughput for a given corpus. We did not change any other part.

We run our experiments on two types of server machines. Type (a) has a 40-core Intel Xeon processor @ 3.0 GHz and 256 GB RAM, and

	Baseline	FUSS	reduction
boringssl	1,697	87	94.9%
c-ares	85	14	83.0%
file	62,660	12,388	80.2%
harfbuzz	28,969	1,870	93.5%
http-parser	83	4	95.3%
libxml2	2,769	196	92.9%
openssl	162,003	38,898	76.0%
pcre	46,682	2,745	94.1%
re2	82,359	12,644	84.6%
sqlite	19,769	1,555	92.1%
woff2	35,623	4,243	88.1%

Table 8: Number of instrumentation atoms executed per testcase, with and without FUSS. On average over all benchmarks, FUSS reduces instrumentation cost by 88%.

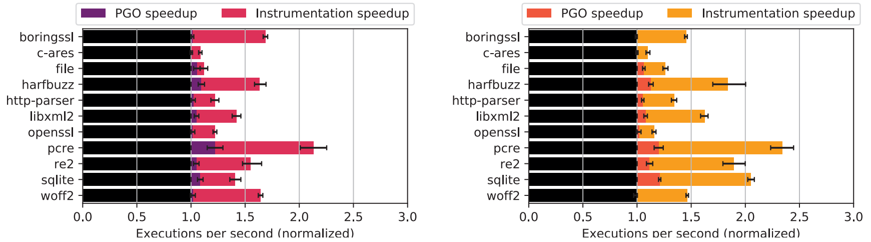
type (b) has a 56-core Intel Xeon processor @ 2.6 GHz and 256 GB RAM. Fuzzing is a CPU-intensive process: our longest experiment requires about 12×40 CPU-days to complete. This means that using powerful machines and parallelization is unavoidable. We take care to run each individual experiment on the same machine type, and usually report values that are normalized to the baseline, so that the different machine speeds do not affect our results. In addition, we run each experiment multiple times and report 95% confidence intervals for all our metrics.

4.5.2.2 *FUSS reduces instrumentation cost*

In this section, we analyze the effect of FUSS at the level of instrumentation atoms. We focus exclusively on coverage instrumentation, because this allows us to measure the effect of FUSS in isolation and exclude noise from other parts of the program under test.

We measure the effect of FUSS using a modified form of coverage instrumentation which, in addition to doing its regular work, increments a global counter each time an atom is executed. We report the average count per testcase, for each of our benchmarks, in Table 8. FUSS reduces that number by 88% on average.

We are interested in these numbers because FUSS can only remove instrumentation from the well-explored core of the program, and has to preserve it in those program areas that have not yet been thoroughly explored. If the removed atoms account for a sufficiently high fraction of the total executions, then FUSS can have an impact on the speed of fuzzers.



(a) Coverage and sanity checks.

(b) Coverage instrumentation only.

Figure 26: Increase in fuzzing speed when using FUSS, normalized to the baseline that does not use FUSS. Speedups are due to better code layout (PGO) and removing expensive instrumentation atoms. On average, FUSS increases speed by 1.44× for programs instrumented with both safety checks and coverage, and 1.55× when using only coverage instrumentation. The difference arises because coverage instrumentation has fewer fixed overheads than safety checks.

For three benchmarks, the optimized programs still execute more than 10,000 instrumentation atoms per testcase. This is due to three main reasons:

First, FUSS may underestimate an atom's cost. Factors such as the low resolution of profile data or code motion during compilation may cause this. Second, the fuzzer may discover some expensive atoms only after FUSS has optimized the program. Third, FUSS decides to keep atoms based on a conservative cost threshold, which does not take the specific nature of each benchmark into account.

4.5.2.3 *FUSS improves fuzzing speed*

In this section, we examine how FUSS affects the raw speed of fuzzers. The metric we consider here is executions per second, i.e., how many inputs the fuzzer can test per second. We first compile a baseline version of the benchmarks, then use the warmup, profiling, and optimization steps to generate a FUSS-optimized version. We run both versions in benchmarking mode for 60 seconds on a single core, on a snapshot of the corpus generated during warmup and profiling. In benchmarking mode, the fuzzers do not react to feedback from the program, and thus keep the corpus fixed. This ensures both the baseline and the optimized version execute inputs that are identically distributed. Each experiment is repeated 20 times. Figures 26a and 26b show the results.

During 60 seconds, a fuzzer executes between 20,000 and 60 million tests, depending on the benchmark. We normalize these numbers, reporting the ratio of the number obtained by the optimized fuzzer and the baseline.

We run two variants of this experiment with different type of instrumentation. The first uses coverage instrumentation and memory sanity checks. This is a common configuration for fuzzing, and the one we use in our bug-finding experiments. Its instrumentation causes an average slowdown of 2.8×, but not all of it is elastic, i.e., due to instrumentation atoms themselves. FUSS cannot remove non-elastic overhead that is due to changes to the memory allocator, cache effects, etc.

The second variant uses coverage instrumentation only. Compared to the version with sanity checks, the slowdown is "only" 1.6×, but FUSS can remove a bigger fraction of it because almost all of it is elastic.

Overall, the geometric mean speedup achieved by FUSS is 1.44× for instrumentation with sanity checks, and 1.55× for pure coverage instrumentation.

When compiling optimized programs, the compiler also makes use of the profiling data gathered by FUSS for other means, e.g., to lay out basic blocks for optimal branch prediction. The PGO bars in Figure 26 isolate this effect: they plot the speed of a program compiled with profile-guided optimizations but full instrumentation. The plot shows that most of the speedup obtained using FUSS is in fact due to optimized instrumentation, rather than other effects.

4.5.2.4 *FUSS explores programs faster*

In this section, we are interested in how FUSS affects the quality of tests generated by the fuzzer. Given that FUSS reduces the amount of information that is gathered while fuzzing, there is a legitimate concern that this would make fuzzers unable to generate high-quality tests.

We measure both the absolute coverage obtained by the fuzzer, as well as the rate at which it discovers new coverage. This is a proxy for test quality. The idea is that reduced test quality would prevent the fuzzer from making progress, and appear in our results as reduced total coverage or higher time to obtain a given coverage level.

Figure 27 plots the coverage obtained for all our benchmarks during a 4-hour fuzzing session. We show averages over 50 experiments, the area around the line being a 95% confidence interval. The data varies by benchmark: the smallest benchmarks tend to reach a coverage plateau, whereas the fuzzer keeps finding new basic blocks for larger benchmarks. The FUSS-optimized fuzzer discovers blocks at a higher rate than its regular counterpart. For benchmarks where coverage keeps growing, FUSS tends to keep ahead of the baseline over the entire experiment.

Figure 28 supports these findings. The figure shows the rate at which new basic blocks are being discovered, measured during the first 60 seconds after generating the optimized program, and normalized against a baseline fuzzer that starts from the same state. FUSS-

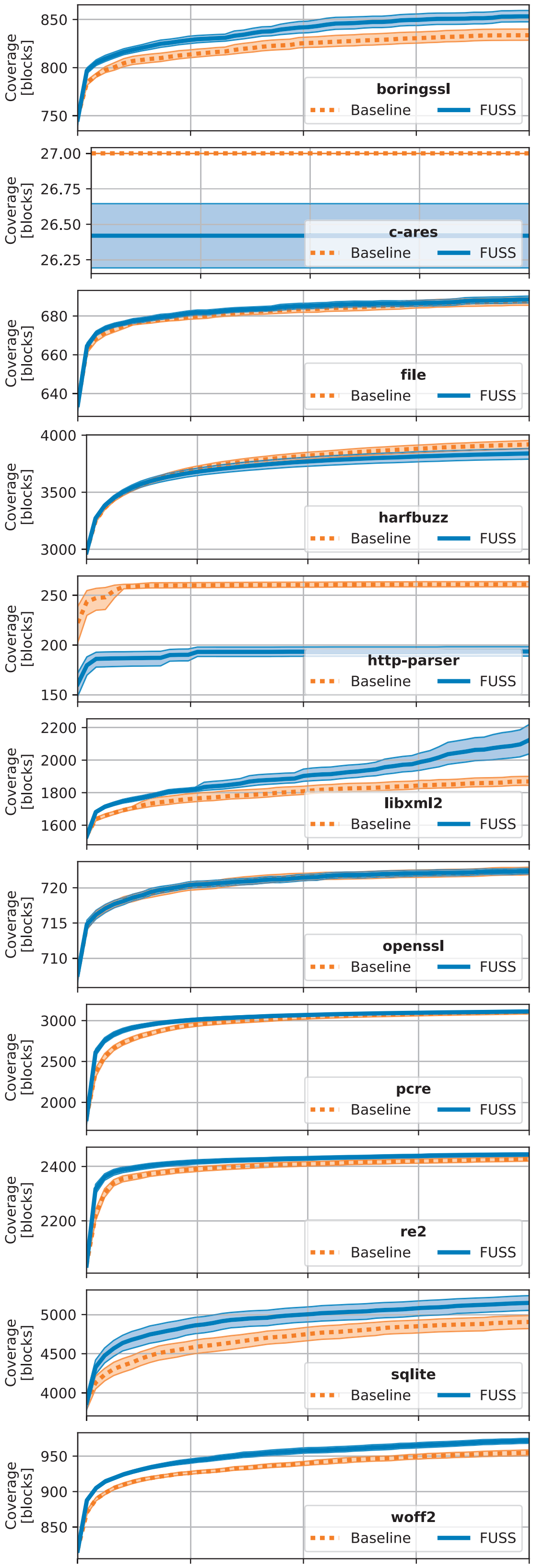
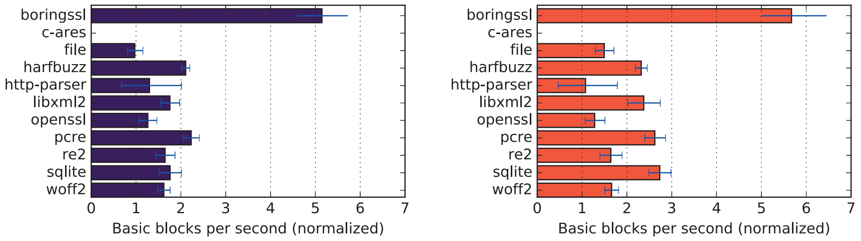


Figure 27: Coverage vs. time for all our benchmarks, when fuzzing for four hours.



(a) Coverage and sanity checks. (b) Coverage instrumentation only.

Figure 28: The rate at which new basic blocks are discovered when using FUSS, normalized relative to the baseline. The geometric mean increases are 1.8× when using sanity checks and 2.0× when using coverage instrumentation only.

optimized fuzzers are on average 2× faster than regular fuzzers, with no signs that that they are blocked from making progress.

Two benchmarks run against this trend and deserve special analysis. Http-parser is the one case where FUSS removes too much instrumentation, so that the fuzzer plateaus at 190 basic blocks, whereas the baseline discovers 260 basic blocks. The cause of this problem is a bad choice in threshold. In fact, this is the smallest of our benchmarks in terms of total code size, containing only 848 instrumentation atoms. During profiling, many of these atoms receive a large share of CPU time, and thus exceed our fixed cost threshold.

The second benchmark is c-ares, which has few explored blocks and suffers from a similar problem. In addition, the fuzzer reaches its maximum coverage almost immediately, and does not discover any new block during the measurement phase. This is why Figure 28 does not show a bar for c-ares.

FUSS can have a surprisingly large effect on the basic block discovery rate. Notably, for boringssl, the effect shown in Figure 28 exceeds the speedup shown in Figure 26. We investigated this, and found that FUSS prevents the fuzzer from generating certain slow testcases. These testcases increase coverage in program parts that are already well explored, but they do not make a difference elsewhere in the program. In fact, the coverage increase comes solely from executing a different number of iterations in an already explored loop. Because FUSS removes instrumentation from expensive loops, the fuzzer no longer considers such testcases interesting, and instead explores other program areas faster.

4.5.2.5 *FUSS finds bugs faster*

Bugs are the gold standard by which fuzzer performance is evaluated. In this section, we examine whether FUSS provides an actual end-to-end benefit for developers. Our metric is the time to reproduce known bugs. We measure this time for all ten crash bugs from

Google's fuzzer-test-suite and two bugs in file. Table 9 reports our results.

We measure the time until the fuzzer finds the crash caused by the target bug, for both a FUSS-optimized fuzzer and the baseline. This time varies by benchmark, from few seconds to many CPU-days. We compensate for this variation by selecting an appropriate timeout and number of cores for each benchmarks. We run benchmarks that terminate quickly on a single core, to maximize their duration and thus minimize relative measurement error. For benchmarks that take a long time, we choose an appropriate number of cores such that most runs terminate within one hour, and increase the timeout once we have used all cores on our machine. Table 9 reports the number of cores for each benchmark.

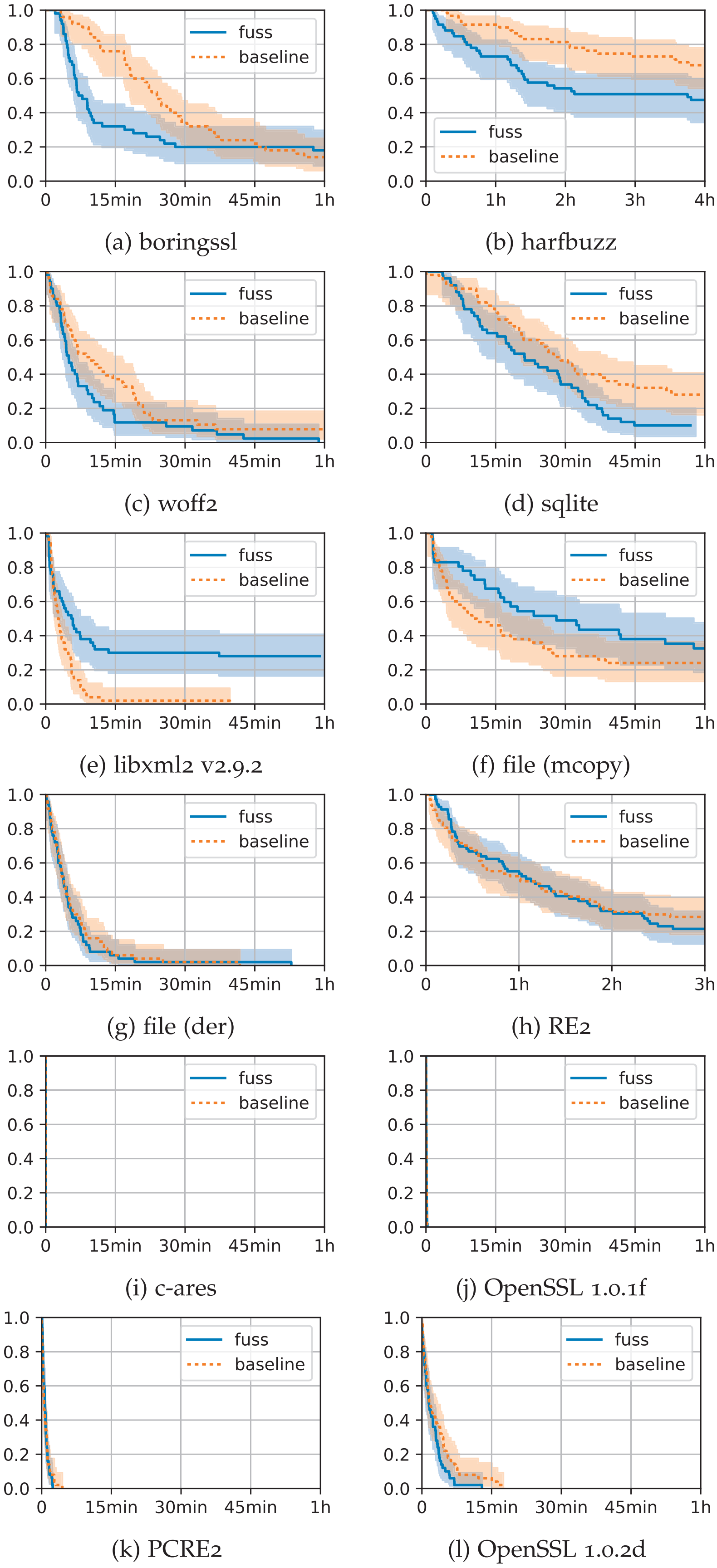
We also report in Table 9 the time to recompile and optimize benchmarks. This time is spent by FUSS unless the bug is found before the warmup and profiling phase have completed. We take recompilation time into account when computing speedups. However, it is not shown in Figure 29, which plots FUSS's effect on fuzzing time only.

Even for a single benchmark, the time to find a bug is highly variable. This is because fuzzing is a random process. More precisely: (1) Testcases are generated randomly, and thus vary in length and execution time. (2) Because new testcases are generated by mutating existing ones, the quality of the first testcases has a large and persisting influence on fuzzing progress. (3) The fuzzer discovers new parts of the program in random order, and so it might spend time exploring bug-free program areas before reaching the buggy code. We carefully handle this variation by repeating each experiment 50 times, showing confidence intervals in our plots, and being conservative when reporting speedups.

Figure 29 presents the time to discover the twelve bugs we study. More precisely, this figure plots the *survival function* for each bug. The survival function S is the probability that a bug will "survive" (i.e., evade discovery) for a given time. Its value is 1.0 initially, and descends to 0.0 as time increases. The median time to find the bug is the time when the survival function intersects the horizontal line at $s = 0.5$. The area below the curve is proportional to the mean time to find the bug.

To understand our results, we find it helpful to group the twelve bugs into four categories:

1. *FUSS is clearly faster.* This category includes boringssl, harfbuzz, and woff2. (a-c in Figure 29). FUSS is also faster for sqlite, but the speedup is not significant at a 95% confidence level.
2. *FUSS is clearly slower.* This happens for libxml2 and one of the bugs in file (Figure 29, e and f). These results are important for understanding the limitations of our technique, and we explain them below.



(a) boringssl

(b) harfbuzz

(c) woff2

(d) sqlite

(e) libxml2 v2.9.2

(f) file (mcopy)

(g) file (der)

(h) RE2

(i) c-ares

(j) OpenSSL 1.0.1f

(k) PCRE2

(l) OpenSSL 1.0.2d

Figure 29: Bug survival times. Each curve $S(t)$ shows the probability of a bug to resist discovery until time t. For example, the median time to find the bug in boringssl is 7 minutes for FUSS, vs. 24 for the baseline.

			FUSS				Baseline			
Benchmark	Bug	Cores	$t_{compile}$	t_{bug}	CI		t_{bug}	CI		Speedup
boringssl	use after free	14	21	424	367	597	1,435	1,089	1,744	3.22
harfbuzz	assertion failure	38	26	13,508	5,040	∞	∞	∞	∞	~ 3.00
woff2	buffer overflow	1	11	297	222	412	521	329	992	1.69
sqlite	out of memory	14	212	1,272	789	1,731	1,650	1,231	2,254	
libxml2	buffer overflow	1	80	331	177	577	159	113	214	0.39
file (mcopy)	buffer over-read	38	15	1,684	937	3,458	598	302	1,335	0.35
file (der)	buffer over-read	1	40	229	160	290	233	176	310	
RE2	buffer overflow	54	6	4,064	2,827	5,907	3,656	2,062	6,345	
c-ares	buffer overflow	1	-	1	1	1	1	1	1	
OpenSSL (1.0.1f)	Heartbleed	1	-	6	4	7	6	4	7	
PCRE2	buffer overflow	1	-	42	26	50	31	20	50	
OpenSSL (1.0.2d)	assertion failure	1	-	88	54	143	93	65	188	

Table 9: The 12 bugs we analyze. $t_{compile}$ is the time for FUSS's recompilation step. t_{bug} is the median time to find the bug, with confidence interval, reported for FUSS and LibFuzzer. All times are in seconds.

3. *FUSS is about as fast as the baseline.* RE2 and one of the bugs in file fall in this category (Fig. 29, g-h).
4. *We can't tell.* This category contains those benchmarks for which the bug is too easy to find, namely c-ares, OpenSSL, and PCRE2 (Figure 29, i-l).

For the benchmarks in category (4), we obtain no information about FUSS because the bugs are found too quickly. The fuzzers discover them within seconds, often during the warm-up or profiling phase before FUSS starts its optimization. We decided to nevertheless report these results to avoid any bias due to benchmark selection. Also, we use these benchmarks in other parts of our evaluation, to measure other metrics like throughput and coverage.

For three out of twelve benchmarks, FUSS produces speedups between 1.7× and 3.2×, when comparing the median time to find bugs as reported in Table 9.

In two cases, fuzzers produced by FUSS clearly take more time to find a bug than the baseline. This shows the technical limitations of FUSS, and according to our analysis is not due to a fundamental problem with the approach. We found that in these cases, FUSS makes poor choices regarding which instrumentation atoms to remove. This seems to be due to problems in mapping profiling data to instrumentation atoms, which leads to wrong estimate of the atoms' cost.

4.5.3 *Results for elastic binary hardening*

In this section, we answer the following questions about BinKungfu:

- How large is BinKungfu's overhead and where does overhead come from? We measure increase in a program's code size due to CFI instrumentation, and break that number down by instruction type.
- How many opportunities are there for elastic instrumentation? We analyze how prevalent directly-called functions are, and thus how many return checks could be made elastic.
- What are the overhead savings due to specialized return checks?
- What are the end-to-end benefits of selective instrumentation? We count the number of programs for which we achieve our goal: CFI protection with less than 5% memory overhead.

4.5.3.1 *Experimental methodology*

We developed and evaluated BinKungfu in the context of the DARPA Cyber Grand Challenge (CGC) [40]. This competition in binary hardening and exploitation influenced our definition of acceptable overhead and is the source of our benchmark programs.

	Avg. count	size [B]	total
indirect calls	4	15	60
indirect call targets	3	9	27
returns	49	20	980
return targets	160	6	960

Table 10: Types of instrumentation atoms in BinKungfu with their count and code size in bytes, averaged over all benchmarks.

We evaluated BinKungfu on 154 benchmark programs that were provided as examples to CGC contestants, or used in the CGC qualifying event. These benchmarks are written in C or C++ and range in size from about 200 to 8,000 lines of code.

To measure code size, we report both the size of instrumentation code in isolation, and also the total size of BinKungfu-generated code. The latter metric includes code size increases due to the patching mechanism, e.g., jumps to divert control flow to instrumented parts of the program. We exclude from these measurements twelve benchmarks that could not be instrumented, e.g., because the disassembler could not obtain a CFG. We also exclude one outlier, a benchmark designed to stress-test the instrumenter using a large amount of automatically generated code.

To measure memory usage, we run a benchmark on the reference workload provided by DARPA and measure the benchmark's resident set size, i.e., the space usage of all virtual memory pages which the program needs to load in memory to perform its work efficiently.

4.5.3.2 *Sources of overhead*

BinKungfu's CFI implementation protects programs from control-flow hijack attacks by protecting indirect control flow transfers. Table 10 reports the number of return instructions and indirect call instructions that need protection, on average for our benchmarks. In addition, we report the number of target locations for these instructions that need to be marked with an ID when using ID-based CFI. We estimate the space needed for this instrumentation based on the typical size of each type of instrumentation atom.

On average, most of the instrumentation code size is due to return checks and IDs. The reason for the low number of indirect calls and targets is that these are only used by few benchmarks, most notably those written in an object-oriented style.

163 out of 216 instrumentation atoms are IDs. These account for almost half of the size of instrumentation code. This shows that elastic instrumentation techniques that can reduce the need for IDs have the potential to significantly reduce overhead.

	direct	indirect	mixed	total
Functions	47	2	4	53
Percent of total	88%	4%	8%	100%

Table 11: Number of functions in our benchmarks, broken down by how they are called.

Our benchmarks also contain a third type of indirect control-flow transfer, not shown in Table 10: indirect `jmp` instructions. The compiler generates these when translating a `switch` statement. We found that the compiler-generated code was always safe, i.e., the compiler enforced that the `jmp` could only target the switch cases. This is why BinKungfu did not add additional checks to these instructions.

4.5.3.3 *Opportunities for elastic instrumentation*

BinKungfu can use specialized checks for functions that are guaranteed to be directly called. Table 11 shows how often this is the case. We report the average number of functions in our benchmarks, classified by how they are called. Overall, 88% of all functions are always called directly, and thus amenable to optimization using specialized return checks.

Note: the total number of functions in our benchmarks is slightly higher than the number of checked return instructions. This is because our benchmarks contain hand-optimized assembly code that does not follow the "one return per function" rule. For example, `sin`, `sinl`, and `sinf` are related functions that all compute the sine of a floating point number and share a single return instruction.

4.5.3.4 *Benefits of elastic return checks*

Table 12 compares traditional ID-based CFI and our variant using `retdirect` checks for returns in directly-called functions. We want to highlight three points.

First, BinKungfu can use specialized ID-less `retdirect` checks for 87% of all return instructions. Second, this reduces the number of return targets that need to be instrumented by 97%. Third, the overall amount of code that BinKungfu generates goes down from 2.5kB to 704B. This number does not only include return checks, but also all other CFI checks and all the hooking code needed to patch these checks into the executable. In other words, using `retdirect` checks reduces the amount of BinKungfu-generated code by 71%.

4.5.3.5 *Benefits of selective instrumentation*

Reducing code size is only part of the solution to achieve a given overhead target. Perhaps more importantly than reducing code size,

	checks			
	ID-based	`retdirect`	target IDs	code size [B]
baseline	49.1	0	159.9	2,489
elastic	6.5	42.7	4.6	704

Table 12: Comparison of ID-based CFI versus a variant that uses specialized, elastic checks. The table shows the number of checks of each type added, the number of call sites instrumented with target IDs, and the total size of added code in bytes.

elastic checks enable selective instrumentation. This is important because memory overhead is determined by the number of 4kB pages required to hold the program's code and data. Thus, reducing code size by 1,785 bytes may or may not matter depending on whether this changes the amount of pages needed. Selective instrumentation can adjust the amount of instrumentation code to ensure that the program's code fits into a desired multiple of 4kB pages.

Without selective instrumentation, BinKungfu uses too many memory pages, and thus causes more than 5% memory overhead, for 17 out of our 154 benchmarks. For the remaining 137, it either has low enough overhead (124 cases) or causes the program to crash due to other, unrelated, problems (13 cases).

With selective instrumentation, BinKungfu can partially protect 10 out of these 17 programs, and achieve the overhead target. The protection offered by BinKungfu varies. In 60% of the cases, it can only add `retdirect` to some, but not all, directly-called functions. In the remaining 40%, it can protect all returns in directly-called functions, and also add some ID-based checks for other indirect control-flow transfers.

To summarize, selective instrumentation reduces the number of cases where BinKungfu is too expensive from 17 to 7.

4.6 DISCUSSION, LIMITATIONS AND NEGATIVE RESULTS

This section discusses topics and limitations that we have encountered when working with elastic instrumentation. First, we discuss two points that have influenced the design of our techniques: our desire for building practical tools (Section 4.6.1) and the observation that heuristics are often more useful than proofs (Section 4.6.2). After that, we discuss five limitations of our work: We show how limitations of profiling techniques affect elastic instrumentation in Section 4.6.3. We describe problems that arise when classifying instrumentation atoms solely by their cost in Section 4.6.4. We discuss residual overhead in Section 4.6.5, the complex effects of small program changes in Section 4.6.6, and our reliance on compile-time decisions in Section 4.6.7.

4.6.1 *The quest for practical tools*

The work in this thesis is focused on practicality. This influenced many of our design decisions:

We wanted our techniques to be applicable to real-world, unmodified systems software and a large number of off-the-shelf instrumentation tools. This is made possible in part by integrating ASAP and FUSS in the LLVM compiler infrastructure. LLVM is a robust framework that compiles large code bases effortlessly, and many instrumentation tools use it as a basis. The one limitation arising from this choice is that ASAP and FUSS require access to the source code of the programs they work with. Because this is also the case for the supported instrumentation tools and fuzzers, we consider this acceptable.

We chose not to use some advanced and expensive compiler techniques which would have reduced the practicality of our tools. For example, we do not use link-time optimization (LTO). All our tools process programs one compilation unit at a time and don't require whole-program analysis. This reduces our ability to reason about instrumentation (e.g., we need to assume that code from different compilation units could influence instrumentation atoms) but makes it possible to use our tools with larger benchmarks.

Our tools became easier to use over time. When we designed FUSS, we chose to use a different profiling mechanism than for ASAP because this simplified the FUSS workflow. We considered that the increase in usability outweighed the lower quality of profiling data.

To prove the practicality of our tools and allow other researchers to reproduce our results, we publish all our tools as open source.

4.6.2 *Heuristics, not proofs*

Throughout our work with elastic instrumentation, we chose heuristics over proofs, and we are willing to forgo formal guarantees. All our tools use program instrumentation selectively, subject to performance constraints, and make no guarantee that the selected instrumentation is sufficient for the user's purpose. Instead, we rely on empirical evaluations to show that selective instrumentation is effective in practice, for the benchmarks we test.

This approach makes sense when working with instrumentation. Instrumentation is a run-time technique used precisely where static analysis falls short. Consider memory safety: for many memory accesses, static analysis can neither prove that the access is safe nor demonstrate the presence of a bug. The consequences of this uncertainty are that developers use run-time checks, and that reasoning about run-time checks is hard.

ASAP historically grew out of a project that used sound static analysis in tandem with run-time checks to ensure memory safety. The project used static analysis as a means for improving performance, by eliminating checks that were provably safe. To estimate the benefits of this project, we wanted to measure the performance gains for a hypothetical perfect static analysis that allowed to remove all checks. In other words, we measured the elasticity of the run-time checks, although we did not call it so at the time.

During these experiments, we realized that performance gains were concentrated around hot code regions, and that proofs for checks in cold regions would not matter much. Moreover, there were only few checks in hot regions. The risk incurred by removing them without proof seemed small relative to the performance gains. Thus, the idea for ASAP came to be.

The idea to forgo proofs turned out to drastically simplify the project. Suddenly, we could work with real-world benchmarks that had been out of scope for static analyzers. We could drop the need for whole-program analysis. There was no more reliance on annotations or constraints that burdened developers. We replaced SMT solvers with a simple greedy strategy to select checks, which could be explained in minutes and executed in milliseconds.

We believe that these benefits outweigh the lack of formal guarantees.

4.6.3 *Obtaining accurate instrumentation cost*

Our tools primarily use cost as a metric to identify the most valuable instrumentation atoms. ASAP and FUSS obtain this cost through profiling, but use different mechanisms with different limitations.

4.6.3.1 *Limitations of profiling techniques*

ASAP uses GCOV-style profiling: it modifies the program to be profiled, adding a counter to each edge in the program's control flow graph. ASAP then executes the program with a special profiling workload. From the data obtained by the counters, ASAP can compute exactly how often each instruction in the program was executed during profiling.

Profiling data obtained this way is precise, but comes at three costs: First, instrumenting the program for profiling requires an extra compilation step. This complicates and slows down the build process. Moreover, the program used for profiling must have exactly the same structure as the program that is later optimized (modulo the counters), otherwise one cannot tell which counter value corresponds to which program point. ASAP's compiler wrapper script is very careful to ensure this.

Second, the program that is being profiled is not the same as the program that is later optimized. Profiling instrumentation itself may affect the performance of surrounding code, e.g., because it occupies instruction caches. These effects are not visible in the execution counts, and make ASAP's cost estimate less accurate.

Third, ASAP needs a special profiling workload to exercise the program, since it cannot use the profiling-instrumented program in production. This workload must be chosen carefully, so that the instrumentation atoms that are executed frequently are those that are most expensive in production, and contribute little to security.

FUSS avoids all these problems by using a statistical profiler. It requires no extra compilation steps and can obtain profiling data "in production", i.e., while the fuzzer performs its work.

The downside is that profiling data obtained by FUSS is less precise, due to two limitations of statistical sampling profilers. First, such profilers have a limited sampling frequency. They identify costly parts of the program by examining the CPU state about 4,000 times per second. As a result, the profiler can only see instrumentation atoms that occupy the CPU at least once when a sample is taken.

Second, the raw profiling data refers to addresses in the executable binary file, which must be mapped to instrumentation atoms. FUSS uses a complex set of transformations to load profiling data and convert samples to atom costs, and each transformation potentially loses precision. Furthermore, FUSS relies on debug information to obtain the atom's location in the source code and to count the number of profiling samples that fall on this location. This makes it hard for FUSS to distinguish atoms that lie on the same source line. This hurts particularly in the presence of macros: in C/C++, the expansion of a macro is done by the preprocessor and leads to many instructions that share the same source location, namely the point where the macro was invoked.

The limitations of profiling techniques have hurt both ASAP and FUSS. For ASAP, we communicated with researchers who were interested in using the open-source tool, but found it difficult to use due to the multiple compilation steps. For FUSS, our evaluation identified several cases where more precise atom costs would have led to a faster program, and one case where FUSS removed an important instrumentation atom because it overestimated its cost.

4.6.3.2 *Potential solutions*

For ASAP, an effective way to obtain good profiling data is to use a program's test suite as profiling workload. If the test suite exercises all performance-sensitive program areas, it will usually be sufficient to uncover the expensive instrumentation atoms. If the test suite is thorough, covers corner-cases, and carefully audits the program's state, then it provides assurance that the program parts which are stressed contain fewer bugs.

This introduces an interesting way for developers to guide ASAP: Code that is well-tested becomes faster, because ASAP can remove more instrumentation atoms in these areas. ASAP instead places instrumentation in less-tested program areas, where it is more likely to prevent vulnerabilities.

For FUSS, we can overcome precision limitations of profiling data through better debug information. One known solution is to use so-called discriminators, i.e., identifiers that FUSS can assign to instructions to distinguish multiple statements that share the same source line.

Finally, static cost estimation techniques like those proposed by Ball and Larus [13] and the work of Tim Wagner et al. [133] might be able to complement imprecise profiling data.

4.6.4 *Optimizing multiple metrics simultaneously*

All techniques proposed in this thesis are limited to using a single metric to prioritize instrumentation atoms: ASAP and FUSS measure cost in CPU cycles via profiling, and BinKungfu considers an instrumentation atom's cost to be proportional to its code size.

4.6.4.1 *Limitations of cost-based optimization*

Cost is not the only metric for classifying atoms. In particular, we think it would be beneficial to consider an atom's *benefit*. The idea is that particularly useful atoms are worth keeping even if their cost is higher than the cost of other atoms. This situation arises in two cases.

First, our techniques might have prior information about atoms that lets us estimate their usefulness. For example, they could consider a memory safety check more useful in a function that processes user-controlled input than in a function that is invoked on trusted data. There is a wide range of literature in software engineering that predicts the location and density of defects in code [102, 108, 144], and such models could estimate the benefit of checks according to its location.

Second, our techniques can be used with multiple types of instrumentation atoms simultaneously. Assuming that these all have the same benefit might be too strong a simplification. For example, FUSS targets fuzzers that use both coverage instrumentation and sanity checks. We might be able to assign a utility to each type, and optimize the overall utility function, as suggested by Candea and Fox [28].

4.6.4.2 *Potential solutions*

In our work, we did not find a good way to incorporate other metrics than cost. This is a negative result. However, we found two reasons why this might not be that strong a limitation.

The first reason is that large variations in cost dominate small differences in benefit. Atom cost has a high dynamic range: in our benchmarks, the most expensive atoms consume billions of CPU cycles, many orders of magnitude more than cheap atoms (Section 4.5.1.3). Thus, for benefit to make a difference, a benefit metric would need to be strong and identify atoms that are orders of magnitude more useful than others. We do not know of such a strong metric.

The second reason is that an atom's benefit often negatively correlates with its cost. This makes cost a stronger metric, because it identifies not only cheap atoms, but also those that have a higher probability of being useful. We found this relationship both in ASAP and FUSS.

For ASAP, we found evidence that bugs and vulnerabilities tend to be in cold (i.e., rarely executed) code. Concretely, we found in Section 4.5.1 that:

- 83% of the public, open-source memory errors from 2014 in the CVE database are in cold code.
- Locations of bug fixes between CPython versions 2.7.0 and 2.7.8 tend to be cold.
- All four bugs in our Heartbleed and CPython case studies are in cold code.

Third-party studies also found many bugs in cold code, e.g., [140].

However, not all bugs follow this pattern. In our study of twelve security vulnerabilities in FFMPEG, we found two vulnerabilities that could not be detected by partially instrumented programs with checks in cold code only.

For FUSS, we found that the cost of atoms predicted the amount of information that a fuzzer can gain from the atom, and the value of further exploring that atom (Section 4.5.2). Low-cost atoms have a long remaining lifetime during which they influence fuzzer behavior. In addition, they lie in program regions that are not yet well explored, where a fuzzer can still make progress quickly.

To summarize, cost is a metric that has a dominating influence for elastic instrumentation. There is additional but weak support for the hypothesis that low-cost atoms have high benefit. Ultimately, we need to accept that we can hardly predict benefit, and that the safest way to maximize it is to pay the price of complete instrumentation.

4.6.5 *Metadata and dependencies make instrumentation less elastic*

Fixed costs limit the elasticity of program instrumentation. For an instrumentation technique to be elastic, it needs to be fine-grained and cost-proportional. Instrumentation techniques that have both fixed and variable costs do not satisfy the cost-proportionality requirement. This limits the effectiveness of our elastic instrumentation techniques, because they can only eliminate the variable parts of the overhead.

We explained the reasons for fixed overheads in sections 4.2.2 and 4.4.2. Here is a summary of those overheads for the instrumentation tools we support:

- Memory safety tools require book-keeping whenever objects are allocated or freed in memory. The amount of work varies by tool. For example, SoftBound creates four words of metadata per `malloc`/`free`. AddressSanitizer performs a fixed amount of work per `malloc`, but a call to `free` requires work proportional to the size of the freed object.

- AddressSanitizer places freed objects in a "quarantine" to detect use-after-free bugs. This prevents freed space from being immediately reused, which reduces data locality and the effectiveness of caches.

- ThreadSanitizer incurs costs to track synchronization operations. These costs are necessary for data race checks to work correctly, and have to be paid irrespective of the number of checks.

- Techniques like control-flow integrity change the program's layout, usually in ways that increase code size and memory usage.

These fixed costs limit the maximum speed-up of selective instrumentation. In addition, the fixed overheads alone might be higher than users are willing to accept. For ASAP, this happened for benchmarks that perform many memory allocations (like the GCC and Povray SPEC benchmarks, which have over 50% fixed overhead).

4.6.5.1 *Potential solutions*

It is often possible to redesign instrumentation tools to be more elastic. This needs a change of mindset: with techniques like ASAP, it is no longer the overall overhead of instrumentation tools that matters, but the residual overhead that remains when all instrumentation atoms have been removed. This is the topic of Section 4.4, where we redesigned control-flow integrity instrumentation with elasticity in mind.

4.6.6 *Small program changes can have complex effects*

In the course of this work, we learned that small changes to a program can have surprisingly complex effects. Although individual instrumentation atoms are small, there are complex interactions during compilation and execution of the program that make it hard to predict exactly the effect of adding/removing an atom.

For example, compilers use heuristics to decide when to inline a function. Removing an atom from a function may bring the function size below the compiler's threshold for inlining. Inlining the function in turn affects the structure of other parts of the program, with unpredictable results.

This phenomenon limited our ability to predict the overhead of selectively instrumented programs. As a result, when evaluating ASAP we did not consider how precisely ASAP achieved a target overhead, only whether the overhead was within the user's budget.

A potential solution is to make our techniques easier to use in an iterative manner. For example, ASAP could accept feedback on the effective overheads observed in production, and then automatically choose a better performance/reliability trade-off.

4.6.7 *Elastic instrumentation requires static decisions*

The techniques presented in this thesis are limited to static selection of instrumentation atoms, done at compile-time or when the program is instrumented. It could be advantageous to defer these decisions to the time when the program is executed, because at that time more information is available about the workload. This might even make it possible for programs to adjust the amount of instrumentation dynamically, and reduce the need for profiling.

4.6.7.1 *Potential solutions*

Self-modifying code: We experimented with an approach inspired by the Linux `ftrace` function tracer [117]. It replaces instrumentation atoms with a `nop` instruction, i.e., an instruction that has no effect on the program. At run-time, our technique converted these instructions into jumps to the actual instrumentation atom. This approach failed because the large number of `nop` instructions reduced program performance, and because the instrumentation atoms themselves had to do extra work to save/restore CPU registers and resume program execution.

Dynamic binary translation: Frameworks such as PIN [90] and DynamoRIO [22] can modify programs at run-time and insert extra instrumentation code. Alas, these tools also have an overhead of their own, so that we were not satisfied with the resulting speedups.

Run-time activation: Banabic [16] developed in collaboration with us an approach to activate instrumentation at run-time. Key to its efficiency is to activate instrumentation atoms in batches, so that the CPU cycles spent in the activation logic are amortized over multiple instrumentation atoms.

To achieve this, the technique generates two versions of every function in the program, one instrumented and the other without instrumentation. It adds a dispatcher to each function that can choose either variant depending on a variable that tracks the current need for instrumentation. The program updates this variable at opportune times depending on the workload and the current overhead.

Banabic's technique has limitations of its own (e.g., generating two versions of every function roughly doubles the size of executables). However, we think that it could make the elastic instrumentation approach even more general.

4.6.8 *Summary of limitations*

Systems software is big, out of reach of many formal reasoning techniques, and highly performance-sensitive. The first set of limitations presented in this section can be traced back to our focus on systems software: It mandates that we use modular and scalable transformation techniques in order to handle big systems, and that we restrict our techniques to the level of program analysis that is typically used in compilers. As a result, our techniques come with no formal guarantees, but explicitly show where trade-offs are made and what combinations of reliability/performance are feasible.

The second set of limitations relates to the difficulty of quantifying reliability and performance. We find that our techniques can not in general estimate the benefit, in terms of reliability gain, of a program transformation. We can however quantify a transformation's cost, subject to limitations that arise from imprecise profiling data. We discussed to what extent this is sufficient, and how better approaches to profiling could improve our techniques. We closed this section by discussing a design for our techniques that does not need profiling at all, and instead adjusts the amount of program transformations at run-time.

4.7 SUMMARY

This chapter presented and evaluated elastic instrumentation techniques, i.e., techniques that control the amount of instrumentation code in a program in order to reach a favorable trade-off between reliability and performance. We developed three techniques for working with three types of instrumentation: ASAP works with safety checks that protect programs against illegal behavior, FUSS works with profiling instrumentation that makes fuzz testing more effective, and BinKungfu works with checks that verify control-flow integrity for binary code.

With ASAP, developers can prioritize safety checks according to their cost, and generate a program that as safe as possible given the developer's overhead budget. FUSS maximizes the effectiveness of coverage-guided fuzzers, by choosing a subset of instrumentation that allows for both high throughput and high test quality. BinKungfu aims to pack the highest number of control-flow integrity checks into a program with limited code size.

Each technique benefits from elasticity in its own way. ASAP, exploiting the Pareto Principle, can fit 87% of the available instrumentation into a 5% overhead budget. FUSS, for most benchmarks, obtains all instrumentation benefits at 12% of the cost of full instrumentation. BinKungfu's elastic checks can provide partial protection in cases where full instrumentation is too expensive.

Our results show that elasticity is beneficial in a large number of use cases. We hope that the scalability of instrumentation will in time become known as the Elasticity Principle, and serve future developers working with instrumentation.

Part III

WRAPPING UP

Where we look toward future work, and yet conclude this thesis.

5 ONGOING AND FUTURE WORK

> One day ladies will take their computers for walks in the park and tell each other "My little computer said such a funny thing this morning!"
>
> Alan Turing, 1951[1]

This thesis presented four ways to use elastic program transformations to obtain a better performance/reliability trade-off for existing systems software. To see the full impact of these techniques, we would like to keep working in the following areas: getting our techniques beyond prototype stage, increasing interoperability between tools that transform programs, and increasing automation.

As a first step, we would like to get our tools beyond prototype stage. A tool's impact is to a large extent driven by how usable and well-known it is. For example, we believe that AFL found so many bugs not only due to technical excellence, but also because AFL is very easy to apply. What could happen if a technique like FUSS were fully integrated into AFL, and could be used simply by enabling a `--fuss` command-line argument?

We also believe that there is a large opportunity in improving interoperability between tools that use program transformations. The LLVM project made compiler transformations available to a wide audience in academia and industry. Its intermediate representation (IR) is a language through which different tools can cooperatively work with programs. We would welcome similar common languages and standards in a wider ecosystem: standardized build systems; formats to represent profiling data, program specifications, and program analysis results; as well as common benchmarks to evaluate tools.

One area where there is particular need for a common representation is the analysis and transformation of program binaries. Techniques that work without access to source code face a difficult task, but are also more generally applicable. The DARPA Cyber Grand Challenge, an automated hacking competition, has renewed interest in testing, analyzing and transforming binaries. A range of techniques, e.g., symbolic execution and fuzzing, were popular among all the seven finalist teams. Yet each team relied on custom-built frameworks, particularly to defend programs against attacks. The tools available for that purpose (e.g., BinKungfu) are new, and the transformations they offer are primitive compared to program trans-

[1] According to a TIME article [56]; I have not been able to find the original source.

formations that are possible inside a compiler. Despite that, these tools show the potential of binary transformations.

Finally, we see a lot of potential in increased automation. Here again, the Cyber Grand Challenge offers a glimpse of what is possible. It shows systems that automatically, within seconds, discover and fix problems in software. In contrast, program transformations today are used rather statically, and require extra work from developers. In the future, opportunities for program transformations will likely grow, as techniques like containers, continuous integration, and testing as a service create new environments where software is being run. We envision that, in this future, developers can just write software and let tools automatically make it as reliable as possible, for each environment where the software is used.

6 CONCLUSIONS

Program transformations are powerful techniques that can significantly improve the speed and reliability of software systems, and our ability to test and verify them. They are particularly useful because the same program operates in many different environments, such as running on a production server or being tested during a nightly build. Program transformations can specialize a program to fit the requirements of these environments as closely as possible.

In this thesis, we use program transformations in an elastic way: We apply them at fine granularity to carefully selected parts of the software. This balances conflicting requirements of fast execution, small software size, ease of testing and verification, protection against attacks, and reliability. Moreover, because of Pareto's Principle of diminishing returns, elastic transformations can often obtain a large amount of the desirable properties at low cost.

This thesis presents four applications where we use program transformations in a fine-grained way to achieve better trade-offs between reliability and speed:

- We present -Overify, the first technique to compile programs for faster verification rather than fast execution. -Overify identifies transformations that are beneficial for verification. It modifies the compiler to use such transformations whenever possible and avoid others that are harmful for verification. This speeds up verification by up to 95×, and 58% on average.
- We present ASAP, a technique that enables the use of off-the-shelf program transformation tools under tight performance constraints. ASAP elastically adjusts the amount of safety checks that these tools add to a program. By carefully selecting cheap checks, it can often achieve a protection level of >80% with only 5% overhead.
- We developed FUSS, a technique that boosts the performance of common coverage-guided fuzzers. It selectively instruments a program under test, adding just enough instrumentation to catch bugs and direct the fuzzer to interesting program parts. Selective instrumentation requires only 12% of the CPU time of full instrumentation, and its use can reduce the time to find bugs by up to 3.2×.
- We introduce a novel safety check that prevents return-oriented programming attacks in binaries. It has no up-front overhead, and its cost scales linearly with the number of protected program locations. Using this check in the BinKungfu project re-

duced the amount of code needed for full protection by 71%, and its elasticity means that programs with tight limits on code size can at least be partially protected.

We hope that the work in this thesis would increase the joy of creating software systems, as machines help developers to get the most out of their software automatically, for each use case. We also hope that affordable hardening, better testing, and verification would make software systems more worthy of our trust.

BIBLIOGRAPHY

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow Integrity." In: *Conf. on Computer and Communication Security*. 2005.

[2] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. *Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software*. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html.

[3] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. "Preventing memory error exploits with WIT." In: *IEEE Symp. on Security and Privacy*. 2008.

[4] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. "Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors." In: *USENIX Annual Technical Conf.* 2009.

[5] Android official blog. *An Update to Nexus Devices*. https://android.googleblog.com/2015/08/an-update-to-nexus-devices.html. 2015.

[6] Matthew Arnold and Barbara G. Ryder. "A Framework for Reducing the Cost of Instrumented Code." In: *Intl. Conf. on Programming Language Design and Implem.* 2001.

[7] Abhishek Arya and Cris Neckar. *Fuzzing for Security*. https://blog.chromium.org/2012/04/fuzzing-for-security.html. 2012.

[8] Todd M Austin, Scott E Breach, and Gurindar S Sohi. "Efficient Detection of all Pointer and Array Access Errors." In: *Intl. Conf. on Programming Language Design and Implem.* 1994.

[9] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. "Enhancing Symbolic Execution with Veritesting." In: *Intl. Conf. on Software Engineering*. 2014.

[10] Domagoj Babic and Alan J. Hu. "Calysto: scalable and precise extended static checking." In: *Intl. Conf. on Software Engineering*. 2008.

[11] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. "The FORTRAN Automatic Coding System." In: *Western Joint Computer Conference: Techniques for Reliability*. 1957.

[12] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. “Thorough Static Analysis of Device Drivers.” In: *ACM EuroSys European Conf. on Computer Systems*. 2006.

[13] Thomas Ball and James R. Larus. “Branch Prediction for Free.” In: *Intl. Conf. on Programming Language Design and Implem.* 1993.

[14] Thomas Ball and James R. Larus. “Optimally Profiling and Tracing Programs.” In: *ACM Transactions on Programming Languages and Systems* (1994).

[15] Steve Ballmer. *Connecting with Customers*. https://www.microsoft.com/mscorp/execmail/2002/10-02customers.mspx. 2002.

[16] Radu Banabic. “Techniques for Identifying Elusive Corner-Case Bugs in Systems Software.” PhD thesis. EPFL, 2015.

[17] Adam Barth, Collin Jackson, Charles Reis, et al. *The security architecture of the Chromium browser*. Tech. rep. 2008.

[18] Emery D Berger and Benjamin G Zorn. “DieHard: probabilistic memory safety for unsafe languages.” In: *Intl. Conf. on Programming Language Design and Implem.* 2006.

[19] Hans-J. Boehm. “How to miscompile programs with "benign" data races.” In: *USENIX Workshop on Hot Topics in Parallelism*. 2011.

[20] Hans-J. Boehm and Sarita V. Adve. “Foundations of the C++ concurrency memory model.” In: *Intl. Conf. on Programming Language Design and Implem.* 2008.

[21] Ella Bounimova, Patrica Godefroid, and David Molnar. *Billions and Billions of Constraints: Whitebox Fuzz Testing in Production*. Tech. rep. MSR-TR-2012-55. Microsoft Research, 2012.

[22] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. “An Infrastructure for Adaptive Dynamic Optimization.” In: *Intl. Symp. on Code Generation and Optimization*. 2003.

[23] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. “Parallel Symbolic Execution for Automated Real-World Software Testing.” In: *ACM EuroSys European Conf. on Computer Systems*. 2011.

[24] David R. Butenhof. *Programming with POSIX Threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-63392-2.

[25] Hanno Böck. *The Fuzzing Project*. https://fuzzing-project.org/.

[26] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *Symp. on Operating Sys. Design and Implem.* 2008.

[27] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: Automatically Generating Inputs of Death." In: *Conf. on Computer and Communication Security*. 2006.

[28] George Candea and Armando Fox. "A Utility-Centered Approach to Building Dependable Infrastructure Services." In: *ACM SIGOPS European Workshop*. 2002.

[29] Scott A Carr and Mathias Payer. "DataShield: Configurable Data Confidentiality and Integrity." In: *Conf. on Computer and Communication Security*. 2017.

[30] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. "Unleashing Mayhem on Binary Code." In: *IEEE Symp. on Security and Privacy*. 2012.

[31] Pohua P. Chang, Scott A. Mahlke, and Wen-Mei W. Hwu. "Using profile information to assist classic code optimizations." In: *Software: Practice and Experience* (1991).

[32] Dehao Chen, David Xinliang Li, and Tipp Moseley. "AutoFDO: automatic feedback-directed optimization for warehouse-scale applications." In: *Intl. Symp. on Code Generation and Optimization*. 2016.

[33] Trishul M. Chilimbi and Matthias Hauswirth. "Low-overhead memory leak detection using adaptive statistical profiling." In: *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2004.

[34] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems." In: *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2011.

[35] Chrome Releases. *Stable Channel Update for Chrome OS*. `https://chromereleases.googleblog.com/2014/03/stable-channel-update-for-chrome-os_14.html`. Announcing Geohot's Pwnium 4 exploit.

[36] Clang User's Manual. *Undefined Behavior Sanitizer*. `http://clang.llvm.org/docs/UsersManual.html`.

[37] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs." In: *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. 2004.

[38] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C Necula. "Dependent types for low-level programming." In: *European Symp. on Programming*. 2007.

[39] John Criswell, Nathan Dautenhahn, and Vikram Adve. "KCoFI: Complete control-flow integrity for commodity operating system kernels." In: *IEEE Symp. on Security and Privacy*. 2014.

[40] DARPA. *Cyber Grand Challenge*. https://www.cybergrandchallenge.com/.

[41] Alan M. Davis. *201 Principles of Software Development*. 1995.

[42] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store." In: *Symp. on Operating Systems Principles*. 2007.

[43] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. "SAFECode: enforcing alias analysis for weakly typed languages." In: *Intl. Conf. on Programming Language Design and Implem.* 2006.

[44] Edsger W Dijkstra. *The threats to computing science*. https://www.cs.utexas.edu/~EWD/transcriptions/EWD08xx/EWD898.html. Speech at the ACM South Central Regional Conference. 1984.

[45] Isil Dillig, Thomas Dillig, and Alex Aiken. "Sound, Complete and Scalable Path-Sensitive Analysis." In: *Intl. Conf. on Programming Language Design and Implem.* 2008.

[46] Hiroaki Etoh and Kunikazu Yoda. *GCC extension for protecting applications from stack-smashing attacks*. 2000.

[47] *FindBugs – Find Bugs in Java Programs*. http://findbugs.sourceforge.net/.

[48] Matthew Finifter, Devdatta Akhawe, and David Wagner. "An Empirical Study of Vulnerability Rewards Programs." In: *USENIX Security Symp.* 2013.

[49] *GCC coverage testing tool*. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html. 2010.

[50] P. Godefroid, N. Klarlund, and K. Sen. "DART: Directed Automated Random Testing." In: *Intl. Conf. on Programming Language Design and Implem.* 2005.

[51] Patrice Godefroid. "Compositional Dynamic Test Generation." In: *Symp. on Principles of Programming Languages*. 2007.

[52] Patrice Godefroid, Michael Y. Levin, and David Molnar. "Automated Whitebox Fuzz Testing." In: *Network and Distributed System Security Symp.* 2008.

[53] Peter Goodman. *GRR: Granary Record/Replay*. https://github.com/trailofbits/grr.

[54] *google/fuzzer-test-suite*. https://github.com/google/fuzzer-test-suite.

[55] *Grand Unified Python Benchmark Suite*. https://hg.python.org/benchmarks/.

[56] Paul Gray. "Computer Scientist: Alan Turing." In: *TIME* (Mar. 1999).

[57] Andrew Griffiths. *High-Performance Binary-Only Instrumentation for AFL-Fuzz*. https://groups.google.com/d/topic/afl-users/lgrWdosb_ps.

[58] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. "Profile-Guided Compiler Optimizations." In: *The Compiler Design Handbook: Optimizations and Machine Code Generation*. 2002.

[59] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. "Defining the Undefinedness of C." In: *Intl. Conf. on Programming Language Design and Implem.* 2015.

[60] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. "Ironclad Apps: End-to-End Security via Automated Full-System Verification." In: *Symp. on Operating Sys. Design and Implem.* 2014.

[61] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. "Software Verification with BLAST." In: *Intl. SPIN Workshop*. 2003.

[62] Michael Hicks. *What is memory safety?* http://www.pl-enthusiast.net/2014/07/21/memory-safety/. 2014.

[63] Martin Hirzel and Trishul Chilimbi. "Bursty tracing: A framework for low-overhead temporal profiling." In: *ACM Workshop on Feedback-Directed and Dynamic Optimization*. 2001.

[64] Charles Antony Richard Hoare. "Assertions: A personal perspective." In: *Software pioneers*. Springer, 2002, pp. 356–366.

[65] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. "Profile-guided Automated Software Diversity." In: *Intl. Symp. on Code Generation and Optimization*. 2013.

[66] Galen Hunt and James Larus. "Singularity: Rethinking the Software Stack." In: *Operating Systems Review* 41.2 (2007).

[67] Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*. http://download-software.intel.com/sites/default/files/319433-015.pdf. 2013.

[68] *ISO/IEC 14882:2011: Information technology – Programming languages – C++*. International Organization for Standardization. 2011.

[69] *ISO/IEC 9899:2011: Information technology – Programming languages – C*. International Organization for Standardization. 2011.

[70] Jakub Jelinek. *FORTIFY_SOURCE: Object size checking to prevent (some) buffer overflows*. https://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html.

[71] Yaoqi Jia, Zheng Leong Chua, Hong Hu, Shuo Chen, Prateek Saxena, and Zhenkai Liang. "The Web/Local Boundary Is Fuzzy: A Security Study of Chrome's Process-based Sandboxing." In: *Conf. on Computer and Communication Security*. 2016.

[72] Richard WM Jones and Paul HJ Kelly. "Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs." In: *Intl. Workshop on Automated Debugging*. 1997.

[73] Joseph M Juran. "The non-Pareto principle; mea culpa." In: *Juran's Quality* (1975).

[74] Baris Kasikci, Thomas Ball, George Candea, John Erickson, and Madanlal Musuvathi. "Efficient Tracing of Cold Code Via Bias-Free Sampling." In: *USENIX Annual Technical Conf.* 2014.

[75] Baris Kasikci, Cristian Zamfir, and George Candea. "RaceMob: Crowdsourced Data Race Detection." In: *Symp. on Operating Systems Principles*. 2013.

[76] Samuel C Kendall. "BCC: Runtime Checking for C Programs." In: *USENIX Annual Technical Conf.* 1983.

[77] James C. King. "A new approach to program testing." In: *Intl. Conf. on Reliable Software*. 1975.

[78] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: Formal Verification of an OS Kernel." In: *Symp. on Operating Systems Principles*. 2009.

[79] Donald E. Knuth. "Structured programming with go to statements." In: *ACM Computing Surveys* (1974).

[80] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "HAFT: Hardware-assisted fault tolerance." In: *ACM EuroSys European Conf. on Computer Systems*. 2016.

[81] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. "Efficient state merging in symbolic execution." In: *Intl. Conf. on Programming Language Design and Implem.* 2012.

[82] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. "Code-Pointer Integrity." In: *Symp. on Operating Sys. Design and Implem.* 2014.

[83] Butler W. Lampson. "Hints for Computer Systems Design." In: *ACM Operating Systems Review* 15.5 (1983), pp. 33–48.

[84] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation." In: *Intl. Symp. on Code Generation and Optimization*. 2004.

[85] Xavier Leroy. "Formal Verification of a Realistic Compiler." In: *Communications of the ACM* (2009).

[86] John Levon and Philippe Elie. *Oprofile*. http://oprofile.sourceforge.net. 1998.

[87] Guodong Li, Indradeep Ghosh, and S. Rajan. "KLOVER: A symbolic execution and automatic test generation tool for C++ programs." In: *Intl. Conf. on Computer Aided Verification*. 2011.

[88] *LibFuzzer—A Library for Coverage-Guided Fuzz Testing*. http://llvm.org/docs/LibFuzzer.html.

[89] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. "Provably Correct Peephole Optimizations with Alive." In: *Intl. Conf. on Programming Language Design and Implem.* 2015.

[90] 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: *Intl. Conf. on Programming Language Design and Implem.* 2005.

[91] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. "LiteRace: Effective sampling for lightweight data-race detection." In: *Intl. Conf. on Programming Language Design and Implem.* 2009.

[92] Scott Matteson. *Millions of devices are still vulnerable, says researcher who discovered Stagefright*. http://www.techrepublic.com/article/millions-of-devices-are-still-vulnerable-says-researcher-who-discovered-stagefright/. 2016.

[93] Steve McConnell. *Code Complete*. Microsoft Press, 2004.

[94] Arnaldo Carvalho de Melo. *The New Linux Perf Tools*. http://vger.kernel.org/~acme/perf/lk2010-perf-paper.pdf.

[95] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert NM Watson, and Peter Sewell. "Into the depths of C: elaborating the de facto standards." In: *Intl. Conf. on Programming Language Design and Implem.* 2016.

[96] Florian Merz, Stephan Falke, and Carsten Sinz. “LLBMC: Bounded model checking of C and C++ programs using a compiler IR.” In: *Verified Software: Theories, Tools, Experiments* (2012).

[97] Bertrand Meyer. “Applying “design by contract”.” In: *IEEE Computer* (1992).

[98] Microsoft. *Project Springfield*. https://www.microsoft.com/en-us/springfield/.

[99] Barton P. Miller. *Foreword for Fuzz Testing Book*. http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html.

[100] B.P. Miller, L. Fredriksen, and B. So. “An Empirical Study of the Reliability of UNIX Utilities.” In: *Communications of the ACM* 33.12 (1990).

[101] Paul Mutton. *Half a million widely trusted websites vulnerable to Heartbleed bug*. https://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html. 2014.

[102] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. “Mining metrics to predict component failures.” In: *Intl. Conf. on Software Engineering*. 2006.

[103] Santosh Nagarakatte, Jianzhou Zhao, Milo M K Martin, and Steve Zdancewic. “CETS: Compiler Enforced Temporal Safety for C.” In: *Intl. Symp. on Memory Management*. 2010.

[104] Santosh Nagarakatte, Jianzhou Zhao, Milo M K Martin, and Steve Zdancewic. “SoftBound: Highly Compatible and Complete Spatial Memory Safety for C.” In: *Intl. Conf. on Programming Language Design and Implem.* 2009.

[105] National Vulnerability Database. https://web.nvd.nist.gov/view/vuln/statistics. 2014.

[106] George C. Necula, Scott McPeak, and Westley Weimer. “CCured: type-safe retrofitting of legacy code.” In: *Symp. on Principles of Programming Languages*. 2002.

[107] Nicholas Nethercote and Julian Seward. “How to Shadow Every Byte of Memory Used by a Program.” In: *Intl. Conf. on Virtual Execution Environments*. 2007.

[108] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. “Predicting vulnerable software components.” In: *Conf. on Computer and Communication Security*. 2007.

[109] Aleksandar Nikolich. *Static Binary Rewriting for American Fuzzy Lop*. https://github.com/talos-vulndev/afl-dyninst.

[110] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. *Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches*. https://intel-mpx.github.io/. 2017.

[111] *Openbenchmarking.org / Phoronix Test Suite*. http://openbenchmarking.org/.

[112] Tavis Ormandy. *Making Software Dumber*. http://taviso.decsystem.org/making_software_dumber.pdf. 2009.

[113] Karl Pettis and Robert C. Hansen. "Profile Guided Code Positioning." In: *Intl. Conf. on Programming Language Design and Implem.* 1990.

[114] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[115] *Pwn2Own contest*. https://en.wikipedia.org/wiki/Pwn2Own.

[116] John Regehr. *A Guide to Undefined Behavior in C and C++*. http://blog.regehr.org/archives/213. 2010.

[117] Steven Rostedt. *ftrace - Function Tracer*. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/trace/ftrace.txt.

[118] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. "AddressSanitizer: A Fast Address Sanity Checker." In: *USENIX Annual Technical Conf.* 2012.

[119] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer - Data race detection in practice." In: *Workshop on Binary Instrumentation and Applications*. 2009.

[120] Julian Seward and Nicholas Nethercote. "Using Valgrind to detect undefined value errors with bit-precision." In: *USENIX Annual Technical Conf.* 2005.

[121] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis." In: *IEEE Symp. on Security and Privacy*. 2016.

[122] Joseph L Steffen. "Adding Run-time Checking to the Portable C Compiler." In: *Software: Practice and Experience* (1992).

[123] *"Strong" stack protection for GCC*. http://lwn.net/Articles/584225/.

[124] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal war in memory." In: *IEEE Symp. on Security and Privacy*. 2013.

[125] *The Clang compiler*. http://clang.llvm.org/.

[126] The PaX team. *Address Space Layout Randomization*. http://pax.grsecurity.net/docs/aslr.txt.

[127] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM." In: *USENIX Security Symp.* 2014.

[128] Nikolai Tillmann and Jonathan De Halleux. "Pex – White Box Test Generation for .NET." In: *Tests and Proofs* (2008).

[129] *TIOBE Programming Community Index*. http://www.tiobe.com/tiobe_index/. Nov. 2004.

[130] Alan Turing. "Checking a large routine." In: *The early British computer conferences*. 1989.

[131] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. "-Overify: Optimizing Programs for Fast Verification." In: *Workshop on Hot Topics in Operating Systems*. 2013.

[132] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. "High System-Code Security with Low Overhead." In: *IEEE Symp. on Security and Privacy*. 2015.

[133] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. "Accurate Static Estimators for Program Optimization." In: *Intl. Conf. on Programming Language Design and Implem.* 1994.

[134] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient Software-Based Fault Isolation." In: *Symp. on Operating Systems Principles*. 1993.

[135] Minghua Wang, Heng Yin, Abhishek Vasisht Bhaskar, Purui Su, and Dengguo Feng. "Binary code continent: Finer-grained control flow integrity for stripped binaries." In: *Annual Computer Security Applications Conf.* 2015.

[136] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. "Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior." In: *Symp. on Operating Systems Principles*. 2013.

[137] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. "RIPE: Runtime Intrusion Prevention Evaluator." In: *Annual Computer Security Applications Conf.* ACM, 2011.

[138] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and understanding bugs in C compilers." In: *Intl. Conf. on Programming Language Design and Implem.* 2011.

[139] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code." In: *IEEE Symp. on Security and Privacy*. 2009.

[140] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems." In: *Symp. on Operating Sys. Design and Implem.* 2014.

[141] Michał Zalewski. *American Fuzzy Lop*. `http://lcamtuf.coredump.cx/afl/`.

[142] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. "Practical Control Flow Integrity and Randomization for Binary Executables." In: (2013).

[143] Mingwei Zhang and R Sekar. "Control Flow Integrity for COTS Binaries." In: (2013).

[144] Thomas Zimmermann, Nachiappan Nagappan, and Andreas Zeller. "Predicting Bugs from History." In: *Software Evolution*. 2008.

Jonas Wagner

Address
Jonas Wagner
EPFL IC IINFCOM DSLAB
INN 321, Station 14
1015 Lausanne
Switzerland

Links
people.epfl.ch/jonas.wagner
github.com/Sjlver
blog.purpureus.net

Email jonas.wagner@epfl.ch

Mobile +41 76 511 22 62

Research Interests

My mission is to create automated program analysis and transformation techniques that help developers construct better software with ease. I prototype these techniques into tools and evaluate them on real-world software systems.

My work lies at the intersection of verification, programming languages, and operating systems. It recognizes that program transformations are always trade-offs. They affect a program's speed, security, ease of verification, and reliability. Tools such as ASAP give developers an automated and precise way to obtain the most favorable trade-off for their particular use-case.

Education

PhD in Computer Science, EPFL: 2011 – 2017

dslab.epfl.ch

Dependable Systems Lab, EPFL, Lausanne
Under the direction of Prof. George Candea
Thesis title: Elastic Program Transformations: Automatically Optimizing the Reliability/Performance Trade-Off in Systems Software

Master in Communication Systems, EPFL: 2008 - 2011

ssc.epfl.ch/master

School of Computer and Communication Systems, EPFL, Lausanne
Specialization in Internet Computing. Master thesis in industry, on automatic detection of bad performance in VPN tunnels.

Bachelor in Communication Systems, EPFL: 2005 - 2008

ssc.epfl.ch/bachelor

Two years at EPFL, Lausanne, and one year at NTU, Singapore

Publications

ASAP: High System-Code Security with Low Overhead

dslab.epfl.ch/proj/asap

Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder
36th IEEE Symposium on Security and Privacy (S&P), 2015

ASAP is an automated approach and tool to instrument programs subject to performance constraints. It combines profiling and compiler techniques to generate programs that are as safe as possible, while satisfying the user's overhead budget.

-Overify: Optimizing Programs for Fast Verification

dslab.epfl.ch/pubs/overify.pdf

Jonas Wagner, Volodymyr Kuznetsov, and George Candea
14th Workshop on Hot Topics in Operating Systems (HotOS), 2013

-Overify is a compiler flag that speeds up software verification by up to 95×. It is based on the insight that compiling for verification requires different optimizations than compiling for fast execution, and introduces a new cost model to generate code that is adapted to the need of verification tools.

Work experience

Cyber Grand Challenge Finalist: February to August 2016 www.cybergrandchallenge.com

I joined team CodeJitsu at UC Berkeley, led by Prof. Dawn Song, to participate in DARPA's Cyber Grand Challenge. In this first ever all-machine hacking tournament, I worked on a system to automatically disassemble, analyze and instrument X86 binaries. Instrumentation protects the binaries against software vulnerabilities, while satisfying strict limits on memory consumption and execution time overhead. Team CodeJitsu ranked fifth out of seven finalists.

Internship at Google: May to August 2015 www.google.com

I built tools to systematically scan all Android apps for security vulnerabilities. The project used program analysis and abstract interpretation techniques in a distributed cloud setup. It is now running in production at Google and warns app developers whenever a new vulnerability is detected.

Master Thesis at Open Systems: Sept. 2010 to March 2011 www.open.ch

Performance measurement of VPN links and automatic detection of performance degradation. This project combined practical application of Perl, C and Unix with engineering and a solid mathematical foundation. The thesis was awarded the maximum grade and was important for upcoming network monitoring efforts at Open Systems.

Internship at MadeinLocal: Feb. to Aug. 2009 www.madeinlocal.com

Web development for MadeinLocal.com, the next generation local guide powered by social networking. In a dynamic start-up team, I assumed responsibility for developing business logic in Ruby on Rails and JavaScript, and connections to external sites such as Facebook.

Teaching and Professional Service

Teaching Assistant

Software Engineering: 2015, 2014, 2013, 2012
Calculus: 2014
Introduction to Programming: 2012, 2008, 2006
Information Theory and Coding: 2010
Stochastic Models: 2009

Project Supervisor

Summer@EPFL Internship by Azqa Nadeem, 2014
Moodle Accessibility Checker Plugin. 1st year master semester project by Quentin Cosendey, 2014
LibABC: A C Library for Software Analysis. 3rd year bachelor semester project by Florian Vessaz, 2013

Shadow Program Committee Member

EuroSys: European Conference on Computer Systems, 2016

External Reviewer

OSDI: USENIX Symposium on Operating Systems Design and Implementation, 2014
EuroSys: European Conference on Computer Systems, 2014 and 2012
CIDR: Conference on Innovative Data Systems Research, 2013
SOCC: ACM Symposium on Cloud Computing, 2012

Miscellaneous

Languages

German: native
English: Cambridge Certificate of Advanced English (level C1)
French: fluent (level B2)
Spanish: basic (level A2)

PolyProg: Organizing Programming Competitions polyprog.epfl.ch

I am a founding member of PolyProg, a student association at EPFL that promotes algorithmic and programming skills. I've contributed to the organization of numerous programming contests as well as related seminars and trainings.