Techniques for Identifying Elusive Corner-Case Bugs in Systems Software

THÈSE N^{O} 6735 (2015)

PRÉSENTÉE LE 14 SEPTEMBRE 2015

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE PROGRAMMATION DISTRIBUÉE
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Radu BANABIC

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2015

Abstract

Modern software is plagued by elusive corner-case bugs (e.g., security vulnerabilities). There are no scalable, automated ways of finding them, therefore such bugs can remain hidden until software is deployed in production. This thesis proposes approaches to solve this problem.

First, we present black-box and white-box fault injection mechanisms, which allow developers to test the behavior of their code in the face of failures in external components, e.g., in libraries, in the kernel, or in remote nodes of a distributed system. We describe a feedback-guided exploration algorithm that prioritizes black-box fault injection tests based on their estimated impact, thus discovering more bugs than random injection. For white-box testing, we proposed and implemented a technique to find Trojan messages in distributed systems, i.e., messages that are accepted as valid by receiver nodes, yet cannot be sent by any correct sender node. We show that Trojan messages can lead to subtle semantic bugs. Our fault injection techniques found new bugs in systems such as the MySQL database, the Apache HTTP server, the FSP file service protocol suite, and the PBFT Byzantine-fault-tolerant replication library.

Testing can find bugs and build confidence in the correctness of a system. However, exhaustive testing is often infeasible, and therefore testing may not discover all bugs before a system is deployed. In the second part of this thesis, we describe how to automatically harden production systems, reducing the impact of any corner-case bugs missed by testing. We present a framework that reduces the overhead imposed by instrumentation tools such as memory error detectors. Lowering the overhead enables system developers to use such tools in production to harden their systems, reducing the impact of any remaining corner-case bugs. We used our framework to generate a version of the Linux kernel hardened with Address Sanitizer. Our hardened kernel has most of the benefit of full instrumentation: it detects the same vulnerabilities as full instrumentation (7 out of 11 privilege escalation exploits from 2013-2014 can be detected using instrumentation tools). Yet, it obtains these benefits at only a quarter of the overhead.

Key words: Automated testing, fault injection, symbolic execution, instrumentation, corner-case bugs

Résumé

Les logiciels modernes sont remplis de bugs difficiles à trouver (des bugs de sécurité par exemple). Ces bugs peuvent rester cachés puis déployés en production, car il n'existe pas de moyens efficaces et automatiques pour trouver ces bugs. Cette thèse propose des approches afin de résoudre ce problème.

Nous présentons tout d'abord des mécanismes d'injection de fautes qui permettent aux développeurs de tester le comportement de leur code en présence de défaillances de composants externes (par exemple dans des librairies, dans le noyau, ou dans les noeuds distants d'un système distribué). Nous explorons l'injection de fautes “boîte noire” et l'injection de fautes “boîte blanche”. Nous décrivons comment injecter des fautes boîte noire plus efficacement, en privilégiant des tests ayant un impact estimé plus élevé. Pour les tests boîte blanche, nous proposons et avons implémenté une technique permettant de trouver des messages de Troie dans les systèmes distribués, c'est-à-dire des messages qui sont considérés comme valides par les noeuds récepteurs, mais qui en réalité ne peuvent pas provenir d'un noeud émetteur correct. Nous montrons que les messages de Troie peuvent mener à des bugs sémantiques subtils. Nous avons utilisé des techniques d'injection de fautes pour révéler des nouveaux bugs dans des systèmes comme la base de données MySQL, le serveur HTTP Apache, la suite de protocoles de service de fichiers FSP, et la libraire de réplication PBFT.

Les tests peuvent permettre de découvrir des bugs et renforcer la confiance dans la justesse d'un système. Il est cependant souvent impossible d'effecuer des tests exhaustifs, et les bugs ne sont ainsi souvent découverts qu'après le déploiement du système. Dans la deuxième partie de cette thèse, nous décrivons comment renforcer automatiquement les systèmes en production, en réduisant l'impact des bugs restants qui auraient été manqués par les tests. Nous présentons un framework qui réduit les coûts de performance engendrés par des outils d'instrumentation tels que les détecteurs d'erreurs de mémoire. La réduction de ces coûts de performance permet aux développeurs de systèmes d'utiliser ces outils d'instrumentation sur des systèmes en production. Nous avons utilisé notre framework pour générer une version du noyau de Linux renforcée avec l'instrument de sécurité de la mémoire. Ce noyau renforcé a la plupart des avantages d'une instrumentation complète : il détecte les mêmes vulnérabilités qu'une instrumentation complète, mais n'engendre qu'un quart des coûts de performance.

Mots clefs : Tests automatiques, injection de faute, exécution symbolique, outils d'instrumentation

Acknowledgements

I am thankful for all the support I have received over the last years. This thesis would not have been possible without it.

I had the opportunity to learn from two great advisors during my PhD. I will be forever grateful to Professor George Candea and Professor Rachid Guerraoui for their guidance, both in terms of research and personal development. I was extremely lucky to have such mentors to model my career.

I am grateful to Professor Edouard Bugnion, Doctor Christian Cachin, and Doctor Marcos Aguilera for agreeing to be members of my PhD defense committee, and to Professor Arjen Lenstra, who served as the jury president. Their questions and feedback were insightful and helped me look at my work from new perspectives.

I would like to thank Paul Marinescu and Jonas Wagner for our work together. Paul helped me get started on my first project when I first joined EPFL as an intern and was influential in my work on fault injection. Jonas and I worked together during the final part of my PhD and our discussions shaped the way I think about software dependability.

During my PhD, I visited IBM Research Zurich as an intern. I was lucky to work with a very talented group of people, Doctor Christian Cachin, Doctor Marko Vukolic and Doctor Alessandro Sorniotti. I thank them for accepting me in their group and challenging me to approach systems from a different perspective.

I wish to thank all members of DSLab and LPD. They offered a lot of feedback and support for my work, but also helped make Lausanne feel like a home. The talent and hard work of each of them motivated me to push my boundaries in order to deserve to be part of such accomplished groups.

I had a lot of support from friends, both here and back home in Romania. I have known Flaviu and Tudor for a long time and I was lucky to have them join me here in Lausanne. I also thank all my friends back home, Alex, Ghita, Mircea, Paul, Sebi, Vlad, and many others who did not let the distance tear us apart.

I am especially grateful to Raluca for all the encouragement and support of my work, but also for helping me relax and enjoy my time here.

I thank my parents, Silvia and Dorel, for the love, guidance and encouragement they provided throughout my upbringing. My accomplishments are their merits as much as mine.

Finally, I would like to thank EPFL, IBM, the Swiss National Science Foundation, and the European Research Council for funding my research.

Contents

List of Figures

List of Tables

1 Introduction

Computer systems are becoming more and more integrated in our life. This is enabled by the rapid evolution of both hardware and software. Today's software systems are the most intellectually complex artifacts ever built by humankind [1]. A modern car can contain hundreds of millions of lines of code. The avionics and support systems of a Boeing 787 plane—critical systems—have 6.5 million lines of code [30]. Even very simple tasks can be unexpectedly complex: displaying the current date and time on a Debian Linux system requires executing $636,562$ CPU instructions[2]. It is difficult for developers to understand, build and test such large systems.

In order to be more understandable, most systems use modular designs. Modularity allows developers to abstract away irrelevant details. In the previous date and time example, more than two thirds of the instructions ($429,426$) are executed in kernel space. The developers of the `date` application do not need to worry about the details of what those instructions do, only about the interaction between their own application and the kernel via system calls. Of the remaining instructions, another large chunk represents instructions in various libraries, which can also be abstracted away by the developers. Due to modular design, the developers of `date` can focus their attention on the code they wrote themselves—a few hundred lines of code—and on the API calls they use to interact with other components.

However, modularity is not a panacea against complexity. Interfaces and communication protocols can still have intricacies that lead to misunderstanding and bugs. Moreover, modern systems often have external components are not completely under the developers' control. By using third party components, developers give up some control over what behaviors they can exercise in their own program; this lack of control can lead to poor testing and thus lack of reliability.

For instance, consider the *write* function[3] in the LibC standard library, which writes data to a file. The function can fail when a user's quota of disk blocks is exhausted (indicated by the `EDQUOT`

[1]Grady Booch, *The Promise, The Limits, The Beauty of Software*, http://csta.acm.org/Resources/sub/ResourceFiles/Booch_Lecture.ppt

[2]obtained using the command `perf stat -e instructions /bin/date`

[3]http://linux.die.net/man/2/write

error). An application that calls *write* should detect this failure and handle it gracefully, e.g., by informing the user of the error. Otherwise, the data may be lost, as it is not written to disk. It is important for the failure recovery code to be correct—the developers need to test the behavior of this code. The problem with this is that the developers of the application calling *write* do not have direct control over when the `EDQUOT` error is triggered—the failure depends on the state of the entire system, not just the calling application. This makes it difficult for developers to exercise failure recovery code in end-to-end testing. Due to this lack of testing, recovery code may hide elusive corner-case bugs that do not appear during the testing phase, but may be triggered in production.

Corner-case bugs in the recovery code of a system can have dire consequences. For example, Ma.gnolia[4] was a social bookmarking service whose main database was corrupted after a failure. The service had a backup and recovery scheme in place but, unfortunately, it was incorrectly set up[5] and therefore the entire database was lost. Ma.gnolia lost a large portion of its users after this failure and was soon shut down.

In this thesis, we describe how we hardened systems against corner-case bugs in the interaction between applications and libraries (chapter 3), between nodes in distributed systems (chapter 4), and between the kernel and user-level applications (chapter 5).

1.1 Problem Definition

This thesis addresses the problem of corner-case bugs. By "corner-case" we mean execution paths that are not part of the core functionality of a system. Users could benefit from all features of a piece of software without ever encountering a corner-case execution. Corner-case execution paths are triggered rarely, e.g., only in case of failures, and are difficult to systematically trigger during end-to-end testing. However, they may execute in production, especially for systems deployed at large scale. We aim to avoid catastrophic behaviors due to bugs in corner-case executions.

Existing solutions are either ineffective, or impractical for wide-spread use. We target our techniques to general-purpose software. Thus, any solution needs to be practical in terms of time, resources, and human effort required. We target techniques that:

- are practical to set up
- are practical to run

1.2 Proposed Solution

We explore a combination of techniques that complement each other in order to minimize the impact of corner-case bugs. Our proposed approaches cover both the development phase and

[4]https://en.wikipedia.org/wiki/Gnolia
[5]http://www.datacenterknowledge.com/archives/2009/02/19/magnolia-data-is-gone-for-good/

the production phase of a system's life cycle. For the development phase, we propose testing techniques that can discover bugs before software is released in production. We present both black-box and white-box testing techniques that aim to discover bugs in corner-case executions. For the production phase of software, we propose an approach to use instrumentation at low cost in order to harden corner-case execution paths.

Black-box fault injection testing refers to techniques that simulate failures in the environment of a system under test in order to search for bugs in failure recovery code. The term "black-box" stems from the fact that the approaches do not require knowledge about the internals of the system under test. Because of this, black-box fault injection tools are easy to set up, with little developer effort. The downside, however, is that due to the lack of knowledge black-box injection tools often resort to random testing, selecting failure scenarios arbitrarily. For large, complex systems this can be ineffective, as there are many possible failure scenarios and the probability of finding bugs is low. In order to make the exploration of failure scenarios more effective than random, we developed a scalable, feedback-guided algorithm for scheduling fault injection scenarios. Despite the lack of prior information about the internals of the system under test, our algorithm learns from the sequence of tests it performs. Our approach can navigate through a large number of possible fault injection scenarios, adapting to the particularities of the system under test and prioritizing fault scenarios that are most likely to expose high-impact bugs.

White-box testing refers to techniques that exploit knowledge of the internals of the system under test. This knowledge can come either from the developers, in the form of models of the system under test, grammars that describe input structure or program annotations, or it can be obtained without human involvement, by a tool that analyzes the code or binary of the system under test. White-box testing requires more effort to set-up, but in return can be more effective at finding bugs. There is significant related work on how to use this extra knowledge to guide testing so that it finds bugs faster than black-box approaches. However, in this thesis, we take a different approach, using white-box information in order to discover a new class of bugs in distributed systems, which would otherwise be difficult to identify even if they were triggered during black-box testing. We define Trojan messages as those that are accepted as valid by a receiver but cannot be generated by any correct sender in a distributed system. Trojan messages enable failures to propagate among nodes of a distributed system, potentially disrupting entire deployments. We show how such messages lead to subtle errors in some real distributed system implementations. In this thesis, we describe a technique to systematically discover Trojan messages in the implementation of a distributed system.

Finally, instrumentation refers to additional instructions that are inserted in a target application in order to improve non-functional properties, such as reliability. Instrumentation comes in different forms: it can be manually added by developers, e.g., logging statements or asserts, or it can be added automatically by third-party tools, e.g., stack canaries. Instrumentation can reduce the impact of a bug in production. For instance, it reduces the time to fix a bug, by providing developers with more information about its behavior, or can even reduce the severity of a bug automatically (e.g., stack canaries can avoid certain types of exploits). However, this benefit comes at a cost, as instrumentation requires extra resources at runtime (CPU cycles, memory, disk I/O). The overhead is often so high that developers opt to avoid instrumentation altogether in production. However, this need not be the case for corner-case execution paths. Such paths are triggered rarely, by definition, and, therefore, any overhead encountered during those executions has little effect on the overall performance of the system. Based on this observation, we propose enabling instrumentation selectively, only on some execution paths in the system under test. We call this concept “Elastic Instrumentation”. The difficulty with Elastic Instrumentation is that there are many execution paths, so developers cannot make the decision of which paths to instrument manually. We developed a framework that helps automate this process. Our framework automates the many small choices between cost and benefit and allows developers to easily apply the Elastic Instrumentation principle to their systems.

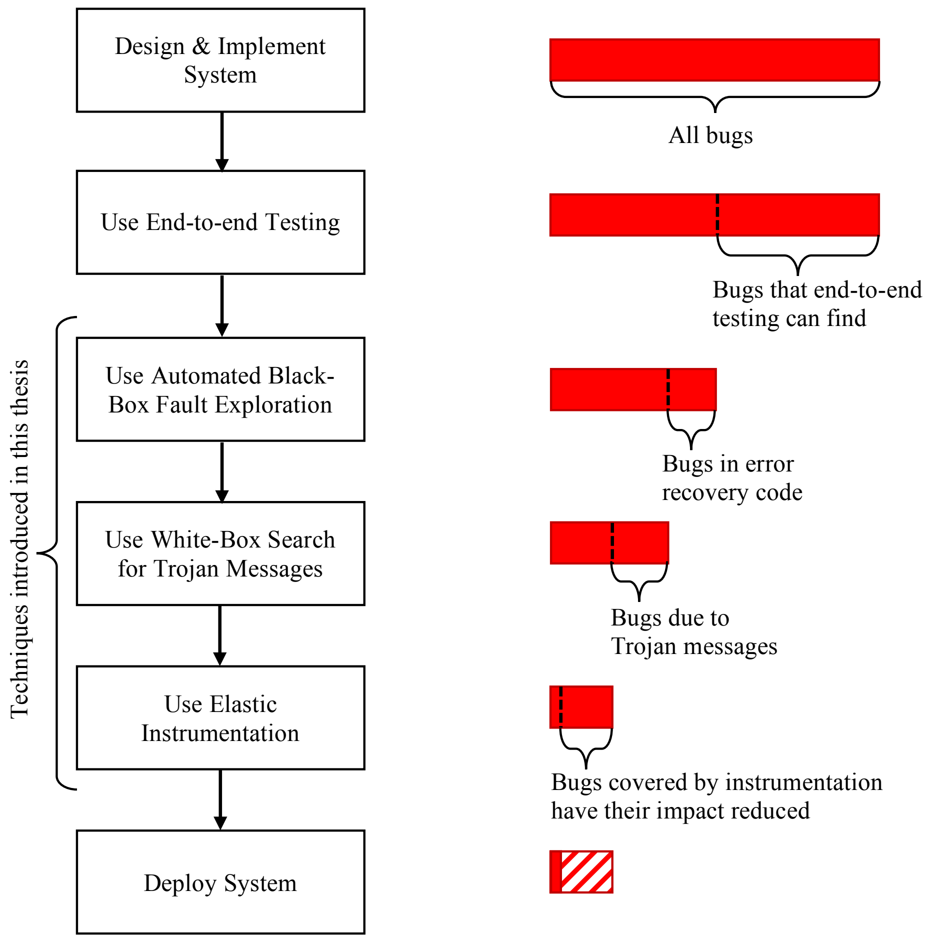
Design & Implement System
Use End-to-end Testing
Use Automated Black-Box Fault Exploration
Use White-Box Search for Trojan Messages
Use Elastic Instrumentation
Deploy System
Techniques introduced in this thesis
All bugs
Bugs that end-to-end testing can find
Bugs in error recovery code
Bugs due to Trojan messages
Bugs covered by instrumentation have their impact reduced

Figure 1.1 – Workflow for applying the techniques described in this thesis, and the effect on each technique on the number of bugs in a system.

Figure 1.1 presents a workflow for using the approaches described in this thesis. The techniques we propose help developers discover and fix bugs that they could not find with regular testing techniques. The first step is to use black-box fault injection testing. Our approach is designed to be easy to use, requiring little involvement from the human users, and thus has a low barrier to entry. Black-box testing can find buggy error recovery code that causes the program to crash, hang, or break assertions. However, there are other, subtler behaviors of bugs that are not as easily detectable, such as Trojan messages. To discover these, the second step in the workflow is to use our white-box fault injection. Testing can discover many bugs, but usually it is unfeasible to explore all possible testing scenarios before deploying a system in production. Therefore, some bugs still lurk in production-ready code. The third step in our workflow is to use instrumentation to reduce the impact of these remaining bugs. Our Elastic Instrumentation framework lowers the cost of instrumentation and allows previously inaccessible tools to fit in the overhead budget that developers can afford.

1.2.1 Automated Black-box Fault Exploration

Fault injection is a form of testing that consists of introducing faults in a system under test, with the goal of exercising the system's error-handling code paths. Fault injection is crucial to system testing, especially as increasingly more general-purpose systems (e.g., databases, backup software, Web servers) are used in business-critical settings. Recent tools, like LFI [74], offer the ability to simulate a wide variety of fine-grained faults that occur in a program's environment and become visible to applications through the application–library interface.

However, with fine-grain control comes the necessity to make hard choices, because these tools offer developers a vast universe of possible fault scenarios. There exist three degrees of freedom: *what* fault to inject (e.g., `read()` call fails with `EINTR`), *where* to inject it (e.g., in the logging subsystem), and *when* to do so (e.g., while the DBMS is flushing the log to disk) [75]. Poor choices can cause test suites to miss important aspects of the tested system's behavior.

One way to avoid making hard choices is to perform brute-force exhaustive testing, such as using large clusters to try out all combinations of faults that the available injectors can simulate—this approach will certainly find the faults that cause most damage. Alas, the universe of possible faults is typically overwhelming: even in our small scale evaluation on MySQL, the fault space consisted of more than 2 million possibilities for injecting a single fault. Exploring such fault spaces exhaustively requires many CPU-years, followed by substantial amounts of human labor to sift through the results. A commonly employed alternative is to not inject all faults, but only a randomly selected subset: the fault space can be uniformly sampled, and tests can be stopped whenever time runs out. However, random injection achieves poor coverage, and the chances of finding the high impact faults is low.

In the first part of the thesis (chapter 3), we propose an approach in which still a subset of the fault space is sampled, but this sampling is guided. We observe that the universe of faults often has an inherent *structure*, which both random and exhaustive testing are oblivious to, and that exploiting this structure can improve the efficiency of finding high-impact faults. Since this structure is hard to identify and specify a priori, it needs to be *inferred*.

We propose a technique that uses a fitness-guided feedback-based algorithm to search for high-impact faults in spaces with unknown structure. It uses the effect of previously injected faults to dynamically learn the space's structure and choose new faults for subsequent tests. The search process continues until a specific target is reached, such as a given level of code coverage, a threshold on the faults' impact level (e.g., "find 3 disk faults that hang the DBMS"), or a time limit. We implemented a prototype of our technique in AFEX, a cluster-based parallel fault injection testing system. Besides prioritizing fault injection scenarios, AFEX also analyzes the injected faults for redundancy, clusters and categorizes them, and then ranks them by severity to make it easier for developers to analyze. Despite being developed primarily as a low-set-up-cost black-box tool, AFEX is also capable of leveraging domain-specific knowledge to improve its search efficiency, whenever humans have such knowledge and can suitably encode it.

Our approach advances the state of the art by introducing an adaptive algorithm for *finding high value faults*. We implemented our algorithm in an extensible system that can automatically test real-world software with fault injection scenarios of arbitrary complexity.

1.2.2 White-box Search for Trojan Messages

In the second part of this thesis (chapter 4), we study what we call *Trojan messages*. These are messages that are intelligible to correct receiver nodes of a distributed system but cannot be generated by any correct sender nodes in that system. Trojan messages result from absence of defensive programming and constitute a source of vulnerabilities, similar to divisions by zero or buffer overflows. We know of no prior work that automatically discovers this type of messages, yet discovering them is crucial.

Trojan messages are particularly dangerous because, by definition, they are a means of failure propagation across nodes in a distributed system. This makes Trojan messages the Achilles' heel of distributed systems. Most reliability techniques in distributed systems are based on the assumption that failures are independent: if a node fails in some way, the impact on other nodes should be minimal, i.e., the failure should not propagate. But even if nodes are programmed independently and placed on geographically remote sites, nodes need to communicate, typically by exchanging messages. If a Trojan message exists in a system, it means that a correct node accepts as valid a message from a failed node, and can potentially become corrupted itself.

Trojan messages can have a major impact on the behavior of distributed systems. For example, the Amazon S3 storage system suffered several hours of downtime in 2008 [1]. The problem was caused by "a handful of messages [...] that had a single bit corrupted". Unfortunately for the system, "the message was still intelligible, but the system state information was incorrect". This allowed the initial corruption to propagate to other, correct nodes of the system, until most of the collective information in the system was corrupt. Such corner cases are difficult to identify. In order to prevent such downtime in the future, Amazon engineers added checks to “log any such messages and then reject them”.

The core problem that caused the Amazon S3 downtime was the fact that nodes accepted the corrupt messages despite the fact that no correct node could have generated such messages in the respective context. We say that S3 nodes accepted Trojan messages. We argue that, if such messages exist in a system implementation, they unnecessarily increase its attack surface—servers should do what correct clients require them to do and nothing more. Since, by definition, correct nodes cannot generate Trojan messages, the likelihood that such messages will be encountered during regular testing is low. Trojan messages are likely to exercise untested code paths and surface hidden, potentially undesired, behavior. These paths can be triggered voluntarily by malicious users of the system, or involuntarily by failed nodes.

We propose Achilles, a system whose purpose is to identify Trojan messages automatically. Achilles uses symbolic execution in order to discover Trojan messages in the implementation of a distributed system. Although this is a hard task, we argue for analysis at the implementation level, rather than specification or model level, as bugs often creep in the implementation. Recognizing the important distinction between specification and implementation, the most stringent operators of distributed systems have started testing their live production systems with fault injection (e.g., Google organizes regular “fire drills” in which engineers intentionally cause failures in critical live systems [63]). Trojan messages are good candidates for such fault injection, if only developers had a way to find and inject them.

Our work advances the state of the art by identifying Trojan messages as a source of weakness in distributed systems and by proposing a technique to systematically discover such messages in the implementation of distributed systems. We introduce a new use case for symbolic execution, aimed at discovering execution paths that accept Trojan messages, and describe a set of optimizations that make the search for Trojan messages efficient.

1.2.3 Elastic Instrumentation

In the third part of the thesis (chapter 5), we focus on techniques for handling corner-case bugs that still lurk in a system after it has been deployed in production. We show how instrumentation can be selectively added only to certain execution paths of the system, keeping most of the benefits of instrumentation, but at only a fraction of its cost.

Developers can choose from several powerful instrumentation methods to systematically enrich their software. Logging and tracing can ease debugging by providing the developer with more information about the behavior of the program. Assertions, code contracts, and memory safety instrumentation can catch buggy behaviors early after a failure has occurred, preventing subtle bug manifestations that are difficult to detect and debug.

However, performance concerns often limit the use of such methods to the development phase and force developers to abandon them for production software. Many instrumentation tools incur high costs, e.g., >100% overhead for strong memory safety enforcement in C programs[6]. While these costs may be acceptable during development, they could affect user experience or violate service level agreements in production.

We argue that it is unnecessary to remove all instrumentation just because of the performance requirements of a few execution paths. In fact, removing all instrumentation can be thought of as overzealous optimization, which is discouraged in the systems building community. Butler Lampson [69] advises to "make the common case fast". Donald Knuth [62] argues for optimizing only the 3% of a system where speed really matters.

We challenge the conventional wisdom that instrumentation is too expensive for production use and propose a novel way to apply instrumentation selectively to a system. Our approach is based on two main insights:

- The cost of instrumentation is often a continuum, rather than binary, because instrumentation is divisible into small, independent *atoms*.
- Instrumentation is cheap on corner-case executions paths, as these are by definition executed rarely

We validate our insights with the help of a study of vulnerabilities in the Linux kernel (subsection 5.1.1). In this study, we found that approximately a third of Linux kernel vulnerabilities fall within the scope of instrumentation tools. More surprisingly, the majority of these vulnerabilities appear on rare execution paths in the kernel—they are triggered by system calls that few user-level applications use. This means that the functionality that benefits from instrumentation the most tends to be the cheapest to protect.

We advance the state of the art by recognizing and distilling the Elastic Instrumentation principle and by introducing a framework to automate the trade-off between cost and benefit of instrumentation.

[6]https://code.google.com/p/address-sanitizer/wiki/PerformanceNumbers

2 Background and Related Work

We are not the first to address the problem of corner-case bugs. Researchers have tried to solve the problem in various ways, which can be grouped broadly in testing, which tries to trigger bugs and allow developers to fix them, and instrumentation, which tries to reduce the impact of bugs. In this chapter we give an overview of some of the related work that addresses corner-case bugs.

We start with a short survey of fault injection techniques (section 2.1). Fault injection is a mechanism through which failures can be generated or simulated in a component of a system. Thus, fault injection tools allow developers to test failure scenarios and search for corner-case bugs in their system's recovery code.

However, writing failure scenarios by hand is an arduous task for developers; since large systems can fail in a variety of ways, it is impractical to rely solely on manually written tests. To solve this problem, researchers have proposed various testing frameworks that can automate the generation of tests. In section 2.2 we present some of these approaches.

Testing frameworks can be either black-box or white-box. Black-box frameworks do not require any information about the internals of the target system, while white-box frameworks do. Black-box testing's main appeal is the low barrier to entry. Many developers can run black-box testing, including fault injection, without needing to invest significant resources. The low barrier to entry also enables third-party testing services to assess the fault-tolerance of systems.

The effectiveness of black-box testing can suffer due to the lack of information about the system under test. In section 2.3 we describe white-box approaches that do use internal system information in order to guide testing. This information can come in many forms. Developers can write machine-readable models or specifications that are understandable by testing frameworks (subsection 2.3.1). These models can be used in order to guide testing towards the most relevant execution paths (e.g., by avoiding redundant tests).

To counteract the need for explicit models, symbolic execution has emerged as a technique that uses the internal structure of a system under test in order to guide testing. Symbolic execution does not need separate models written by developers—it analyzes the code (or binary) of a target system and generates tests that exercise different execution paths. We briefly explain symbolic execution, its benefits and major drawback, as well as a few recent approaches that attempt to alleviate the drawback in subsection 2.3.2. Then, we highlight a few different uses of symbolic execution, from generic bug-finding to protocol interoperability testing in subsection 2.3.3.

Testing is useful for finding bugs during development. However, given limited resources, testing is often incomplete and therefore some corner-case bugs can still lurk in production code. The idea of instrumenting programs with additional code has long been used to improve program reliability. The additional code can consist of asserts, which developers write to check the internal state of their program and catch failures early, logging statements, which help developers debug problems by providing them with more information about the behavior of a bug, memory safety checks, which verify memory accesses to avoid bugs like buffer overflows, etc. In subsection 2.4.1 we enumerate a few instrumentation techniques that can harden programs against bugs. We then describe some approaches that attempt to reduce the overhead of these techniques in order to make them suitable for use in production (subsection 2.4.2).

2.1 Fault Injection Tools

Fault injection is an approach to test a system's response to failures. The tests are produced by fault injection tools, which either trigger or simulate failures in system components, in order to assess the recovery of the rest of the system. Fault injection can be applied to all layers in a system.

At the hardware design level, MEFISTO [54] allows fault injection to be performed on VHDL models, to evaluate their fault tolerance. Even after hardware is built, failures can be introduced precisely in a circuit using lasers [90]. Such techniques can be used to check if a piece of hardware, such as memory using error-correcting codes (ECC), can properly tolerate faults without propagating them to the software level.

At the interface between hardware and software, fault injection tools introduce failures in hardware components, in order to test the resilience of software. FERRARI [58] uses a software-based approach to inject faults in hardware (e.g., flipping bits in CPU registers or in memory). FTAPE [99] uses similar techniques for fault injection, but also runs a workload that stresses the system under test, in order to increase failure propagation and find more bugs.

At the software level, fault injection has been applied to the communication between different software components. TestApi [98] allows injecting faults in the communication between .NET components. JBoss Byteman[1] can inject code in arbitrary parts of a Java program, and is often used for fault injection. At a higher level, Ju et al. describe techniques to inject faults in between services in OpenStack [57].

[1]http://www.jboss.org/byteman

Our black-box exploration algorithm is oblivious to the fault injection tools; it navigates through an abstract fault space, where the specifics of how a fault is injected are not important. In our evaluation we used LFI [75, 73], a tool that can inject faults in the communication between an application and shared libraries.

LFI analyzes shared libraries in order to discover the failures they can expose and then generates a shim library that can simulate those failures on demand. It provides a mechanism for testers to precisely control what faults are simulated, and where and when they are injected in the system under test. We chose LFI because of its ability to inject faults accurately, and because it can inject in standard C libraries, which are frequently used by systems. Moreover, LFI's library analysis tool fits our framework well, as we do not need to analyze the library interface manually.

2.2 Black-box Testing Frameworks

Fault injection tools give powerful capabilities to developers, who can use such tools to augment their test suites with failure scenarios. Fault injection tools have low set-up costs; there are many mature tools that can abstract away the details of fault injection, allowing developers to specify the failure scenarios they want to inject in a high-level language (e.g., SETSUDŌ provides a high-level language for describing perturbation sequences, while LFI allows developers to define injection triggers in an XML format). However, many of the injection tools we described in the previous section require human testers to design each injection scenario individually. This is an arduous task, as there can be millions of different failure scenarios for a system. In this section, we describe testing frameworks that automate the task of developing and running failure scenarios.

The purpose of testing is to cover as many behaviors of the system under test as possible, in order to find and eliminate bugs. Testing frameworks systematically explore points from a fault space that encompasses all possible failure scenarios for a given system. Frameworks such as FATE [50] and EDFI [44] take this approach. As black-box tools, neither of these frameworks requires a separate definition of the space of failure scenarios that can be injected, but rather they compute the space themselves. FATE runs the target system under a given workload and intercepts I/O calls, computing an identifier for each I/O point. A separate failure service coordinates fault injection, deciding which I/O calls to fail and when. EDFI inserts potential failure points into a target program via static instrumentation, and uses a controller to activate the failures at runtime. This strategy allows users of EDFI to precisely target parts of their program with fault injection.

However, exhaustive exploration of all possible failure scenarios is often infeasible, given the limited resources allocated. The straightforward way to achieve coverage while only exploring a subset of the possible failure scenarios is to randomly choose tests to execute. Random fault injection is used in practice by several systems. Netflix assess the reliability of their systems using a tool called Chaos Monkey [2], which randomly terminates virtual machines in Netflix services.

[2]http://techblog.netflix.com/2012/07/

Random injection is also the approach chosen by Hadoop's Fault Injection framework [51] and the Linux kernel [11], but also by research projects such as the SETSUDŌ fault injection framework [56] (when in black-box mode).

We propose a search algorithm that prioritizes fault injection scenarios. Our black-box fault injection framework uses a metaheuristic algorithm to learn from already executed test scenarios and improve the efficiency of the search for bugs. The idea of using metaheuristic search techniques in software testing is not new; McMinn compiled a survey of such techniques [79]. This survey covers various heuristics, such as Genetic Algorithms or Simulated Annealing, applied to testing techniques that range from white-box to black-box. The survey relates to exploring the control flow graph of a program, rather than fault injection in particular.

2.3 White-box Testing Frameworks

White-box testing techniques use information about the system under test in order to make testing more efficient. This information can be models written by the developers, or it can be extracted by the testing tool itself, by analyzing the system under test. In this section we describe both approaches. We put particular emphasis on the latter category, focusing on symbolic execution, a technique originally developed in the 1970s [61] that has recently received a lot of attention for its capability of detecting corner-case bugs.

2.3.1 Model-based Testing

Researchers have attempted to automatically derive models from prior observations of system behavior, and then develop heuristic methods for fault or test selection. As an interesting example, Tahat et al. [97] present a novel approach to requirement-based test generation. Based on a set of individual requirements expressed in plain text or Specification and Description Language (SDL), a system model is automatically created with requirement information mapped to the model. The system model is used to automatically generate test cases related to individual requirements.

For fault injection, SAMC [70] and Relyzer [52] use semantic information about the system under test in order to perform state-space reduction policies and avoid redundant tests.

In order to reduce the developer effort required to write code and models separately, some tools generate the code themselves based on models written by developers. For example, Mace [60] provides a language in which developers can write distributed system specifications. Mace can use the specifications for efficient model checking, discovering bugs early during development. After the specification is validated, Mace can generate a C++ code skeleton, which developers use to build their system.

2.3.2 Symbolic Execution and Path Explosion

In our work, we use symbolic execution [61] for white-box testing. Symbolic execution is a technique that can systematically enumerate execution paths in a given system, checking if any of the execution paths lead to a bug. It does not require developers to write a model or specification of the system under test; instead, the symbolic execution engine analyzes the code (or binary) of the system in order to guide testing towards new execution paths. At a high level, symbolic execution works by analyzing how inputs affect the conditional branches in a system under test. The symbolic execution engine uses an SMT solver, such as STP [43] or Z3 [37] to find which combinations of inputs are feasible, and furthermore which trigger different execution paths through the program.

```
void foo(unsigned int x) {
    if (x == 3) {
        if (x == 2) {
            printf("a");
        } else {
            printf("b");
        }
    } else {
        printf("c");
    }
}
```

Figure 2.1 – A simple function to be analyzed by symbolic execution

Consider the example function in Figure 2.1. There are two feasible execution paths through this function: one execution path prints ’b’, while the other prints ’c’. There is no execution path that can print ’a’, as this would require both `x == 3` and `x == 2` to be true at the same time, which is impossible (we do not consider multithreading in this example). In symbolic execution, an engine interprets the program code and, for each conditional branch, evaluates the feasibility of both possible outcomes using an SMT solver. For the first `if` statement, both outcomes are possible: `x` can take value 3, or can take a value different from 3. Therefore, the engine will continue interpreting the program, and keep track of the two possible execution paths, and the conditions on `x` that make each path possible. For the next conditional statement, `x == 2` cannot be true, because this execution path already has the precondition that `x == 3`. Therefore, the symbolic execution engine discards the branch `x == 2`. Eventually, symbolic execution finds both feasible paths in the function in Figure 2.1. The power of symbolic execution is in the fact that it systematically discovers all possible paths in the system, including elusive corner cases; in the example in Figure 2.1, symbolic execution only needs to generate two tests to explore the two feasible execution paths, while naive black-box testing would have to iterate through all possible values of `x` to be sure that it discovered all feasible paths.

The big problem of symbolic execution is the so-called path explosion problem. Systems have huge numbers of feasible execution paths—on the order of 2^n different paths, where n is the number of conditional branches in the system. While not directly related to corner-case bugs, path explosion is a problem that needs to be addressed in order to make symbolic execution practical.

There have been several attempts to solve the path explosion problem. Patrice Godefroid proposed Compositional Dynamic Test Generation [46]. The idea behind this work is that software systems are made of different re-usable components, and the analysis of these components can be reused. Thus, instead of exploring each component again and again for each piece of software, the analysis can be built bottom-up, analyzing components in isolation and then combining the summaries of each component analysis into the analysis of an entire software system.

Godefroid et al. also proposed SAGE [47], a system that combines symbolic execution and fuzz testing (random input generation). The combination of these two techniques draws advantages from both, allowing SAGE to be successfully used to find bugs in large software systems. There are reports that SAGE found one-third of all the bugs discovered by file fuzzing during the development of Microsoft's Windows 7 [48].

Cloud9 [25] addresses the path explosion problem in two ways. On the one hand, it allows system developers to "throw hardware at the problem": Cloud9 is a parallel symbolic execution framework that can scale to thousands of nodes. On the other hand, Cloud9 comes with a complete set of POSIX models, which are specifically written with symbolic execution in mind. This allows the symbolic execution engine to avoid dealing with the complex semantics of different POSIX-compliant implementations and focus the analysis on the software under test itself.

Like Cloud9, S2E [32] also addresses the interaction between a program and its environment, giving developers control over what to test with symbolic execution. S2E is a symbolic execution platform that runs an entire virtual machine. In order to focus the analysis on specific components of the system under test, S2E provides different consistency models and a wide selection of plugins that control the exploration of execution paths.

Kuznetsov et al. [66] propose a technique for merging symbolic execution states, reducing the number of paths that need to be analyzed individually (but increasing the analysis time for each path). The authors developed a way to decide when merging is beneficial to the overall analysis time.

Guerraoui and Yabandeh [49] address the problem of path explosion in the model checking of distributed systems. In their work, the authors proposed model checking individual nodes in isolation, separated from the complexity of network message interleaving. The entire system is only model checked for any bugs discovered by the individual analyses, in order to validate if there is any global execution that can exercise the bug.

2.3.3 Uses of Symbolic Execution

In our work we do not attempt to address the path explosion problem, but describe a technique to use symbolic execution for a precise purpose (finding Trojan messages in distributed systems). We do not modify the symbolic execution mechanism, therefore any solution that addresses path explosion for symbolic execution in general can also be applied to our work.

Finding Trojan messages can be seen as a form of fault injection. We are not the first to use symbolic execution for this purpose. KLEE [27], a popular symbolic execution tool, also has an optional fault injection operation mode. Unlike black-box fault injection, KLEE's mechanism can avoid exploring redundant failure scenarios (e.g., if two different failure codes are handled by the same code in a system under test).

To detect corner-case bugs in distributed systems, SymNet [91] was developed as a symbolic execution engine that supports systems deployed across multiple interconnected virtual machines. In order to find higher-level protocol bugs, approaches such as SymbexNet [93] add support for developers to write protocol specifications, and then use symbolic execution to check if these are ever violated.

Caballero et al. [26] used symbolic execution to find bugs in malware. This type of "adversarial" programs introduce the challenge of handling encryption and obfuscation techniques. To address this, the authors developed a technique that only solves subsets of path constraints and then restitches the individual solutions to obtain a complete input. As many distributed systems use encryption and authentication, many of the techniques proposed by Caballero et al. to bypass such code can also be used for analyzing distributed systems without developer support.

Another use-case of symbolic execution is equivalence testing. Maurer and Brumley [78] propose using symbolic execution to compare different code versions of a program, in order to analyze code patches. They use tandem symbolic execution of the pre- and post-patch versions of a given program and check that the patch only affects the buggy executions it was meant to fix. A similar approach is taken by Person et al. [88] to characterize code changes. In the same spirit, SOFT [67] uses symbolic execution to compare multiple implementations of the same protocol, checking for corner cases where the different versions behave inconsistently.

Closer to our work, PIC [87] uses symbolic execution to check for interoperability bugs in distributed systems. However, they focus on discovering messages that can be generated by a correct sender node, but are rejected by the receiver node (the exact opposite of what we call a Trojan message). Thus, PIC complements our work on Trojan messages.

Corner cases in a distributed system can be triggered not only by the contents of messages, but also by their timing. MODIST [104] is a technique that systematically explores different reorderings of events.

2.4 Instrumentation

After a program is released in a production environment, instrumentation can help reduce the impact of remaining bugs. By inserting additional code in a program under test, developers can check for undesired behaviors or can log state to ease debugging. In this section we describe how instrumentation can help with reliability, and techniques to reduce the overhead of the additional code in order to make it practical for use in production.

2.4.1 Instrumentation Tools

The idea of instrumenting programs with additional code to improve reliability and traceability has a long history. Alan Turing is credited with proposing the idea of adding assertions to programs to simplify their verification [100]. In the 1980's, Eric Allman invented the syslog logging standard, which provided a unified mechanism for programs to produce log messages. From the early 1990's, systems like RTCC [94] allowed to automatically add instrumentation to programs (safety checks to find illegal memory accesses in this case).

Nowadays, automatic instrumentation tools are used for a wide variety of purposes such as detecting memory access errors in languages such as C [94, 55, 85, 15, 14, 39, 84, 41], detecting concurrency errors [24, 76, 92], catching undefined behavior like integer overflows [33], diagnosing the cause of bugs [72, 20, 105, 106, 45], finding performance problems [38], etc.

2.4.2 Reducing Instrumentation Overhead

An instrumentation tool's overhead is a crucial factor affecting its widespread use [96]. Therefore, many techniques have been developed to reduce overhead. A frequently used method is to *weaken* the instrumentation. For example, [15, 14] use memory safety checks that do allow some illegal accesses to happen, but require only little metadata for book keeping and are fast to compute. StackProtector [41] checks for stack-based buffer overflows only, a common type of error that can be detected with <1% performance impact.

Another approach uses *sampling* to run only a small fraction of the inserted instrumentation code. For bug diagnosis, sampling can produce enough evidence for statistical techniques to pinpoint the bug [72]. In other cases, sampling does reduce a tool's effectiveness, but is necessary because overheads would otherwise be prohibitive [19, 24]

The LiteRace data race detector [76] uses sampling, but attempts to maximize the benefit of instrumentation by biasing sampling towards cold code. The authors write that, because memory accesses in cold code are more likely to participate in a data race, they can detect 70% of all data races in a program by instrumenting only 2% of its memory accesses. Similarly, StackProtector can optionally trade some protection for performance by focusing on functions with a high risk of buffer overflows. In this case, it analyzes the function's code and only instruments functions that store arrays on the stack or access stack variables through pointers.

We propose a generalization of these schemes. Like them, we recognize that 100% instrumentation is often not practical. Unlike them, we measure both cost and benefit of each independent piece of instrumentation, and use an optimization algorithm to select the best subset that fits the users' constraints.

Our approach can be seen as minimizing the amount of code that is not covered by instrumentation, subject to performance constraints. This is similar to techniques like AppArmor [13], SubDomain [36] or Kernel Tailoring [64] which minimize attack surface by blocking access to unused APIs. The difference is that in our approach, whether code is protected/unprotected or used/unused is not a binary property, but a continuous range where the best trade-off can be chosen.

The Split Kernel project [65] dynamically enables extra security checks in the Linux kernel when running untrusted user-level applications. This is similar to our evaluation of Elastic Instrumentation on the Linux kernel section 4.5. We consider Elastic Instrumentation as a generalization of the Split Kernel proposal. Users of our framework can obtain selective instrumentation with relatively little effort (our evaluation of the Linux kernel required less than 20 lines of code to be manually added to the kernel).

ASAP [101] reduces the overhead of instrumentation approaches that add safety checks to programs. It depends on profiling to measure the cost of these safety checks, and removes the most expensive ones from the program. In contrast, we propose to measure the benefit as well as the cost of instrumentation, and generalize this approach to instrumentation beyond safety checks. Our framework also allows to enable and disable instrumentation at runtime, thus reducing the need for a profiling workload.

3 Automated Black-Box Fault Exploration

3.1 Intuition

In this chapter, we describe an algorithm that systematically executes fault injection scenarios in a system under test. The algorithm navigates through a fault space that defines all possible failure scenarios. Rather than choosing them randomly, our algorithm selects fault injection scenarios based on their expected impact.

We observe empirically that, often, there are some patterns in the distribution of fault impact over the fault space, as suggested by Fig. 3.1, which shows the impact of library-level faults on the `ls` utility in UNIX. The patterns that emerge in the fault space are caused by structure and modularity in the underlying code of the system being tested. Engler et al. [40] made a similar observation of bug patterns, which emerge due to implementation-based correlations between bodies of code. When viewing the fault space as a search space, this structure can help make the search more efficient than random sampling; when describing our algorithm (section 3.3), we draw an analogy between fault space exploration and the Battleship game.

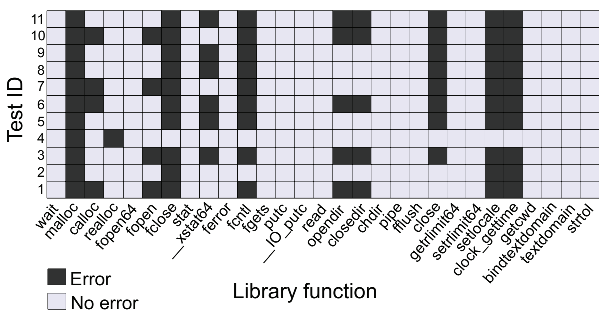

Figure 3.1 – Part of the fault space created by LFI [74] for the `ls` utility. The horizontal axis represents functions in the C standard library that fail, and the vertical axis represents the tests in the default test suite for this utility. A point (x, y) in the plot is black if failing the first call to function x while running test y leads to a test failure, and is gray otherwise.

3.2 Definitions

We start by defining three concepts that are central to this work: fault space, fault impact, and fault space structure.

Fault Space A fault space is a concise description of the failures that a fault injector can simulate in a target system's environment. A fault injection tool T defines implicitly a fault space by virtue of the possible values its parameters can take. For example, a library-level fault injector can inject a variety of faults at the application–library interface—the library call in which to inject, the error code to inject, and the call number at which to inject represent three axes describing the universe of library-level faults injectable by T. We think of a fault space as a hyperspace, in which a point represents a fault defined by a combination of parameters that, when passed to tool T, will cause that fault to be simulated in the target system's environment. This hyperspace may have holes, corresponding to invalid combinations of parameters to T.

We define the attributes of a fault ϕ to be the parameters to tool T that cause ϕ to be injected. If Φ is the space of possible faults, then fault $\phi \in \Phi$ is a vector of attributes $< \alpha_1, ..., \alpha_N >$ where α_i is the value of the fault's i-th attribute. For example, in the universe of failed calls made by a program to POSIX library functions, the return of error code -1 by the 5-th call to `close` would be represented as $\phi =< \mathsf{close}, 5, -1 >$. The values of fault attributes are taken from the finite sets $A_1, ..., A_N$, meaning that, for any fault $\phi =< \alpha_1, ..., \alpha_N >\in \Phi$, the attribute value $\alpha_i \in A_i$.

In order to lay out the values contained in each A_i along an axis, we assume the existence of a total order $\prec_i$ on each set A_i, so that we can refer to attribute values by their index in the corresponding order. In the case of $\phi =< \mathsf{close}, 5, -1 >$, we can assume A_1 is ordered as $(\mathsf{open}, \mathsf{close}, \mathsf{read}, \mathsf{write}, ...)$, A_2 as $(1, 2, 3, ...)$, and A_3 as $(-1, 0, ...)$. If there is no intrinsic total order, then we can pick a convenient one (e.g., group POSIX functions by functionality: file, networking, memory, etc.), or simply choose an arbitrary one.

We now define a *fault space* Φ to be spanned by axes $X_1, X_2, ...X_N$, meaning $\Phi = X_1 \times X_2 \times .. \times X_N$, where each axis X_i is a totally ordered set with elements from A_i and order $\prec_i$. A fault space represents all possible combinations of values from sets $A_1, ..., A_N$, along with the total order on each such set. Using the example shown above, the space of failed calls to POSIX functions is spanned by three axes, one for call types $X_1 : (\mathsf{open} \prec \mathsf{close} \prec ...)$, one for the index of the call made by the caller program to the failed function $X_2 : (1 \prec 2 \prec ...)$, and one for the return value of the POSIX function $X_3 : (-1 \prec 0 \prec ...)$. This enables us to represent fault $\phi =< \mathsf{close}, 5, -1 >$ as $\phi =< 2, 5, 1 >$, because the index of close on axis X_1 under order $\prec_1$ is 2, the index of 5 on X_2 under $\prec_2$ is 5, and the index of -1 on X_3 under $\prec_3$ is 1. A fault space can have holes corresponding to invalid faults, such as close returning 1.

Impact Metric Our proposed approach uses the effect of past injected faults to guide the search for new, higher impact faults. Therefore, we need an *impact metric* that quantifies the change effected by an injected fault in the target system's behavior (e.g., the change in number of requests per second served by Apache when random TCP packets are dropped). Conceptually, an impact metric is a function $I_S : \Phi \to \mathbb{R}$ that maps a fault ϕ in the fault space Φ to a measure of the impact that fault ϕ has on system S. Since Φ is a hyperspace, I_S defines a hypersurface. The (adversarial) goal of a tester employing fault injection is to find peaks on this hypersurface, i.e., find those faults that have the largest impact on the behavior of the system under test. Since testing is often a race against time, the tester needs to find as many such peaks as possible in the allotted time budget.

Fault Space Structure To characterize the structure of a fault space, we use a *relative linear density* metric ρ as follows: given a fault $\phi =< \alpha_1^0, ..., \alpha_k^0, ..., \alpha_N^0 >$, the relative linear density at ϕ along an axis X_k is the average impact of faults $\phi' =< \alpha_1^0, ..., \alpha_k, ..., \alpha_N^0 >$ with the same attributes as ϕ except along axis X_k, scaled by the average impact of all faults in the space. Specifically, $\rho_\phi^k = \frac{avg[\ I_S(<\alpha_1^0,...,\alpha_k,...,\alpha_N^0>),\ \alpha_k \in X_k\ ]}{avg[\ I_S(\phi_x),\ \phi_x \in \Phi\ ]}$. If $\rho_\phi^k > 1$, walking from ϕ along the X_k axis will encounter more high-impact faults than along a random direction. In practice, it is advantageous to compute ρ_ϕ over only a small vicinity of ϕ, instead of the entire fault space. This vicinity is a subspace containing all faults within distance D of ϕ, i.e., all faults ϕ'' s.t. $\delta(\phi, \phi'') \leq D$. Distance $\delta : \Phi \times \Phi \to \mathbb{N}$ is a Manhattan (or city-block) distance [23], i.e., the shortest distance between fault ϕ and ϕ'' when traveling along Φ's coordinate axes. $\delta(\phi, \phi'')$ gives the smallest number of increments/decrements of attribute indices that would turn ϕ into ϕ''. Thus, the D-vicinity of ϕ consists of all faults that can be obtained from ϕ with no more than D increments/decrements of ϕ's attributes $\alpha_j, 1 \leq j \leq N$.

To illustrate, consider fault $\phi =< \text{fclose}, 7 >$ in Fig. 3.1, and its 4-vicinity (the faults within a distance $\delta \leq 4$). If the impact corresponding to a black square is 1, and to a gray one is 0, then the relative linear density at ϕ along the vertical axis is $\rho_\phi^2 = 2.27$. This means that walking in the vertical direction is more likely to encounter faults that cause test errors than walking in the horizontal direction or diagonally. In other words, exploration along the vertical axis is expected to be more rewarding than random exploration.

3.3 Design

Fault exploration "navigates" a fault space in search of faults that have high impact on the system being tested. Visiting a point in the fault space incurs a certain cost corresponding to the generation of a test, its execution, and the subsequent measurement of the impact. Thus, ideally, one would aim to visit as few points as possible before finding a desired fault.

Exhaustive exploration (as used for instance by Gunawi et al. [50]) iterates through every point in the fault space by generating all combinations of attribute values, and then evaluates the impact of the corresponding faults. This method is complete, but inefficient and, thus, prohibitively slow for large fault spaces. Alternatively, random exploration [11] constructs random combinations of attribute values and evaluates the corresponding points in the fault space. When the fault space has no structure, random sampling of the space is no less efficient than any other form of sampling. However, if structure is present (i.e., the relative linear density is non-uniformly distributed over the fault space), then there exist more efficient search algorithms.

Fitness-guided Exploration We propose a fitness-guided algorithm for searching the fault space. This algorithm uses the impact metric I_S as a measure of a fault's "fitness," in essence steering the search process toward finding "ridges" on the hypersurface defined by (Φ, I_S) and then following these ridges to the peaks representing high-impact faults. Exploring a structured fault space like the one in Fig. 3.1 is analogous to playing Battleship, a board game involving two players, each with a grid. Before play begins, each player arranges a number of ships secretly on his/her own grid. After the ships have been positioned, one player announces a target square in the opponent's grid that is to be shot at; if a ship occupies that square, then it takes a hit. Players take turns and, when all of a ship's squares have been hit, the ship is sunk. A typical strategy is to start by shooting randomly until a target is hit, and then fire in the neighborhood of that target to guess the orientation of the ship and sink it.

Similarly, our algorithm exploits structure to avoid sampling faults that are unlikely to be interesting, and instead focus on those that are near other faults with high impact. For example, if a developer has a poor understanding of how some I/O library works, then there is perhaps a higher likelihood for bugs to lurk in the code that calls that library than in code that does not. Once the search algorithm finds a call to the I/O library whose failure causes, say, data corruption, it will eventually focus on failing other calls to that same library, persisting for as long as it improves the achieved fault impact. Note that the algorithm focuses on related but distinct bugs, rather than on multiple instances of the same bug. If the fault space definition contains duplicates (i.e., different points in the fault space expose the same bug), then the impact should reflect this. We describe means to detect and avoid duplicates in subsection 3.5.4, and evaluate these approaches in subsection 3.6.5.

Another perspective on the search algorithm is that impact-guided exploration merely generates a priority in which tests should be executed—if each fault in Φ corresponds to a test, the question is which tests to run first. The exploration algorithm both generates the tests and executes them; to construct new tests, it mutates previous high-impact tests in ways that it believes will further increase their impact. It also avoids re-executing any tests that have already been executed in the past. Since it does not discard any tests, rather only prioritizes their execution, the algorithm's coverage of the fault space increases proportionally to the allocated time budget.

Our proposed exploration algorithm is similar to the Battleship strategy mentioned earlier, except it is fully automated, so there are no human decisions or actions in the critical path. The search algorithm consists of the following steps:

1. Generate an initial batch of tests randomly, execute the tests, and evaluate their impact
2. Choose a previously executed high-impact test ϕ
3. Modify one of ϕ's attributes (injection parameters) to obtain a new test ϕ'
4. Execute ϕ' and evaluate its impact
5. Repeat step 2

In order to decide which test attribute to mutate, we could rely on the linear density metric to suggest the highest-reward axis. However, given that the fault space is not known a priori, we instead compute dynamically a sensitivity (described later) based on the historical benefit of choosing one dimension vs. another. This sensitivity calculation steers the search to align with the fault space structure observed thus far, in much the same way a Battleship player infers the orientation of her opponent's battleships.

When deciding by how much to mutate an attribute (i.e., the magnitude of the increment), we choose a value according to a Gaussian distribution, rather than a static value, in order to keep the algorithm robust. This distribution favors ϕ's closest neighbors without completely dismissing points that are further away. By changing the standard deviation of the Gaussian distribution, we can control the amount of bias the algorithm has in favor of nearby points.

Finally, we use an "aging" mechanism among the previously executed tests: the fitness of a test is initially equal to its impact, but then decreases over time. Once the fitness of old tests drops below a threshold, they are retired and can never have offspring. The purpose of this aging mechanism is to encourage improvements in test coverage concomitantly with improvements in impact—without aging, the exploration algorithm may get stuck in a high-impact vicinity, despite not finding any new high-impact faults. In the extreme, discovering a massive-impact "outlier" fault with no serious faults in its vicinity would cause an algorithm with no aging to waste time exploring exhaustively that vicinity.

Algorithm 1 embodies steps 2–3 of the search algorithm. We now describe Algorithm 1 in more detail.

Choosing Which Test to Mutate The algorithm uses three queues: a priority queue $Q_{priority}$ of already executed high-impact tests, a queue $Q_{pending}$ of test cases that have been generated but not yet executed, and a set *History* containing all previously executed tests. Once a test in $Q_{pending}$ is executed and its impact is evaluated, it gets moved to $Q_{priority}$. $Q_{priority}$ has a limited size; whenever the limit is reached, a test case is dropped from the queue, sampled with a probability inversely proportional to its fitness (tests with low fitness have a higher probability of being dropped). As a result, the average fitness of tests in $Q_{priority}$ increases over time. When old test cases are retired from $Q_{priority}$, they go into *History*. This history set allows the algorithm to avoid redundant re-execution of already evaluated tests.

```
Data: $Q_{priority}$: priority queue of high-fitness fault
          injection tests (already executed)
      $Q_{pending}$: queue of tests awaiting execution
      $History$: set of all previously executed tests
      $Sensitivity$: vector of $N$ sensitivity values, one for
          each test attribute $\alpha_1, \ldots \alpha_N$
1  foreach fault injection test $\phi_x \in Q_{priority}$ do
2      $testProbs[\phi_x] := assignProbability(\phi_x.fitness)$
3  end
4  $\phi := sample(Q_{priority}, testProbs)$
5  $attributeProbs := normalize(Sensitivity)$
6  $\alpha_i := sample(\{\alpha_1, \ldots, \alpha_N\}, attributeProbs)$
7  $oldValue := \phi.\alpha_i$                    // remember that oldValue $\in A_i$
8  $\sigma := chooseStdDev(\phi, A_i)$
9  $newValue := sample(A_i, Gaussian(oldValue, \sigma))$
10 $\phi' := clone(\phi)$
11 $\phi'.\alpha_i := newValue$
12 if $\phi' \notin History \wedge \phi' \notin Q_{priority}$ then
13     $Q_{pending} := Q_{pending} \cup \phi'$
14 end
```

Algorithm 1: Fitness-guided generation of the next test. Execution of tests, computation of fitness and sensitivity, and aging occur outside this algorithm.

On lines 1–4, the algorithm picks a parent test ϕ from $Q_{priority}$, to mutate into offspring ϕ'. Instead of always picking the highest fitness test, it samples $Q_{priority}$ with a probability proportional to fitness—highest fitness tests are favored, but others still have a non-zero chance to be picked.

Mutating the Test The new test ϕ' is obtained from parent test ϕ by modifying one of ϕ's attributes.

On lines 5–6, we choose the fault/test attribute α_i with a probability proportional to axis X_i's normalized sensitivity. We use *sensitivity* to capture the history of fitness gain: the sensitivity of each axis X_i of the fault space reflects the historical benefit of modifying attribute α_i when generating a new test. This sensitivity is directly related to relative linear density (from section 3.2): the inherent structure of the system under test makes mutations along one axis to be more likely to produce high-impact faults than along others. In other words, if there is structure in the fault space, the sensitivity biases future mutations to occur along high-density axes. Given a value n, the sensitivity of X_i is computed by summing the fitness value of the previous n test cases in which attribute α_i was mutated. This sum helps detect "impact ridges" present in the currently sampled vicinity: if X_i's density is high, then we expect this sum of previous fitness values—the sensitivity to mutations along X_i—to be high as well, otherwise not. Our use of sensitivity is similar to the fitness-gain computation in Fitnex [103], since it essentially corresponds to betting on choices that have proven to be good in the past.

Sensitivity guides the choice of *which* fault attribute to mutate; next we describe *how* to mutate the chosen attribute in order to obtain a new fault injection test.

On lines 7–9, we use a discrete approximation of a Gaussian probability distribution to choose a new value for the test attribute to be mutated. This distribution is centered at *oldValue* and has standard deviation σ. The chosen standard deviation is proportional to the number of values the α_i attribute can take, i.e., to the cardinality of set A_i. For our evaluation, we chose $\sigma = \frac{1}{5} \cdot |A_i|$. σ can also be computed dynamically, based on the evolution of tests in the currently explored vicinity within the fault space—we leave the pursuit of this alternative to future work.

Our use of a Gaussian distribution implicitly assumes that there is some similarity between neighboring values of a test attribute. This similarity of course depends on the meaning of attributes (i.e., parameters to a fault injector) and on the way the human tester describes them in the fault space. In our experience, many parameters to fault injectors do have such similarity and, by using the Gaussian distribution, we can make use of this particularity to further improve on the naive method of randomly choosing a new attribute value. Revisiting the example from section 3.2, it is not surprising that there is correlation between library functions (e.g., close is related to open), call numbers (e.g., successive calls from a program to a given function are likely to do similar things), or even tests from a suite (e.g., they are often grouped by functionality). Profiling tools, like LibTrac [22], can be used to discover such correlation when defining the fault space.

Finally, Algorithm 1 produces ϕ' by cloning ϕ and replacing attribute α_i with the new value (lines 10–11). If ϕ' has not been executed before, it goes on $Q_{pending}$ (lines 12–14).

Alternative Algorithms The goal of our search algorithm was to have an automated general-purpose fault exploration system that is not tied to any particular fault injection tool. We evaluate it using a library-level fault injection tool and a bit corruption tool, but believe it to be equally suitable to other kinds of fault injection, such as flipping bits in data structures [99] or injecting human errors [59].

In an earlier version of our system, we employed a genetic algorithm [89], but abandoned it, because we found it inefficient. The search algorithm aims to optimize for "ridges" on the fault-impact hypersurface, and this makes global optimization algorithms (such as genetic algorithms) difficult to apply. Moreover, the hypersurface can be dynamic: in some of our experiments we consider code coverage as part of the impact metric, and, thus, exploring a point changes the shape of the surface (the impact of other injection scenarios that execute the same recovery code is reduced). The algorithm we present here is, in essence, a variation of stochastic beam search [89]—parallel hill-climbing with a common pool of candidate states—enhanced with sensitivity analysis and Gaussian value selection.

3.4 Developer Trade-Offs in Defining the Fault Space

Leveraging Domain Knowledge By default, our algorithm is meant for purely black-box mode exploration, in that it has no a priori knowledge of the specifics of the system under test or its environment. This makes it a good fit for generic testing, such as that done in a certification service [28].

However, developers often have significant amounts of domain knowledge about the system or its environment, and this could enable them to reduce the size of the fault space, thus speeding up exploration. For example, when using a library-level fault injector and an application known to use only blocking I/O with no timeouts, it makes sense to exclude `EAGAIN` from the set of possible values of the `errno` attribute of faults injected in `read`. In section 4.2 we describe how developers can add domain knowledge about the system under test, the fault space, and/or the tested system's environment; we evaluate in subsection 3.6.6 the benefit of doing so.

The exploration algorithm can also benefit from static analysis tools, which provide a complementary method for detecting vulnerable injection points (LFI's callsite analyzer [75] is such a tool). For example, the results of the static analysis can help the initial generation phase of test candidates. By starting with highly relevant tests from the beginning, the search algorithm can quickly learn the structure of the fault space, which is likely to boost its efficiency. This increase in efficiency can manifest as finding high impact faults sooner, as well as finding additional faults that were not suggested by the static analysis.

Injection Point Precision An injection point is the location in the execution of a program where a fault is to be injected. Even though the algorithm has no knowledge of how injection points are defined, the definition does affect its accuracy and speed.

In our evaluation we use LFI [74], a library fault injection tool that allows the developer to fine-tune the definition of an injection point according to their own needs. Since we aim to maximize the accuracy of our evaluation, we define an injection point as the tuple $\langle$ *testID, functionName, callNumber* $\rangle$. *testID* identifies a test from the test suite of the target system (in essence specifying one execution path, modulo non-determinism), *functionName* identifies the called library function in which to inject an error, and *callNumber* identifies the cardinality of the call to that library function that should fail. This level of precision ensures that injection points are unique on each tested execution path, and all possible library-level faults can be explored. However, this produces a large fault space, and introduces the possibility of test redundancy (we will show how AFEX explores this space efficiently in subsection 3.6.2 and mitigates redundancy in subsection 3.6.5).

A simple way to define an injection point is via the callsite, i.e., file and line number where the program is to encounter a fault (e.g., failed call to a library or system call, memory operation). Since the same callsite may be reached on different paths by the system under test, this definition is relatively broad. A more accurate definition is the "failure ID" described by Gunawi et al. [50], which associates a stack trace and domain-specific data (e.g., function arguments or state of system under test) to the definition of the injection point. This definition is more tightly related to an individual execution path and offers more precision. However, unlike the 3-tuple definition we use in our evaluation, failure IDs ignore loops and can lead to missed bugs.

The trade-off in choosing injection points is one between precision of testing and size of the resulting fault space: on the one hand, a fine-grain definition requires many injection parameters (fault attributes) and leads to a large fault space (many possible combinations of attributes), which takes longer to explore. On the other hand, more general injection points reduce the fault space, but may miss important fault scenarios (i.e., incur false negatives).

For example, the first MySQL bug we describe in subsection 3.6.2 is triggered when injecting a fault in a call to LibC's *close* from *my_close*, a low-level I/O function. *my_close* is called from several different places in MySQL. However, the bug we discovered only manifests in *mi_create*, one of the many functions that may call *my_close*. Thus, injecting in *my_close* in a single test scenario may miss the bug if the injection is not performed on an execution path in which *mi_create* is the caller.

3.5 A Prototype Implementation

We built AFEX, a prototype that embodies the techniques presented in this chapter. The user provides AFEX with a description of the explorable fault space Φ, dictated by the available fault injectors, along with scripts that start/stop/measure the system under test S. AFEX automatically generates tests that inject faults from Φ and evaluates their quality. Our prototype is a parallel system that runs on clusters of computers, thus taking advantage of the parallelism inherent in AFEX.

The core of AFEX consists of an explorer and a set of node managers, as shown in Fig. 3.2. The explorer receives as input a fault space description and an exploration target (e.g., find the top-10 highest impact faults), and produces a set of fault injection tests that satisfy this target. AFEX synthesizes configuration files for each injection tool and instructs the various node managers to proceed with the injection of the corresponding faults. The managers then report to the explorer the results of the injections, and, based on this, the explorer decides which faults from Φ to inject next.

Each of the managers is associated with several fault injectors and sensors. One manager is in charge of all tests on one physical machine. When the manager receives a fault scenario from the explorer (e.g., "inject an EINTR error in the third read socket call, and an ENOMEM error in the seventh malloc call"), it breaks the scenario down into atomic faults and instructs the corresponding injectors to perform the injection. The sensors are instructed to run the developer-provided workload scripts (e.g., a benchmark) and perform measurements, which are then reported back to the manager. The manager aggregates these measurements into a single impact value and returns it to the explorer.

The goal of a sequence of such injections—a fault exploration session—is to produce a set of faults that satisfy a given criterion. For example, AFEX can be used to find combinations of faults that cause a database system to lose or corrupt data. As another example, one could obtain the top-50 worst faults performance-wise (i.e., faults that affect system performance the most). Prior to AFEX, this kind of test generation involved significant amounts of human labor.

To further help developers, AFEX presents the faults it found as a map, clustered by the degree of redundancy with respect to code these faults exercise in the target *S*. For each fault in the result set, AFEX provides a generated script that can run the test and replay the injection. Representatives of each redundancy cluster can thus be directly assembled into (or inserted into existing) regression test suites.

3.5.1 Architecture

Since tests are independent of each other, AFEX enjoys "embarrassing parallelism." Node managers need not talk to each other, only the explorer communicates with node managers. Given that the explorer's workload (i.e., selecting the next test) is significantly less than that of the managers (i.e., actually executing and evaluating the test), the system has no problematic bottleneck for clusters of dozens of nodes, maybe even larger. In this section, we provide more details on our prototype's two main components, the explorer and the node manager.

Explorer The AFEX explorer is the main control point in the exploration process. It receives the fault space description and the search target, and then searches the fault space for tests to put in the result set. The explorer can navigate the fault space in three ways: using the fitness-guided Algorithm 1, exhaustive search, or random search.

Node Manager The node manager coordinates all tasks on a physical machine. It contains a set of plugins that convert fault descriptions from the AFEX-internal representation to concrete configuration files and parameters for the injectors and sensors. Each plugin, in essence, adapts a subspace of the fault space to the particulars of its associated injector. The actual execution of tests on the system *S* is done via three user-provided scripts: A startup script prepares the environment (setting up workload generators, necessary environment variables, etc.). A test script starts up *S* and signals the injectors and sensors to proceed; they in turn will report results to the manager. A cleanup script shuts *S* down after the test and removes all side effects of the test.

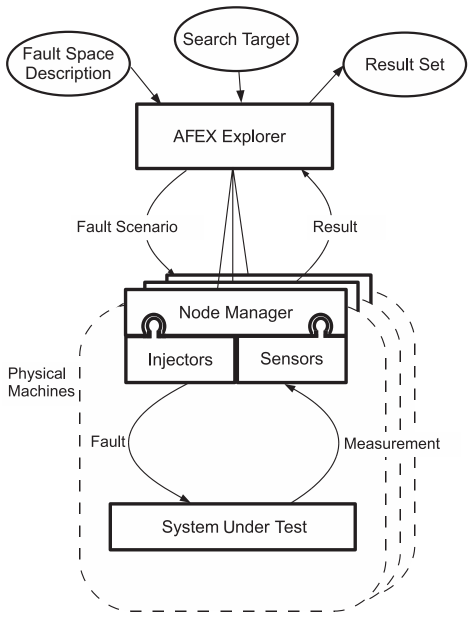

Figure 3.2 – AFEX prototype architecture: an explorer coordinates multiple managers, which in turn coordinate the injection of faults and measurement of the injections' impact on the system under test.

```
syntax        = {space};
space         = (subtype | parameter )+";";
subtype       = identifier;
parameter     = identifier ":"
( "{" identifier ( "," identifier )+ "}" |
  "[" number "," number "]"              |
  "<" number "," number ">"
);
identifier    =  letter ( letter | digit | "_" )*;
number        = (digit)+;
```

Figure 3.3 – AFEX fault space description language.

3.5.2 Input

AFEX takes as input descriptions of the fault spaces to be explored, sensor plugins to measure impact metrics (which AFEX then uses to guide fault exploration), and search targets describing what the user wants to search for, in the form of thresholds on the impact metrics.

Fault Description Language The language for describing fault spaces must be expressive enough to allow the definition of complex fault spaces, but succinct and easy to use and understand. Fig. 3.3 shows the grammar of the fault space description language used in AFEX.

```
function   : { malloc, calloc, realloc }
errno      : { ENOMEM }
retval     : { 0 }
callNumber : [ 1 , 100 ] ;

function   : { read }
errno      : { EINTR }
retVal     : { -1 }
callNumber : [ 1 , 50 ] ;
```

Figure 3.4 – Example of a fault space description.

```
function malloc errno ENOMEM retval 0
callNumber 23
```

Figure 3.5 – Example of a fault scenario description.

Fault spaces are described as a Cartesian product of sets, intervals, and unions of subspaces (subtypes). Subspaces are separated by ";". Sets are defined with "{ }". Intervals are defined using "[ ]" or "< >". The difference between these two is that, during fault selection, intervals marked with "[ ]" are sampled for a single number, while intervals marked with "< >" are sampled for entire sub-intervals (we say < 5,10 > is a sub-interval of < 1, 50 >).

Fig. 3.4 shows an example representing the fault space for a library fault injection tool. This fault space is a union of two hyperspaces, separated by ";". The injection can take place in any of the first 100 calls to memory allocation, or in any of the first 50 calls to read. One possible fault injection scenario the explorer may sample from this fault space is shown in Fig. 3.5. This scenario would be sent by the explorer to a node manager for execution and evaluation.

3.5.3 Output

AFEX's output consists of a set of faults that satisfy the search target, a characterization of the quality of this fault set, and generated test cases that inject the faults of the result set into the system under test *S* and measure their impact on *S*. In addition to these, AFEX also reports operational aspects, such as a synopsis of the search algorithms used, injectors used, CPU/memory/network resources used, exploration time, number of explored faults, etc.

Quality Characterization: The quality characterization provides developers with additional information about the tests, in order to help guide their attention toward the most relevant ones. The two main quality metrics are redundancy and repeatability/precision, described in subsection 3.5.4. AFEX aims to provide a confidence characterization, similar to what a Web search engine user would like to get with her search results: which of the returned pages have the same (or similar) content. A practical relevance evaluation of the fault set is optionally available if the developer can provide the corresponding statistical model (subsection 3.5.4).

Test Suites: AFEX automatically generates test cases that can directly reproduce each fault injection and the observed impact on the system. Each test case consists of a set of configuration files for the system under test, configuration of the fault injector(s), a script that starts the system and launches the fault injector(s), and finally a script that generates workload on the system and measures the faults' impact. We find these generated test scripts to save considerable human time in constructing regression test suites.

3.5.4 Quantifying Result Quality

In addition to automatically finding high-impact faults, AFEX also quantifies the level of confidence users can have in its results. We consider three aspects of interest to practitioners: cutting through redundant tests (i.e., identifying equivalence classes of redundant faults within the result set), assessing the precision of our impact assessment, and identifying which faults are representative and practically relevant.

Redundancy Clusters One measure of result quality is whether the different faults generated by AFEX exercise diverse system behaviors—if two faults exercise the same code path in the target system, it is sufficient to test with only one of the faults. Being a black-box testing system, AFEX cannot rely on source code to identify redundant faults.

AFEX computes clusters (equivalence classes) of closely related faults as follows: While executing a test that injects fault ϕ, AFEX captures the stack trace corresponding to ϕ's injection point. Subsequently, it compares the stack traces of all injected faults by computing the edit distance between every pair of stack traces (specifically, we use the Levenshtein distance [71]). Any two faults for which the distance is below a threshold end up in the same cluster. In subsection 3.6.5 we evaluate the efficiency of this technique in avoiding test redundancy. In choosing this approach, we were inspired by the work of Liblit and Aiken on identifying which parts of a program led to a particular bug manifestation [72].

Besides helping developers to analyze fault exploration results, redundancy clusters are also used online by AFEX itself, in a feedback loop, to steer fault exploration away from test scenarios that trigger manifestations of the same underlying bug. This improves the efficiency of exploration.

Impact Precision Impact precision indicates, for a given fault ϕ, how likely it is to consistently have the same impact on the system under test S. To obtain it, AFEX runs the same test n times (with n configured by the developer) and computes the variance $Var(I_S(\phi))$ of ϕ's impact across the n trials. The impact precision is $\frac{1}{Var(I_S(\phi))}$, and AFEX reports it with each fault in the result set. The higher the precision, the more likely it is that re-injecting ϕ will result in the same impact that AFEX measured. In other words, a high value for impact precision suggest that the system's response to that fault in that environment is likely to be deterministic. Developers may find it easier, for example, to focus on debugging high-precision (thus reproducible) failure scenarios.

Practical Relevance The final quality metric employed by AFEX is a measure of each fault's representativeness and, thus, practical relevance. Using published studies [95, 21] or proprietary studies of the particular environments where a system will be deployed, developers can associate with each class of faults a probability of it occurring in practice. Using such statistical models of faults, AFEX can automatically associate with each generated fault a probability of it occurring in the target environment. This enables developers to better choose which failure scenarios to debug first.

3.5.5 Extensibility and Control

The AFEX system was designed to be flexible and extensible. It can be augmented with new fault injectors, new sensors and impact metrics, custom search algorithms, and new result-quality metrics.

We present below the steps needed to use AFEX on a new system under test *S*. In our evaluation, adapting AFEX for use on a new target *S* took on the order of hours.

1. *Write fault injection plugins.* These are small Java code snippets required to wrap each fault injector tool to be used (~150 lines of code).
2. *Choose fault space.* The developer must write a fault space descriptor file (using the language specified in Fig. 3.3). We found that the simplest way to come up with this description is to analyze the target system *S* with a tracer like *ltrace*, or to use a static analysis tool, such as the profiler that ships with LFI [75].
3. *Design impact metric.* The impact metric guides the exploration algorithm. The easiest way to design the metric is to allocate scores to each event of interest, such as 1 point for each newly covered basic block, 10 points for each hang bug found, 20 points for each crash, etc.
4. *Provide domain knowledge.* Optionally, the developer can give AFEX domain knowledge in various ways. For example, if the system under test is to be deployed in a production environment with highly reliable storage, it may make sense to provide a fault relevance model in which I/O faults are deemed less relevant (since they are less likely to occur in practice), unless they have catastrophic impact on *S*. This will discourage AFEX from exploring faults of little interest.
5. *Write test scripts.* Developers must provide three scripts: startup, test, and cleanup. These are simple scripts and can be written in any scripting language supported on the worker

nodes.

6. *Select search target.* The tester can choose to stop the tests after some specified amount of time, after a number of tests executed, or after a given threshold is met in terms of code coverage, bugs found, etc.

7. *Run AFEX.* AFEX is now ready to start. It provides progress metrics in a log, so that developers can follow its execution, if they wish to do so.

8. *Analyze results.* AFEX produces tables with measurements for each test (fitness, quality characterization, etc.), and it identifies a representative test for each redundancy cluster (as described in subsection 3.5.4). AFEX also creates a folder for each test, containing logs, core dumps, or any other output produced during the test.

3.6 Evaluation

In this section, we address the following questions about the AFEX prototype: Does it find bugs in real-world systems (subsection 3.6.2)? How efficient is AFEX exploration compared to random and exhaustive search (subsection 3.6.3)? To what extent does fault space structure improve AFEX's efficiency (subsection 3.6.4)? Can AFEX leverage result-quality metrics to improve its search efficiency (subsection 3.6.5)? To what extent can system-specific knowledge aid AFEX (subsection 3.6.6)? How does AFEX's usefulness vary across different stages of system development (subsection 3.6.7)? How well does the AFEX prototype scale (subsection 3.6.8)?

3.6.1 Setup

Evaluation Targets Most of our experiments focus on four real-world code bases: the MySQL 5.1.44 database management system, the Apache httpd 2.3.8 Web server, the coreutils 8.1 suite of UNIX utilities, and PBFT (sfslite version). These systems range from large ($> 10^6$ lines of code in MySQL) to small ($\sim 10^3$ lines of code per UNIX utility). We used AFEX to find new bugs in MySQL and Apache httpd, both mature systems considered to be highly reliable. We reproduced one known bug, and found a bug that we did not know about ahead of time in PBFT. The UNIX utilities, being small yet still real-world, allow us to closer examine various details of AFEX and show how the various ideas described in the chapter come together into a fitness-guided exploration framework that is significantly more efficient than random exploration, while still requiring no source code access. We also report measurements on the MongoDB NoSQL database, versions 0.8 and 2.0.

Fault Space Definition Methodology AFEX can explore both single-fault and multi-fault scenarios, but we limit our evaluation to only single-fault scenarios, which offer sufficient opportunity to examine all aspects of our system. Even this seemingly simple setup produces fault spaces with more than 2 million faults, which are infeasible to explore with brute-force approaches.

Our fault space is defined by the fault injection tools we use, along with profiling tools and, for some specific experiments, general knowledge of the system under test. We used the LFI library-level fault injector [74] for the MySQL, Apache httpd and coreutils experiments. We focused on injecting error returns into calls made to functions in the standard C library, libc.so. This library is the principal way for UNIX programs to interact with their environment, so we can use LFI to simulate a wide variety of faults in the file system, network, and memory. To define the fault space, we first run the default test suites that ship with our test targets, and use the ltrace library-call tracer to identify the calls that our target makes to libc and count how many times each libc function is called. We then use LFI's callsite analyzer, applied to the libc.so binary, to obtain a fault profile for each libc function, indicating its possible error return values and associated errno codes.

We use this methodology to define a fault space for the UNIX coreutils that is defined by three axes: X_{test} corresponds to the tests in the coreutils test suite, with $X_{test} = (1,...,29)$. X_{func} corresponds to a subset of libc functions used during these tests, and their index values give us $X_{func} = (1,...,19)$. Finally, X_{call} corresponds to the call number at which we want to inject a fault (e.g., the *n*-th call to malloc). In order to keep the fault space small enough for exhaustive search to be feasible (which allows us to obtain baseline measurements for comparison), we restrict these values to $X_{call} = (0,1,2)$, where 0 means no injection, and 1 or 2 correspond to the first or second call, respectively. The size of the resulting fault space $\Phi_{coreutils}$ is $29 \times 19 \times 3 = 1,653$ faults. To measure the impact of an injected fault, we use a combination of code coverage and exit code of the test suite. This encourages AFEX to both inject faults that cause the default test suite to fail and to cover as much code as possible.

For the MySQL experiments, we use the same methodology to obtain a fault space with the same axes, except that $X_{test} = (1,...,1147)$ and $X_{call} = (1,...,100)$, which gives us a fault space Φ_{MySQL} with 2,179,300 faults. If we assume that, on average, a test takes 1 minute, exploring this fault space exhaustively would take on the order of 4 CPU-*years*. MySQL therefore is a good example of why leveraging fault space structure is important. We use a similar impact metric to that in coreutils, but we also factor in crashes, which we consider to be worth emphasizing in the case of MySQL.

For the Apache httpd experiments, we have the same fault space axes, but $X_{test} = (1,...,58)$ and $X_{call} = (1,...,10)$, for a fault space Φ_{Apache} of 11,020 possible faults.

For PBFT, we wrote a custom fault injector that corrupts Message Authentication Codes (MAC) in the client nodes. Inspired by Aardvark [35], we set out to check if AFEX can find the Big MAC attack described by Clement et al. [35]. We deployed PBFT on Emulab [102] and set up a fault injector that corrupts the Message Authentication Code (MAC) in the client nodes. AFEX controls three test parameters: which MACs to corrupt, how many correct clients to connect to

PBFT and how many malicious clients to connect to PBFT. Each parameter corresponds to a dimension in the hyperspace explored by AFEX. The dimension corresponding to which MAC to corrupt has $4,096$ values (see below for explanation), the number of correct clients varies between 10 and 250, with an increment of 10 (25 values), while the number of malicious clients is 1 or 2. Thus, AFEX explores a hyperspace with $4,096 * 25 * 2 = 204,800$ possible scenarios.

The parameter describing which MAC to corrupt is a 12-bit-wide bit mask, where bit n decides whether to corrupt or not the (n mod 12)-th call to the `generateMAC` function in the malicious client. In order to implement the *mutateDistance* parameter, the 12-bit number is encoded in Gray code. Thus, a small *mutateDistance* entails choosing a neighboring value (in Gray code, consecutive numbers always differ in only one binary position).

Metrics To assess efficiency of exploration, we count the number of failing tests from the X_{test} axis and analyze the generated coredumps. While imperfect, this metric has the benefit of being objective. Once fault injection becomes more widely adopted in test suites, we expect developers to write fault injection-oriented assertions, such as "under no circumstances should a file transfer be only partially completed when the system stops," in which case one can count the number of failed assertions.

For the PBFT experiments, we also measure the average throughput and latency observed by the *correct* clients.

Experimental Platform The experiments reported in subsection 3.6.2 and subsection 3.6.4–subsection 3.6.6 ran on an Intel quad-core 2.4GHz CPU with 4GB RAM and 7200rpm HDD. The experiments in subsection 3.6.3 ran on 5 small Amazon EC2 instances [16], the ones reported in subsection 3.6.7 on a quad-core Intel 2.3GHz CPU with 16GB RAM and Intel SSD, and the ones reported in subsection 3.6.8 on a range of 1–14 small Amazon EC2 instances. The PBFT results were obtained on Emulab [102].

3.6.2 Effectiveness of Exploration

In this section, we show that AFEX can be used to successfully test the recovery code of large, production-grade software, such as MySQL and the Apache httpd server, with minimal human effort and no access to source code. After running for 24 hours on a small desktop computer, AFEX found 2 new bugs in MySQL, 1 new bug in Apache httpd and 1 new bug in PBFT (albeit we later discovered that the bug was already reported in a different paper).

MySQL After exploring the Φ_{MySQL} fault space for 24 hours, AFEX found 464 fault injection scenarios that cause MySQL to crash. By analyzing the generated core dumps, we found two new bugs in MySQL. The results are summarized in Table 3.1. Comparison to exhaustive search is impractical, as it would take multiple CPU-years to explore all of Φ_{MySQL}.

	MySQL test suite	**Fitness-guided**	**Random**
Coverage	54.10%	52.15%	53.14%
# failed tests	0	1,681	575
# crashes	0	464	51

Table 3.1 – Comparison of the effectiveness of fitness-guided fault search vs. random search vs. MySQL's own test suite.

We find that AFEX's fitness-guided fault search is able to find almost 3× as many failed tests as random exploration, and cause more than 9× as many crashes. Of course, not all these crashes are indicative of bugs—many of them result from MySQL aborting the current operation due to the injected fault. The random approach produces slightly higher general code coverage, but spot checks suggest that the *recovery* code coverage obtained by fitness-guided search is better. Unfortunately, it is impractical to estimate recovery code coverage, because this entails manually identifying each block of recovery code in MySQL. We now describe briefly the two bugs found by AFEX.

The first one [82] is an example of buggy error recovery code—the irony of recovery code is that it is hard to test, yet, when it gets to run in production, it cannot afford to fail. Since MySQL places great emphasis on data integrity, it has a significant amount of recovery code that provides graceful handling of I/O faults. However, in the code snippet shown in Fig. 3.6, the recovery code itself has a bug that leads to an abort. It occurs in a function that performs a series of file operations, any of which could fail. There is a single block of error recovery code (starting at line mi_create.c:836) to which all file operations in this function jump whenever they fail. The recovery code performs some cleanup, including releasing the THR_LOCK_myisam lock (on line 837), and then returns an error return code up the stack. However, if it is the call to my_close (on line 831) that fails, say due to an I/O error, then the recovery code will end up unlocking THR_LOCK_myisam twice and crashing.

```
mi_create.c
...
830: pthread_mutex_unlock(&THR_LOCK_myisam);
831: if (my_close(file,MYF(0)))
       goto err;
...
836: err:
837:   pthread_mutex_unlock(&THR_LOCK_myisam);
```

Figure 3.6 – Buggy recovery code in MySQL.

The second MySQL bug is a crash that happens when a read from errmsg.sys fails. This bug is a new manifestation of a previously discovered bug [81] that was supposedly fixed. MySQL has recovery code that checks whether the read from errmsg.sys was successful or not, and it correctly logs any encountered error if the read fails. However, after completing this recovery, regardless of whether the read succeeded or not, MySQL proceeds to use a data structure that should have been initialized by that read. This leads to MySQL crashing.

It is worth noting that MySQL does use an (admittedly ad-hoc) method for fault injection testing. The code has macros that override return values and errno codes after calls to libc, in order to simulate faults. These macros are enabled by recompiling MySQL with debug options. However, doing fault injection this way is laborious, because the macros need to be manually placed in each location where a fault is to be injected, and the specifics of the injected fault need to be hardcoded. This explains, in part, why MySQL developers appear to have lacked the human resources to test the scenarios that AFEX uncovered. It also argues for why AFEX-style automated fault exploration is useful, in addition to the fact that AFEX allows even testers unfamiliar with the internal workings of MySQL—such as ourselves—to productively test the server in black-box mode.

Apache httpd We let AFEX explore the Φ_{Apache} fault space of 11,020 possible faults, and we stopped exploration after having executed 1,000 tests. Using fitness-guided exploration, AFEX found 246 scenarios that crash the server; upon closer inspection, we discovered one new bug.

Table 3.2 summarizes the results. When limited to 1,000 samplings of Φ_{Apache}, AFEX finds $3\times$ more faults that fail Apache tests and almost $12\times$ more tests that crash Apache, compared to random exploration. It finds 27 manifestations of the bug in Figure 3.7, while random exploration finds none.

	Fitness-guided	**Random**
# failed tests	736	238
# crashes	246	21

Table 3.2 – Effectiveness of fitness-guided fault search vs. random search for 1,000 test iterations (Apache httpd).

An industrial strength Web server is expected to run under high load, when it becomes more susceptible to running out of memory, and so it aims to handle such errors gracefully. Thus, not surprisingly, Apache httpd has extensive checking code for error conditions like NULL returns from malloc throughout its code base. The recovery code for an out-of-memory error generally logs the error and shuts down the server. Nevertheless, AFEX found a malloc failure scenario that is incorrectly handled by Apache and causes it to crash with no information on why. Fig. 3.7 shows the code.

What the Apache developer missed is that strdup itself uses malloc, and can thus incur an out-of-memory error that is propagated up to the Apache code. When this happens, the code on line config.c:579 dereferences the NULL pointer, triggers a segmentation fault, and the server crashes without invoking the recovery code that would log the cause of the error. As a result, operators and technical support will have a hard time understanding what happened.

```
config.c
...
578: ap_module_short_names[m->module_index]
                         = strdup(sym_name);
579: ap_module_short_names[m->module_index][len]
                         = '\0';
```

Figure 3.7 – Missing recovery code in Apache httpd.

This bug illustrates the need for black-box fault injection techniques: an error behavior in a third-party library causes the calling code to fail, because it does not check the return value. Such problems are hard to detect with source code analysis tools, since the fault occurs not in the test system's code but in the third-party library, and such libraries are often closed-source or even obfuscated.

PBFT Our experiment confirmed the Big MAC attack. AFEX showed that by corrupting the MAC in all messages sent by a malicious client, PBFT will perform a view change and crash.

AFEX has also helped us discover an unexpected bug in PBFT[1]. We noticed that despite a malicious client altering the MACs for the replicas, the protocol did not always perform a View Change (specifically, if every retransmission from the malicious client was correct). The PBFT protocol specifies a timer associated to each request received by replicas directly from clients. If a replica receives such a message, it forwards it to the primary and starts a view change timer. If the respective message is not executed before the timer expires, the replica will initiate a view change (this ensures that malicious primaries cannot ignore clients forever). However, in the implementation of PBFT there is a single such timer, rather than one per request. If a message is received by a replica directly from a client, the timer is set. If *any* such message is executed before the timer expires, the timer is reset. Therefore, a malicious primary only has to execute one client request per timer period (5 seconds by default), diminishing PBFT throughput to 0.2 requests / second. If the respective client is also malicious, cooperating with the primary, the primary can ignore all messages from correct clients decreasing the useful throughput of PBFT to 0. This is a particular instance of "slow primaries", which Clement et al. [35] also broadly described; Aardvark avoids this bug by enforcing minimum throughput thresholds for each primary.

The MySQL and Apache experiments in this section show that automatic fault injection can test real systems and find bugs, with minimal human intervention and minimal system-specific knowledge. We now evaluate the efficiency of this automated process.

[1]We later discovered that this bug is similar to the pre-prepare delay attack described by Amir et al. [17]

	Fitness-guided	**Random**	**Exhaustive**
Code coverage	36.14%	35.84%	36.17%
# tests executed	250	250	1,653
# failed tests	74	32	205

Table 3.3 – Coreutils: Efficiency of fitness-guided vs. random exploration for a fixed number (250) of faults sampled from the fault space. For comparison, we also show the results for exhaustive exploration (all 1,653 faults sampled).

3.6.3 Efficiency of Exploration

To evaluate efficiency, we compare fitness-based exploration not only to random search but also to exhaustive search. In order for this to be feasible, we let AFEX test a couple UNIX coreutils, i.e., explore the $\Phi_{coreutils}$ fault space (1,653 faults). This is small enough to provide us an exhaustive-search baseline, yet large enough to show meaningful results.

First, we evaluate how efficiently can AFEX find interesting fault injection scenarios. We let AFEX run for 250 test iterations, in both fitness-guided and random mode, on the ln and mv coreutils. These are utilities that call a large number of library functions. We report in the first two columns of Table 3.3 how many of the tests in the test suite failed due to fault injection. These results show that, given a fixed time budget, AFEX is 2.3× more efficient at finding failed tests (i.e., high-impact fault injections) than random exploration. Exhaustive exploration finds 2.77× more failed tests than fitness-guided search, but takes 6.61× more time (each test takes roughly the same amount of time, and ~90% of the time in each test iteration is taken up by our coverage computation, which is independent of injected fault or workload).

Fitness-guided exploration is efficient at covering error recovery code. Consider the following: running the entire coreutils test suite without fault injection obtains 35.53% code coverage, while running additionally with exhaustive fault exploration (third column of Table 3.3) obtains 36.17% coverage; this leads us to conclude that roughly 0.64% of the code performs recovery. Fitness-guided exploration with 250 iterations (i.e., 15% of the fault space) covers 0.61% additional code, meaning that it covers 95% of the recovery code while sampling only 15% of the fault space.

Code coverage clearly is not a good metric for measuring the quality of reliability testing: even though all three searches achieve similar code coverage, the number of failed tests in the default test suite differs by up to 6×.

In Fig. 3.8 we show the number of failed tests induced by injecting faults found via fitness-guided vs. random exploration. As the number of iterations increases, the difference between the rates of finding high-impact faults increases as well: the fitness-guided algorithm becomes more efficient, as it starts inferring (and taking advantage of) the structure of the fault space. We now analyze this effect further.

Similar results can be seen in the PBFT experiment. Figure 3.9 shows the performance impact on

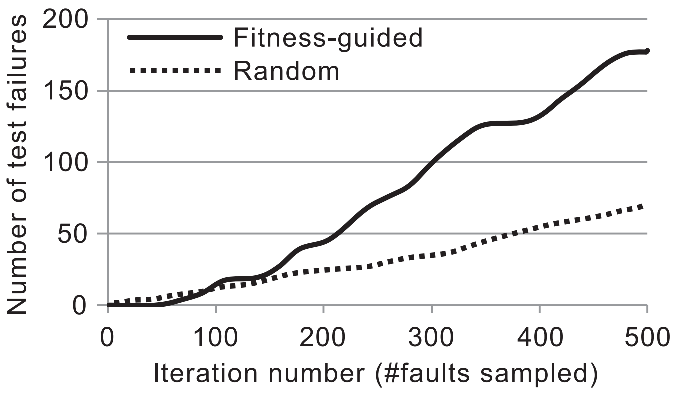

Figure 3.8 – Number of test-failure-inducing fault injections in coreutils for fitness-guided vs. random exploration.

correct clients observed during random fault injection and during our fitness-guided exploration.

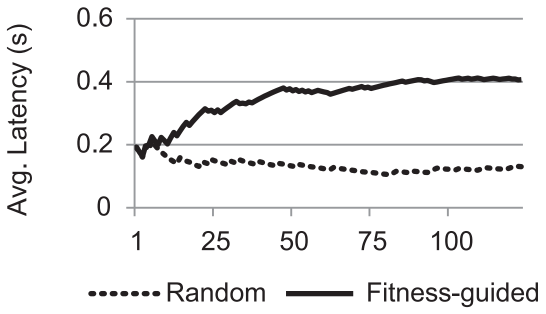

Figure 3.9 – Evolution of average latency of requests from correct clients of the PBFT system, as induced by attacks generated by the fitness-guided exploration of AFEX, versus random exploration, over 125 executed tests

3.6.4 The Impact of the Fault Space Structure

To assess how much AFEX leverages the structure of the fault space, we evaluate its efficiency when one of the fault space dimensions is randomized, i.e., the values along that X_i are shuffled, thus eliminating any structure it had. If the efficiency of AFEX is hurt by such randomization, then it means that the structure along that dimension had been suitably exploited by AFEX. We perform this experiment on Apache httpd with Φ_{Apache}.

The results are summarized in Table 3.4: the randomization of each axis results in a reduction in overall impact. For example, 25% of the faults injected by AFEX with the original structure of Φ_{Apache} led to crashes; randomizing X_{test} causes this number to drop to 22%, randomizing X_{func} makes it drop to 13%, and randomizing X_{call} makes it drop to 17%. The last column, random search, is equivalent to randomizing all three dimensions. This is clear evidence that AFEX takes substantial advantage of whatever fault space structure it can find in each dimension in order to improve its efficiency.

	Original structure	**Rand. X_{test}**	**Rand. X_{func}**	**Rand. X_{call}**	**Random search**
# failed tests	73%	59%	43%	48%	23%
# crashes	25%	22%	13%	17%	2%

Table 3.4 – Efficiency of AFEX in the face of structure loss, when shuffling the values of one dimension of the fault space (Apache httpd). Percentages represent the fraction of injected faults that cause a test in Apache's test suite to fail, respectively crash (thus, 25% crashes means that 25% of all injections led to Apache crashing).

Additionally, in the MySQL experiments of subsection 3.6.2, we inspected the evolution of the sensitivity parameter (described in section 3.3) and the choice of test scenarios, in order to see what structure AFEX infers in the Φ_{MySQL} fault space. The sensitivity of X_{func} converges to 0.1, while that of X_{test} and X_{call} both converge to 0.4 for MySQL. Table 3.4 suggests that Φ_{Apache} is different from Φ_{MySQL}: randomizing X_{func}, which was the least sensitive dimension in the case of MySQL, causes the largest drop in number of crashes, which means that for Φ_{Apache} it is actually the most sensitive dimension.

3.6.5 The Impact of the Result Quality Feedback

Another source of improvement in efficiency is the use of immediate feedback on the quality of a candidate fault relative to those obtained so far. AFEX aims to generate a result set that corresponds as closely as possible to the search target, and it continuously monitors the quality of this result set. One important dimension of this assessment is the degree of redundancy. In this section, we show how AFEX automatically derives redundancy clusters and uses these online, in a feedback loop, to increase its exploration efficiency.

As mentioned in subsection 3.5.4, AFEX uses the Levenshtein edit distance for redundancy detection. In this experiment, we compare the stack traces at the injection points in the Apache tests, cluster them, and tie the outcome into a feedback loop: When evaluating the fitness of a candidate injection scenario, AFEX computes the edit distance between that scenario and all previous tests, and uses this value to weigh the fitness on a linear scale (100% similarity ends up zero-ing the fitness, while 0% similarity leaves the fitness unmodified).

The results are shown in Table 3.5. Even though the use of the feedback loop produces fewer failed tests overall, the search target is more closely reached: fitness-guided exploration with feedback produces about 40% more "unique" failures (i.e., the stack traces at the injection points are distinct) than fitness-guided exploration without feedback, and 75% more "unique" crashes. Of course, the method is not 100% accurate, since injecting a fault with the same call stack at the injection point can still trigger different behavior (e.g., depending on the inputs, the system under test may or may not use a NULL pointer generated by an out-of-memory error), but it still suggests improved efficiency.

	Fitness-guided	**Fitness-guided with feedback**	**Random search**
# failed tests	736	512	238
# unique failures	249	348	190
# unique crashes	4	7	2

Table 3.5 – Number of unique failures/crashes (distinct stack traces at injection point) found by 1,000 tests (Apache).

Having assessed AFEX's properties when operating with no human assistance, we now turn our attention to evaluating the benefits of human-provided system-specific knowledge.

3.6.6 The Impact of System-Specific Knowledge

So far, we used AFEX purely in black-box mode, with no information about the system being tested. We now construct an experiment to evaluate how much benefit AFEX can obtain from knowledge of the tested system and environment.

We choose as search target finding all out-of-memory scenarios that cause the ln and mv coreutils to fail. Based on an exhaustive exploration of $\Phi_{coreutils}$, we know there are 28 such scenarios for these two utilities. Our goal is to count how many samplings of the fault space are required by the AFEX explorer to find these 28 faults.

	Fitness-guided	**Exhaustive**	**Random**
Black-box AFEX	417	1,653	836
Trimmed fault space	213	783	391
Trim + Env. model	103	783	391

Table 3.6 – Number of samples (injection tests) needed to find all 28 malloc faults in $\Phi_{coreutils}$ that cause ln and mv to fail, for various levels of system-specific knowledge.

Table 3.6 shows the results for three different levels of system-specific knowledge. First, we run AFEX in its default black-box mode; this constitutes the baseline. Then we trim the fault space by reducing X_{func} to contain only the 9 libc functions that we know these two coreutils call—this reduces the search space. Next, we also add knowledge about the environment in the form of a statistical environment model, which specifies that malloc has a relative probability of failing of 40%, all file-related operations (fopen, read, etc.) have a combined weight of 50%, and opendir, chdir a combined weight of 10%. We use this model to weigh the measured impact of each test according to how likely it is to have occurred in the modeled environment.

The results show that trimming the fault space improves AFEX's efficiency by almost 2×, and adding the environment model further doubles efficiency—AFEX is able to reach the search target more than 4× faster than without this knowledge. Furthermore, compared to uninformed random search, leveraging this domain-specific knowledge helps AFEX be more than 8× faster, and compared to uninformed exhaustive search more than 16× faster.

3.6.7 Efficiency in Different Development Stages

So far, we have evaluated AFEX only on mature software; we now look at whether AFEX's effectiveness is affected by the maturity of the code base or not. For this, we evaluate AFEX on the MongoDB DBMS, looking at two different stages of development that are roughly 3 years apart: version 0.8 (pre-production) and version 2.0 (industrial strength production release). We ask AFEX to find faults that cause the tests in MongoDB's test suite to fail, and we expose both versions to identical setup and workloads. We let AFEX sample the fault space 250 times and compare its efficiency to 250 random samplings. The results are shown in Figure 3.10.

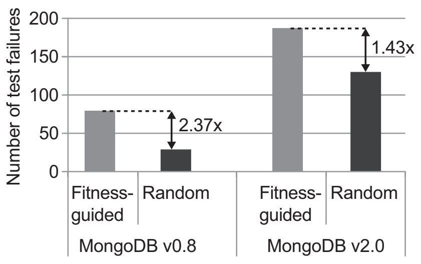

Figure 3.10 – Changes in AFEX efficiency from pre-production MongoDB to industrial strength MongoDB.

For early versions of the software, AFEX is more efficient at discovering high-impact faults: compared to random search, AFEX finds 2.37× more faults that cause test failures; this efficiency drops in the industrial strength version to 1.43×. What may seem at first surprising is that AFEX causes more failures in v2.0 than in v0.8—this is due to increased complexity of the software and heavier interaction with the environment, which offers more opportunities for failure. Ironically, AFEX found an injection scenario that crashes v2.0, but did not find any way to crash v0.8. More features appear to indeed come at the cost of reliability.

3.6.8 Scalability of our Fault Injection Framework

We have run AFEX on up to 14 nodes in Amazon EC2 [16], and verified that the number of tests performed scales linearly, with virtually no overhead. This is not surprising. We believe AFEX can scale much further, due to its embarrassing parallelism. In isolation, the AFEX explorer can generate 8,500 tests per second on a Xeon E5405 processor at 2GHz, which suggests that it could easily keep a cluster of several thousand node managers 100% busy.

4 White-box Search for Trojan Messages

4.1 Intuition

We define Trojan messages as those that are accepted by correct server nodes in a distributed system but cannot be generated by any correct client node, as illustrated in Figure 4.1. By "correct" we mean nodes that execute the unaltered implementation of the distributed system, in contrast to incorrect nodes, which are nodes that have encountered a fault (e.g., memory bit flip), or are controlled by malicious users. Thus, in our definition, "correct" does not necessarily imply that nodes follow the high-level specification of the protocol, or do what the developers intended. Messages that respect our definition of Trojan are useless in the system and potentially dangerous, as they are a means of failure propagation.

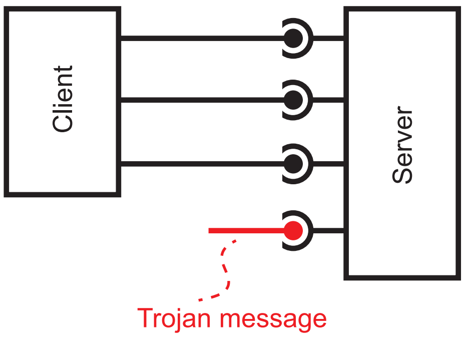

Figure 4.1 – Trojan messages are messages that are accepted by a server but cannot be generated by any correct client.

In this thesis we propose Achilles, a technique designed to identify Trojan messages in a distributed system. For simplicity, we consider in our description of Achilles only client-server systems where the client generates requests and the server replies; it is straightforward to generalize the approach to peer-to-peer or other types of distributed systems.

Conceptually, there are three steps to find Trojan messages: (1) find what clients can send, (2) find what servers accept as valid, and (3) compute the difference between the two. This is what

Achilles does, albeit with a few tricks to make the approach practical (subsection 4.2.2). In a first phase, it computes a predicate that defines all messages that can be generated by correct clients. We call this the *client predicate* P_C. The set of messages that can be generated by correct clients is $\mathscr{C} = \{msg \mid P_C(msg)\}$. In the second phase, Achilles computes a predicate that defines all messages that are accepted by the server, the *server predicate* P_S. The set of messages that are accepted as correct by correct servers is $\mathscr{S} = \{msg \mid P_S(msg)\}$. Then, the set of Trojan messages is defined as $\mathscr{T} = \mathscr{S} \setminus \mathscr{C}$, which can be expressed as $\mathscr{T} = \{msg \mid P_S(msg) \wedge \neg P_C(msg)\}$.

In essence, Achilles extracts the grammar of the communication protocol, as implemented in both the client and the server. It then operates on the two grammars, trying to determine the difference between the server's implementation of the grammar and the client's. As we describe later (section 4.2), Achilles uses symbolic execution in order to analyze the implementation of clients and servers; it represents the extracted grammars using symbolic constraints.

4.1.1 Working Example

```
#define DATASIZE 100
peers = initializePeers();
data = new int[DATASIZE];
while(TRUE) {
  msg = receiveMessage();
  if (!isInSet(msg.sender, peers))
    continue;
  if (!isValidCRC(msg, msg.CRC))
    continue;
  switch (msg.request) {
    case READ:
      if (msg.address >= DATASIZE)
        continue;
      //Security vulnerability: forgot to check
      //           address < 0
      sendMessage(msg.sender, REPLY,
                  data[msg.address]);
      continue;
    case WRITE:
      if (msg.address >= DATASIZE)
        continue;
      if (msg.address < 0)
        continue;
      data[msg.address] =
                  msg.value;
      sendMessage(msg.sender, ACK);
      continue;
    default:
      continue};
  }
}
```

Figure 4.2 – A simple server that handles requests from clients. The server accepts Trojan messages, as it does not correctly validate the address field of read requests.

To illustrate the idea underlying our approach, consider the sample server implementation presented in Figure 4.2.

This is a simplified example, in which a server handles read or write requests from clients. The server first initializes its internal data structures (lines 2-3). Next, the server enters an event loop in which it handles client requests. The program receives a message and stores it in the local variable *msg* (line 5). Then, the server checks that the message sender is in a pre-configured group of known peers (line 6) and if the CRC error-detection code is correct.

If the message passes the general validation tests, the server reads the *request* field to get the message type—either a *READ* or a *WRITE* request. If the request is not one of the two, the message is discarded (line 29). For either of the request types, the server validates the address field and handles the respective request.

The client shown in Figure 4.3 follows the same protocol as the server. It reads user requests from standard input, validates the data (lines 5-7) and computes corresponding requests for the server.

```
1  #define DATASIZE 100
2  peerID = getPeerID();
3  operationType = readFromKeyboard();
4  address = readFromKeyboard();
5  if (address >= DATASIZE)
6    exit(1);
7  if (address < 0)
8    exit(1);
9  //Client only sends addresses in [0,100)
10 if (operationType == READ) {
11   msg = new ReadMessage();
12   msg.sender = me;
13   msg.request = READ;
14   msg.address = address;
15   msg.CRC = computeCRC(msg);
16   sendMessage(server, msg);
17 }
18 if (operationType == WRITE) {
19   value = readFromKeyboard();
20   msg = new WriteMessage();
21   msg.sender = me;
22   msg.request = WRITE;
23   msg.address = address;
24   msg.value = value;
25   msg.CRC = computeCRC(msg);
26   sendMessage(server, msg);
27 }
```

Figure 4.3 – A simple client that generates messages. Correct clients validate the address field, therefore cannot expose the bug in the server.

The sample system has a Trojan message. The server validates that the requested address is below the maximum *DATASIZE*, however, for *READ* requests it does not ensure that the address is greater than zero. The client, however, validates the input from the user before contacting the server. Therefore, no correct client can generate *READ* messages with negative offsets, although they are accepted by the server. Thus, any *READ* message with a negative address is a Trojan message.

In this example, the Trojan message can lead to a potential privacy leak, as attackers can read from negative offsets in the data array and discover, for instance, the list of peers that communicate with the server.

4.2 Design

At a high level, Achilles works as follows. In a first phase, it obtains the client predicate P_C. Then, it pre-processes P_C to eliminate redundancy and to pre-compute structure information for the next phase (details follow in item 4.2.2). In the second phase, Achilles computes the server predicate P_S, as well as incrementally discovering Trojan messages. As an optimization, our implementation does not compute the entire formula of P_S before computing Trojan messages, but rather searches for Trojan messages incrementally, as it builds P_S.

In this section, we first give an overview of how Achilles uses symbolic execution in order to extract the grammar from both the client and the server (subsection 4.2.1). We then describe how Achilles finds Trojan messages efficiently using off-the-shelf constraint solvers (subsection 4.2.2). We present optimizations that enable Achilles to handle large client predicates (item 4.2.2). Finally, we describe how Achilles handles local state in the client and in the server (subsection 4.2.3).

4.2.1 Symbolic Execution and Message Grammars

Achilles uses symbolic execution [61] to extract the grammars of messages that can be sent/received by an implementation of a distributed system. Symbolic execution is a technique that systematically explores execution paths in a system. Rather than exercising a system with concrete inputs, a symbolic execution engine provides "symbolic" data, which can conceptually take any value. The symbolic execution engine then executes the system under test and interprets expressions symbolically. At each branching point that depends on symbolic data, the symbolic execution engine invokes a Satisfiability Modulo Theories (SMT) solver to assess the feasibility of each branch, based on the current state of the system. When a branch is deemed feasible, the execution engine follows the respective branch and also updates the symbolic data, keeping track of the constraints that need to be satisfied such that the branch is feasible. If both paths are feasible, the execution engine forks the exploration and follows both branches.

At every step of the symbolic execution, the engine keeps track of several possible execution states of the system, each corresponding to a possible execution path. Figure 4.4 shows a sample symbolic execution of a small piece of code. A symbolic execution state consists of a *symbolic store*, which maps variables from the system under test to expressions on symbolic inputs, and

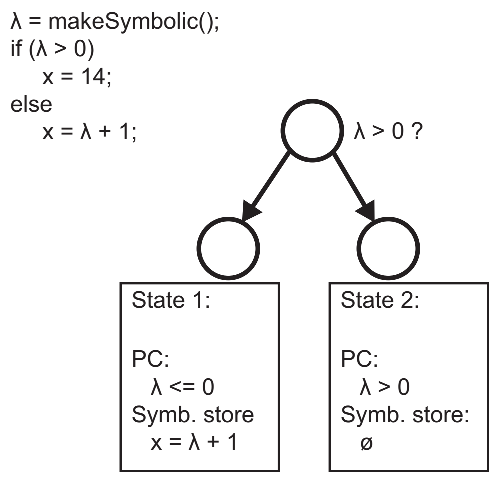

Figure 4.4 – An example of symbolic execution for a small piece of code.

the set of *path constraints*, which represent the conditions on symbolic input that must hold for the respective execution path to be feasible.

In Achilles, we represent message grammars as sets of constraints obtained during symbolic execution. The server predicate P_S is expressed by the set of path constraints that correspond to execution paths that handle accepted messages. The client predicate P_C is expressed by the symbolic messages that the client sends over the network on different execution paths, along with their respective path constraints.

Client Predicate. In a distributed system, clients receive "local" inputs, parse the inputs, generate corresponding messages conforming to the specification of the protocol (or, more precisely, to the client's implementation of that specification), and finally send the messages to the server. The client predicate is a representation of all valid messages that can be *sent*.

In order to obtain the set of all possible messages that can be generated by a client in a distributed system, we conceptually need to provide the client with all possible "local" inputs and then capture the messages it sends over the network. We start the client in a symbolic environment such that any local system call that the client emits in order to read input is intercepted, and the result is replaced with symbolic data. The symbolic execution engine keeps track of how the symbolic data flows through the client and, when the client attempts to send a message over the network, Achilles captures and stores the contents of the message and the corresponding path constraints. The message payload may contain concrete data, but also expressions on symbolic data.

Thus, the predicate P_C is the disjunction of predicates $path_{C_1} \vee \cdots \vee path_{C_n}$, where each $path_{C_i}$ is a conjunction of path constraints and symbolic expressions for an execution path that sends a message. We call each such predicate $path_{C_i}$ a *client path predicate*.

```
∃ symb_PeerID, symb_Address, symb_Value :

message.sender = symb_PeerID
⋀ message.request = READ
⋀ message.address = symb_Address
⋀ message.CRC = CRC(message)
⋀ symb_Address < 100
⋀ symb_Address >= 0
⋁
message.sender = symb_PeerID
⋀ message.request = WRITE
⋀ message.address = symb_Address
⋀ message.value = symb_Value
⋀ message.CRC = CRC(message)
⋀ symb_Address < 100
⋀ symb_Address >= 0
```

Figure 4.5 – The client predicate $P_C(message)$ of the client in Figure 4.3, as discovered by the symbolic execution phase of Achilles.

For the example client in Figure 4.3, Achilles discovers the predicate presented in Figure 4.5. Names that begin with *symb_* indicate symbolic data automatically inserted by Achilles. There are two main execution paths in the client that lead to messages being sent. One path corresponds to a *READ* request, while the other corresponds to a *WRITE* request. In both cases, the *message.request* header contains concrete data. This is because there is no data flow dependency between the *message.request* header and any symbolic input; the values for the two paths only differ because of control flow dependencies.

All other headers except *message.request* contain symbolic data. The *message.value* and *message.sender* headers, for example, contain the respective unconstrained symbolic inputs. The *CRC* header contains an expression on all other symbolic inputs (in Figure 4.5, the expression is summarized by the CRC function, but in real deployments, the expression contains the full chain of operations that transform the symbolic inputs). Finally, the *address* header contains constrained symbolic data. There are no operations directly performed on *symb_Address*; however, the execution path that leads to the message imposes constraints on the possible values: *symb_Address* needs to have values between $[0, 100)$ in order for the message to be sent. This is reflected in the path constraints that are captured and stored by Achilles.

In distributed systems, messages usually have clearly differentiated fields. Developers might be interested to check only a subset of those fields for Trojan messages (e.g., only check the contents of the *address* field). We support this selective approach in Achilles (subsection 4.4.2) and also refer to techniques that automatically avoid complex authentication or encryption functions (chapter 2). Since the grammar extraction phase of Achilles is based on vanilla symbolic execution, Achilles can benefit from a variety of approaches that enhance symbolic execution.

Server Predicate. In a distributed system, the server receives messages from a client, parses the message, and executes the operation requested by the client. The server should discard messages that do not follow the specification, and only execute operations corresponding to valid messages. The server predicate P_S is a representation of all valid messages that can be *received*.

Each message received by the server triggers the execution of code to handle it. For simplicity, we refer to the sequence of executed instructions as the *execution path* triggered by the message. The execution path starts after the return of the *receive* instruction and ends when the message processing is complete—either the server exits, or it listens for new events.

After receiving a message, the server decodes it and parses its fields, checking if the message conforms to the specification of the protocol (or, more precisely, to the server node's implementation of that specification). In order to infer the set of valid messages, Achilles first classifies execution paths in the system under test as either *accepting* or *rejecting*. *Accepting* execution paths are those that are triggered by messages that pass the initial parsing stages of the system under test and cause the server to perform an action. Conversely, *rejecting* execution paths are those that are triggered by messages that are rejected by the server. Achilles automates the classification of paths, using some simple observations on the behavior of the server. Human operators of Achilles can also manually place tags in the code in order to speed up the analysis (details follow in subsection 4.4.2).

In order to obtain the server predicate, Achilles executes the server node symbolically, feeding it an unconstrained symbolic message. The server analyzes the message and branches into different execution paths, one for each type of message in the protocol specification. Symbolic execution enumerates all these paths, keeping track of the constraints that make each path feasible.

We define the server predicate P_S as the disjunction of all path constraints corresponding to *accepting* execution paths. Thus, P_S represents all possible messages that can trigger *accepting* execution paths in the server.

Similarly to P_C, the server predicate P_S is obtained using vanilla symbolic execution. Therefore, Achilles can benefit from any optimization that enhances the performance of symbolic execution engines.

In the example in Figure 4.2, we define *rejecting* execution paths as those that do not send a reply to the client (reach lines 7, 9, 13, 21, 23 or 29 of the server code), and *accepting* execution paths as those paths that send a reply (reach line 17 or 26). The server predicate, as discovered by our technique, is presented in Figure 4.6.

4.2.2 Systematic Search for Trojan Messages

Trojan messages are, by definition, all messages in $\mathscr{S} \setminus \mathscr{C}$. This is equivalent to $\mathscr{T} = \{msg \mid P_S(msg) \land \neg P_C(msg)\}$. In Achilles, Trojan messages are computed incrementally, as it builds up

```
isInSet(message.sender, peers) = TRUE
⋀ isValidCRC(message, message.CRC) = TRUE
⋀ message.request = READ
⋀ message.address < 100
⋁
isInSet(message.sender, peers) = TRUE
⋀ isValidCRC(message, message.CRC) = TRUE
⋀ message.request = WRITE
⋀ message.address < 100
⋀ message.address ≥ 0
```

Figure 4.6 – The server predicate $P_S(message)$ of the server in Figure 4.2, as discovered by the symbolic execution phase of Achilles.

the formula of P_S. This point is essential, as clients are usually less complex than servers. The extraction of the client predicate P_C is easier than the extraction of the server predicate P_S.

For every explored execution path in the server, Achilles keeps track of a list of client path predicates that contain messages that can trigger the respective path, as shown in Figure 4.7. At every branch point encountered during the symbolic execution of the server, Achilles checks which of the client messages can still trigger each path, and whether the paths can be triggered by any Trojan messages. As soon as an execution path cannot be triggered by any Trojan messages, it is dropped from the exploration. Therefore, by construction, any execution path in the server that reaches an *accepting* marker has Trojan messages. For such execution paths, Achilles outputs a symbolic expression and a concrete example of the Trojan message.

Note that Trojan messages might be exclusive on execution paths (e.g., the accepting path on the right in Figure 4.7), or they might be bundled with other, non-Trojan messages (e.g., the accepting path on the left). The latter case means that classic symbolic execution by itself does not alleviate the problem of finding Trojan messages by much—it weeds out execution paths with dropped messages from those with accepted messages, but Trojan messages may be anywhere among the execution paths that handle accepted messages. In the example in Figure 4.2, the Trojan messages (those with $msg.address < 0$) are on the same execution path with valid messages (those that satisfy $msg.address >= 0$ and $msg.address < DATASIZE$); symbolic execution by itself would not point out the presence of a potential bug. This is also the case of the wildcard bug found by Achilles in FSP (subsection 4.5.3) and the vulnerability found by Achilles in PBFT (subsection 4.5.3).

We optimize the discovery of Trojan messages by adapting Achilles to the particularities of symbolic execution, as described in the following subsections.

Constraint Solving. Predicates P_C and P_S are disjunctions of path predicates. Each path predicate $path \in P$ is a conjunction of constraints and assignments. A predicate defines a class of

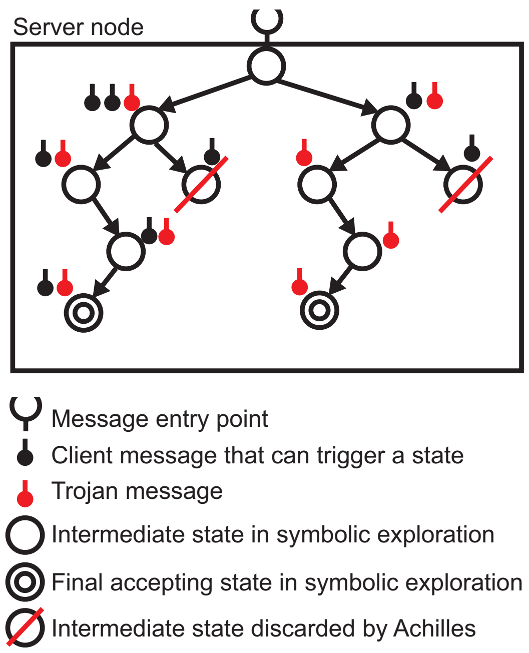
Server node
Message entry point
Client message that can trigger a state
Trojan message
Intermediate state in symbolic exploration
Final accepting state in symbolic exploration
Intermediate state discarded by Achilles

Figure 4.7 – Symbolic execution of a server node in Achilles. Using the predicate collected from the client, Achilles restricts the server exploration to paths that can be triggered by Trojan messages.

messages generated by a client, or accepted by a server. In order to compare two predicates $path_1$ and $path_2$, Achilles must combine the two in a single formula and then call an SMT solver in order to check satisfiability.

In order to combine a client and a server path predicate, Achilles adds a conjunction between their respective path constraints, and also adds an equality constraint between the value of the message generated in the client and the value of the message received in the server. In essence, the combination between a server and a client path predicate, $path_S$ and $path_C$, represents messages msg such that:

- msg_S satisfies the server predicate
- msg_C satisfies the client predicate
- $msg_S = msg_C = msg$

SMT solvers, such as STP [43] or Z3 [37], verify the satisfiability of a set of constraints and expressions on symbolic variables. Furthermore, SMT solvers can compute a concrete assignment of the symbolic variables that satisfies a set of expressions and constraints. For example, an SMT solver can determine that the set of constraints $\lambda > 0 \wedge \lambda < -5$ is unsatisfiable. However, $\lambda > 0 \wedge \lambda < 5$ is satisfiable and $\lambda = 3$ satisfies the constraints.

In terms of symbolic expressions, the definition of the set of Trojan messages $\mathscr{T}$ can be written as the set of messages that satisfy the server predicate, but not the client predicate:

$$\mathscr{T} = \{msg \mid P_S(msg) \wedge \neg P_C(msg)\}$$

P_C contains an existential quantifier; negating it produces a universal quantifier. This makes solving the expression above difficult using current SMT solvers. Z3 has limited support for quantifiers, using heuristics and patterns to eliminate quantification. Achilles calls Z3, attempting to solve the predicate and check for the presence of Trojan messages. However, the heuristics may fail; in this case, Z3 cannot answer whether an expression is satisfiable or not. We discuss how we under-approximate the negation of client path predicates and eliminate the universal quantifier in the following subsection.

Negating Path Predicates. Recall that a client path predicate $path_C \in P_C$ represents all possible messages that can be generated on a particular execution path. Let $negate(path_C)$ be an operator that generates a predicate representing all possible messages that *cannot* be generated on that path.

Figure 4.8 shows the expression of READ messages generated by the sample client in Figure 4.3. There are three symbolic variables in the message, $\lambda_{PeerID}, \lambda_{Address}$ and λ_{CRC}, and one concrete variable, containing the value *READ*. λ_{PeerID} is the unconstrained value of the *symb_PeerID* input. $\lambda_{Address}$ is the value of the *symb_Address* input, but it is subjected to the restrictions in the path constraint. Finally, λ_{CRC} is a more complex operation on $\lambda_{PeerID}, \lambda_{Address}$ and *READ*.

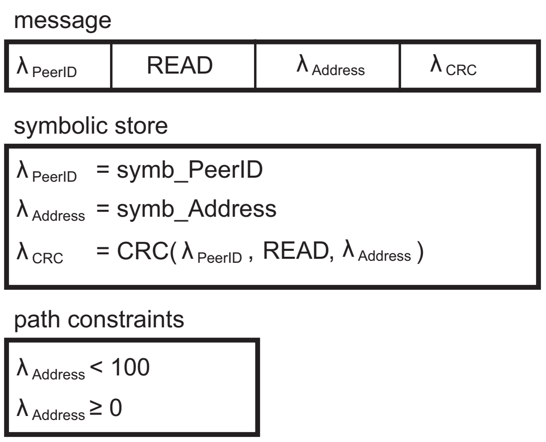
message
λ_{PeerID}
READ
$\lambda_{Address}$
λ_{CRC}
symbolic store
λ_{PeerID} = symb_PeerID
$\lambda_{Address}$ = symb_Address
λ_{CRC} = CRC(λ_{PeerID} , READ, $\lambda_{Address}$)
path constraints
$\lambda_{Address} < 100$
$\lambda_{Address} \geq 0$

Figure 4.8 – The expression of READ messages generated by the sample client.

Achilles under-approximates the true *negate* operator as follows. Achilles' custom operator computes the negation of a path predicate as a disjunction of the negation of each individual value in the message buffer. In order to negate a value, there are two cases:

1. If the value is a concrete value C, then replace the respective value with a new symbolic value λ and add the constraint $\lambda \neq C$.
2. If the value is an expression on symbolic variables, then negate the set of constraints that influence the respective variables. If there are no constraints available, then abandon the negation of the current value.

For the example in Figure 4.8, the negation of the path predicate is a representation of the set of message where:

- the *request* field is different from *READ* or
- the *address* field is smaller than 0 or greater than 100 or
- the *CRC* field is the CRC function of a message where the *address* is smaller than 0 or greater than 100 or its *request* is different from READ.

In constraint form, this is written as:

$$
\begin{aligned}
&(message.request = \lambda_{req}) \wedge \lambda_{req} \neq READ \\
&\vee(message.address = \lambda_{Address}) \\
&\qquad\qquad \wedge(\lambda_{Address} \geq 100 \vee \lambda_{Address} < 0) \\
&\vee(message.CRC = CRC(READ, \lambda_{Address}, \lambda_{PeerID})) \\
&\qquad\qquad \wedge(\lambda_{Address} \geq 100 \vee \lambda_{Address} < 0)
\end{aligned}
$$

We discuss the soundness and completeness of the custom *negate* operator in a dedicated section (section 4.3).

Handling Large Client Predicates. The server analysis phase of Achilles is a search problem—find all possible execution paths that accept Trojan messages. Symbolic execution enumerates the possible execution paths in the server, with the added precondition that there exists at least one message that satisfies the path constraints in the server and the conjunction of all negated client path predicates. The negation operator allows checking for Trojan messages with a single query to the constraint solver; however, this query can be complex.

Achilles needs a strategy to efficiently handle the thousands of expressions that can appear in a client predicate. The key idea is to keep track of which path predicates can trigger an execution path explored in the server.

The P_S extraction phase of Achilles incrementally adds constraints to each $path_S \in P_S$, as it explores execution paths. Rather than just checking whether there are Trojan messages on the current path (if $path_S \wedge negate(path_{C_1}) \wedge \cdots \wedge negate(path_{C_n})$ is satisfiable), Achilles also checks whether $path_S \wedge path_{C_i}$ is satisfiable, for each $path_{C_i} \in P_C$. Whenever the latter is not satisfiable, Achilles drops $negate(path_{C_i})$ from the solver query that checks for Trojan messages—if $path_S \wedge path_{C_i}$ is false, then $path_S \wedge negate(path_{C_i})$ is implicitly true (this is because $path_S$ and $path_{C_i}$ are both satisfiable, by construction). Once dropped, $path_{C_i}$ will never be checked again on the respective execution path; $path_{C_i}$ does not change, while $path_S$ only becomes more constrained, meaning that the conjunction can never become feasible in the future. As the symbolic execution engine explores longer execution paths, the number of client path predicates that can trigger the path decreases, and the size of the SMT query that checks for Trojan messages becomes smaller (we evaluate the effect of this optimization in our experiments subsection 4.5.4).

In order to further optimize the search for Trojan messages, Achilles also exploits the process through which path predicates are generated. The intuition behind this optimization is that messages generated by the client are similar to each other. Two client path predicates can represent the same values for a certain field in a message, e.g., $path'_C = (msg.x = 1) \wedge (msg.y = 2)$ and $path''_C = (msg.x = 1) \wedge (msg.y = 7)$ have the same values for field x. In this case, if $path'_C$ is dropped due to a new server constraint on field x, then $path''_C$ can also be dropped without further checking.

Achilles pre-computes a data structure that stores information about relations between client path predicates. The data structure can be seen as a three-dimensional matrix *differentFrom*, where $differentFrom[i][j][field] = TRUE$ means that there exists at least one message $msg_i \in path_{C_i}$ such that there is no message $msg_j \in path_{C_j}$ with $msg_i.field = msg_j.field$. The *differentFrom* data structure allows Achilles to discard several client path predicates at a time. Intuitively, when the symbolic execution of the server in Figure 4.2 encounters the condition `if (operationType == READ)` it can drop *all* client path predicates that generate READ messages without invoking the SMT solver each time.

The data structure is computed by applying the *negate* operator for each field of messages, between each pair of client path predicates. The data structure is only computed for message fields that are independent, meaning that they do not appear in constraints with other fields. As can be seen in our evaluation (section 4.5), this pre-computation is fast; moreover, it is trivially parallelizable.

While symbolically executing a server node, Achilles uses the information from *differentFrom* at every point where the exploration forks. If a branching point depends on field a from the received message (and not on any other fields), and the new constraints of the branching point make $path_{C_i} \wedge path_S$ no longer satisfiable, then Achilles can drop $path_{C_i}$ from the current state, but also all path predicates $path_{C_j}$ such that $differentFrom[j][i][a] = FALSE$. In other words, if $path_{C_i}$ no longer holds, due to the additional checks on field a, then all path predicates $path_{C_j}$ can be dropped, because they cannot have any additional values for field a.

For the sample client predicates in Figure 4.5, $differentFrom[1][2][request] = TRUE$, because client predicate 1 is satisfied by an assignment $message.request = READ$, while predicate 2 is not. However, $differentFrom[1][2][address] = FALSE$, because the $message.address$ field of the

READ request is not satisfied by any other values than those that also satisfy the address of the *WRITE* request.

4.2.3 Support for Local State in Distributed Systems

Distributed systems often keep local state across several rounds of messages and change their behavior based on the contents of the local state. For example, Paxos [68] uses three phases to achieve consensus. In each phase, Paxos nodes accept different types of messages—the predicate P_S depends on the local state of the nodes. Thus, Achilles needs a mechanism to control the local state of the analyzed nodes. Achilles provides several alternatives to support local state.

Concrete Local State. The analysis phase of Achilles can be started at any point in the execution of a distributed system. Achilles is implemented on top of S2E [32], a symbolic execution platform that can run whole systems: operating systems, libraries and user applications. This enables the possibility of running the distributed system concretely up to some point, implicitly building up concrete local state.

This mode of operation is useful when developers are interested in the behavior of their implementation in a specific scenario. For example, in the case of basic Paxos, developers might be interested in what happens when a Paxos Acceptor has just entered the second phase, with proposed value 7. It should only validate *Accept* messages for value 7—any other message is a Trojan message.

Another example where Concrete Local State is useful is the Amazon S3 bug [1], described in chapter 1. The Trojan message that caused the outage incorrectly reported a high failure rate in the system. The message was not necessarily Trojan in all possible scenarios—there may be scenarios where there are indeed a lot of failures in the system. However, the message was Trojan in the concrete scenario in which it occurred. By building concrete state in a deployment with few failures, Achilles could discover that no correct client node can report high failure rates, yet the servers accept such messages.

Constructed Symbolic Local State. As a generalization of the Concrete Local State mode, Achilles is able to pass symbolic messages between nodes of a distributed system, such that the nodes build up symbolic local state. The Constructed Symbolic Local State mode can capture entire sets of possible concrete execution scenarios. However, due to the additional symbolic variables and constraints, the Constructed Symbolic Local State mode can encounter difficulties with complex data structures, like any symbolic execution tool.

Consider again the example of Paxos. In order to be confident that there are no Trojan messages in the second phase of the protocol, developers need to re-use Achilles for every combination of concrete values in the local state. Using Concrete Local State, this entails re-running Paxos with different proposed values (1, 2, etc.) and then applying Achilles for each scenario. Alternatively, with Constructed Symbolic Local State, developers can run Paxos once, with a symbolic proposed value. Paxos nodes will store the symbolic value in their local state. Thus, Achilles can be applied a single time.

Over-approximate Symbolic Local State. The third mode of operation allows distributed system developers to place annotations that describe the local state. Developers can insert annotations in the code of the distributed system nodes, or insert them directly in the binary at runtime, using S2E plugins that monitor execution. Essentially, developers can manually define memory as symbolic local state and specify constraints on the stored values.

In the Paxos example, developers can annotate the functions that deal with local state in order to directly return symbolic values. This approach is useful when the implementation uses complex data structures to store local state, as it makes it possible for developers to reduce the burden on SMT solvers.

4.3 Soundness and Completeness

The purpose of Achilles is to discover Trojan messages, i.e., messages that are accepted by a server but cannot be generated by any correct client. Achilles can have both false positives and false negatives and is therefore not guaranteed to be sound or complete. False positives, however, can be kept under control by the operator of Achilles, as described below. We believe this makes Achilles a useful testing tool for distributed system developers, who can incorporate the messages discovered by Achilles in fault injection testing.

4.3.1 False Positives

We say that Achilles has false positives when there exists a message *m* that is falsely identified as Trojan. This can happen either when *m* is rejected by the server, or when *m* can be generated by a correct client.

Achilles relies on symbolic execution in order to extract predicates. Symbolic execution incrementally builds up an expression of an entire program by systematically exploring feasible execution paths. As long as symbolic execution does not completely explore all paths, the expression under-approximates the program. When the client is under-approximated, it might happen that a message *m* can only be generated on the execution paths that were not yet explored — this leads to false positives in Achilles.

In our experiments in section 4.5, we did not encounter any false positives of Achilles. We bounded the maximum messages size in both the client and server, allowing symbolic execution to complete. We also restricted symbolic execution to only some fields of a message. Essentially, we restrict symbolic execution to only a subset of the input space, as has also been proposed by Person et al. [88].

Achilles generates concrete values for the Trojan messages it discovers, as well as symbolic expressions, allowing testers to inject the concrete messages in a real deployment (such as in live fire drills) and check the effect of the messages, weeding out harmless messages. While this concretization does not disambiguate between Trojan and non-Trojan messages, it helps discover any adverse effects of the suspected Trojan messages.

It is worth pointing out that Achilles does not also encounter false positives due to inaccuracy in Achilles' implementation of the *negate* operator. The implementation is strictly an under-approximation of the ideal *negate* operator. For each negated expression generated by the *negate*, we use the SMT solver to check if there is any common solution between the original expression and its negation; whenever there is such a solution, we discard the negated expression, eliminating any false positives that could have occurred.

4.3.2 False Negatives

We say that Achilles has false negatives when there exists a message m that is not reported as a Trojan message, although it is actually accepted by the server and cannot be generated by any correct client.

False negatives can occur when the symbolic execution of the server does not exhaust all execution paths. If a message m cannot be generated by any correct client, but is accepted by the server on an execution path not explored by Achilles, then m is an undiscovered Trojan message.

False negatives can also occur due to the fact that our implementation of the *negate* operator under-approximates the real negate. As described in the Design section (subsection 4.2.2), we use two alternative approaches to compute the negation: one approach relies on quantifier support in the Z3 SMT solver, while the other approach relies on tweaked constraints that are solved using the STP SMT solver. Z3 uses heuristics to eliminate quantifiers, and may fail to determine satisfiability. When this happens, Achilles falls back to customized constraints sent to STP. Achilles' customized *negate* is under-approximate; for example *negate*$((\lambda = 2*x) \wedge (x > 0))$ produces $((\lambda = 2*x) \wedge (x \leq 0))$, even though -1 and 1 are also values for λ that do not satisfy the initial expression. In our evaluation (section 4.5), though, we found that although under-approximate, Achilles' customized implementation of *negate* can find Trojan messages in real distributed systems.

4.4 A Prototype Implementation

In this section, we describe some implementation details of our Achilles prototype. We built Achilles on top of the S2E symbolic execution platform [32]. S2E can run whole systems; this means that S2E allows running the implementation of distributed system nodes in their real environment, enabling Achilles to also discover Trojan messages that are due to third party system components, such as libraries. S2E is also highly extensible through a rich plugin API, simplifying the development of Achilles.

4.4.1 Intercepting System Calls

Achilles uses the *LD_PRELOAD* mechanism in Linux to intercept calls to the standard *libc* library. Thus, Achilles can insert symbolic data automatically by overwriting calls that read inputs, replacing concrete values with symbolic data.

Achilles also intercepts calls to network operations, in order to automatically mark server execution paths as *accepting* or *rejecting*. By default, we consider that any execution path that sends a reply to the user is *accepting*, while any execution path that waits for new messages is *rejecting*. The assumption behind this is that servers have a single event loop to listen for messages; restarting the loop means that the previous message was processed. This can be trivially extended to handle other common error signaling mechanisms (e.g., 4xx status codes in HTTP). Of course, human operators can also provide additional domain knowledge by manually placing *accept* or *reject* markers in the system under test. For example, if the target protocol uses encryption, it makes sense for the human operator to place an *accept* marker before the encryption of the reply.

Achilles uses system call interception in order to implement the Constructed Symbolic Local State mode. In order to avoid unnecessary forking in device drivers, Achilles intercepts network operations that send symbolic data and reroutes them through custom channels (currently, nodes are deployed within the same S2E instance and messages are rerouted through shared memory).

4.4.2 Analysis API for Distributed System Developers

Developers can use annotations to speed up Achilles. The annotations can be inserted in the code of the distributed system, or can be injected in the system at runtime, using S2E's plugin support.

- *mark_accept* is a server-side annotation that marks an execution path as *accepting*, triggering the check for any Trojan messages.
- *mark_reject* is a server-side annotation that marks an execution path as *rejecting*.
- *function_start* and *function_end* are used in order to over-approximate the behavior of a function. The operator can add constraints between the two markers, in order to impose constraints on return values.
- *drop_path* is used in conjunction with the *function_* markers in order to impose constraints on the return values.

- *return_symbolic* is used in conjunction with the *function_* markers in order to set the return value of a function.

- *make_symbolic* makes a system variable symbolic; it can be used in order to mark local state as symbolic.

For example, a developer can over-approximate the *getPeerID*() function from the client code in Figure 4.3, as shown in Figure 4.9. The over-approximation bypasses the code of the function completely and returns a new symbolic value constrained to the interval $[0,10]$. In our evaluation (section 4.5) we used annotations to bypass cryptography and authentication code and speed up predicate extraction.

```
int getPeerID() {
  function_start();
  int toRet = makeSymbolic();
  if ( toRet < 0 ) drop_path();
  if ( toRet > 10 ) drop_path();
  return_symbolic(toRet);
  function_end();
  ...  //actual code of getPeerID
```

Figure 4.9 – An example of using Achilles annotations in order to over-approximate a function to return values $[0,10]$

Achilles supports a mask to “hide” certain message fields from the analysis. The symbolic execution of the server still branches on the hidden values; however, Achilles does not check for Trojan messages involving those fields. The mask increases the signal-to-noise ratio of Achilles by hiding uninteresting results, and also makes the analysis faster, since it reduces the workload of the SMT solver (Achilles applies the mask before calling the SMT solver).

4.5 Evaluation

In this section, we evaluate the ability of Achilles to analyze an implementation of a distributed system and extract Trojan messages. After a description of our experimental setup (4.5.1), we answer the following questions:

- How accurate is Achilles at finding Trojan messages in real distributed systems (§ 4.5.2)?

- What is the impact of Trojan messages discovered by Achilles on real distributed systems(§ 4.5.3)?

- How does Achilles manage large symbolic expressions (§ 4.5.4)?

4.5.1 Setup

We ran experiments on a 16-core (dual Intel E5-2690) machine with 256 GB of RAM running Ubuntu Linux 11.04.

Achilles gathered the client and server predicates by running individual system nodes in the S2E [31] platform. The S2E virtual machine ran 32-bit Ubuntu Linux 10.04.

We applied Achilles to FSP 2.8.1b26 [42], a UDP-based file transfer protocol, and to the latest version of PBFT [29], a Byzantine-fault-tolerant replication system.

FSP is an implementation of a file transfer protocol. An FSP deployment consists of a server and several client utilities, which emulate well-known UNIX core utilities, such as *cp*, *mv*, *rm*, etc. The FSP implementation of these utilities parses command-line arguments (usually a target file path and a set of flags), tweaks the path to follow some protocol-specific rules (e.g., always start paths with '/'), and finally generates a corresponding command for the server. The server parses the command, performs the corresponding action on its local file system, and replies to the client.

An FSP command message has the following fields:

- *cmd* - 1-byte identifier of the requested action
- *sum* - 1-byte checksum
- *bb_key* - 2-byte message key
- *bb_seq* - 2-byte message sequence number
- *bb_len* - 2-byte length of file path
- *bb_pos* - 4-byte position of block in a file
- *buf* - arbitrary-length payload (file path + file data)

In our evaluation, we approximated the values of *sum*, *bb_key*, *bb_seq* and *bb_pos*: we added annotations in both the client and the server in order to bypass the checks on these fields, such that the client writes a predefined constant value and the server checks that value (there are other approaches that can alleviate the problem of checksums, authentication or encryption for symbolic execution [26]). We focused our evaluation on how FSP parses file paths. We started the FSP clients with symbolic command line arguments (of fixed length) and inserted symbolic data in any system calls from the client. In the FSP server, we set *accept* markers at the point where it invokes system calls to make changes to its local file system, as requested by the command received from the client.

PBFT is a Byzantine fault-tolerant replication protocol implementation. PBFT clients send requests to a set of replicas. The replicas ensure total order among requests from all clients, and forward them to an upper layer application.

A PBFT client request has the following fields:

- *tag* - 2-byte identifier of the message type
- *extra* - 2-byte flags field (1 bit per flag)
- *size* - 4-byte length of message
- *od* - 16-byte message digest
- *replier* - 2-byte identifier of responsible replica
- *command_size* - 2-byte length of command
- *cid* - 2-byte client id
- *rid* - 2-byte request id
- *command* - arbitrary-length command payload
- *MAC* - list of message authenticators, signed with a private key for each replica

In our evaluation, we approximated the values of the digest and authenticator fields: we used annotations in both the client and server in the same way as for FSP, replacing the digest and authenticators with predefined constant values. In the PBFT replica, we also over-approximated local state: the replicas keep an internal data structure to track previous requests from a given client, which we over-approximate with unconstrained symbolic values. We started a PBFT client and generated a request with symbolic *extra*, *replier*, *rid*, *cid*, and *command*. We set a fixed length for the *command*, list of authenticators, and for the overall message. We considered a message to be accepted when the replica generates a *Pre_prepare* message for the client request, initiating the agreement protocol.

4.5.2 Accuracy of Achilles

The purpose of this experiment is to quantify the accuracy of Achilles at finding Trojan messages. We compute the overlap between the expression of Trojan messages computed by Achilles and a known set of Trojan messages in the protocol under test. Ideally, the overlap should be perfect—the representation generated by Achilles should cover all known Trojan messages, and no known non-Trojan messages.

We evaluated the accuracy of Achilles on real Trojan messages in FSP. Achilles discovered that the FSP server accepts messages where the file path length is less than the length reported in the message header (more details in subsection 4.5.3).

To enable symbolic execution to complete, we restricted the FSP clients and servers to only handle file paths with length less than 5. For this scenario, we can mathematically compute how many different types of Trojan messages exist: there is one Trojan message for reported length 1 (the file path is empty), two Trojan messages for reported length 2 (file path is empty or has length 1), three Trojan messages for reported length 3 and four Trojan messages for reported length 4. We analyze eight FSP clients that have a single file path as argument. Therefore, there are in total $(1+2+3+4)*8 = 80$ Trojan messages that differ in the reported length of the file path, request type or true length of the path.

The bounded path size enabled symbolic execution to run to completion. The total testing time for Achilles to find all Trojan messages was approximately 1 hour, which was split as follows:

- Gathering the client predicate took 3 minutes
- Preprocessing the client predicate took 15 minutes.
- Analyzing the server took 45 minutes.

Figure 4.10 shows the percentage of the known Trojan messages discovered by Achilles, as a function of server analysis time. Achilles produced the first Trojan message after 20 minutes of server analysis and discovered all Trojan messages in 43 minutes. Achilles did not produce any false positives. The figure shows how Achilles produces Trojan messages incrementally, as it analyzes the server; even if the analysis is interrupted before completion, Achilles produces valuable results.

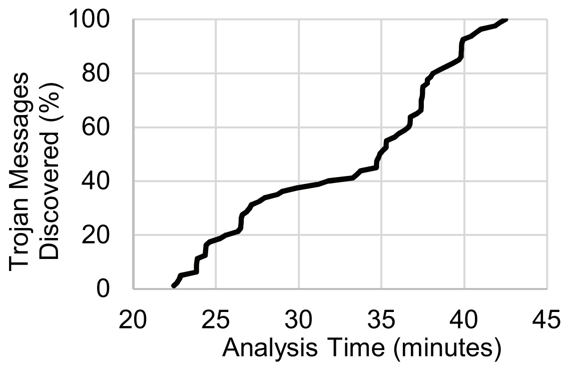

Figure 4.10 – Percentage of real Trojan messages in FSP discovered by Achilles, as a function of time.

We compared Achilles to classic symbolic execution of a server node. We ran the FSP server in vanilla S2E, under the same conditions as Achilles (same bounds, annotations and approximation). Table 4.1 summarizes the results. Achilles discovers all Trojan messages and produces no false negatives. Classic symbolic execution also found all Trojan messages in 2 minutes, but with low signal-to-noise ratio. Trojan messages are “hidden” among valid messages, and it is left to the developer to sift among the results.

	Achilles	Classic symbolic execution
True Positives	80	80
False Positives	0	7,520

Table 4.1 – Results obtained by Achilles in 1 hour, compared to classic symbolic execution

As expected, classic symbolic execution of the server is faster than Achilles, since it performs fewer computations (it does not need to combine server and client constraints). However, classic symbolic execution cannot identify Trojan messages among other, valid messages. Even worse, in the case of FSP, all Trojan messages are on the same server execution paths as non-Trojan messages (as opposed to having execution paths that only contain Trojan messages and execution paths that only contain non-Trojan messages). This makes it difficult for the human operators to sift through results and discover Trojan messages—they need to investigate not only each execution path, but also each message on each execution path. This is a difficult task; as we also discuss in the Design section (subsection 4.2.2), current SMT solvers are not designed to enumerate all values that satisfy a given constraint and are inefficient at doing so.

We ran the same experiment for PBFT. Achilles discovered a single type of Trojan message (more details in subsection 4.5.3). Surprisingly, PBFT replicas make few checks on the data received from clients. They verify that request ids are recent and have not already been handled, verify that the client id is in a set of known clients and also check if the flags field marks the request as read-only.

Due to the simplicity of checks on the client request fields, Achilles completed the PBFT analysis in just a few seconds. The Trojan message discovered by Achilles appears on all execution paths in the server; however, just like for FSP, it is bundled on the same execution paths with valid messages. Thus, classic symbolic execution cannot easily identify the Trojan message among the valid messages, while Achilles identifies it accurately.

We also make a theoretical comparison between Achilles and a naive black-box fuzzing tool. There are other, more sophisticated fuzzers, which use various heuristics to improve performance. However, we are not aware of any fuzzer that optimizes towards anything similar to Trojan messages, so it would be difficult to obtain an unbiased comparison.

We first measured the maximum throughput that a fuzzing tool could achieve on FSP on our testbed: 75,000 tests per minute. Then, we compute the probability of discovering a Trojan message at random. In order to be fair, we only fuzz the same message fields that are analyzed by Achilles and by classic symbolic execution. There are 8 bytes relevant to the Trojan messages: *cmd*, *bb_len* and *buf*. There are 8 relevant possible values for the *command* field (corresponding to commands with a single file path argument). There are 4 relevant values for the *length* field. The FSP server only accepts printable characters in the file path (ASCII codes 33 to 126). Thus, there are $94 * 94 * 94 * 8 = 6.6$ million messages that exhibit a Trojan message for length 1 (first character in the message is '\0', second must be '\0' due to server checks, other three can be any ASCII character with code 33-126). In total, there are 66 million Trojan messages out of the $256^8 = 1.8 * 10^{19}$ possible bit combinations. This means that the expected number of Trojan messages found in 1 hour of fuzzing is 0.00001. Fuzzing also produces 4.5 million non-Trojan messages, which correspond to false positives.

Our experiments show that Achilles is significantly more efficient at discovering Trojan messages than fuzzing or classic symbolic execution. While classic symbolic can find all messages accepted by the server faster than Achilles, it cannot disambiguate between Trojan and non-Trojan messages. Black-box fuzzing is several orders of magnitude worse than symbolic execution.

4.5.3 Bugs Triggered by Trojan Messages

FSP: The Wildcard Character. Achilles discovered a bug in FSP related to the handling of the '*' wildcard character. In essence, FSP clients always expand the '*' wildcard before sending a command to the server. There is no possibility of escaping the '*' character, thus correct clients cannot send a command with '*' in a source file path to the server. The server, however, accepts any printable character, including '*' in the file paths it receives. This leads to an interesting behavior where it is possible to create files containing '*' on the server, but not possible to delete them directly.

In UNIX environments, shells use a simple form of pattern matching in order to expand file paths using wildcard characters (this simple pattern matching is referred to as globbing). For example, the command *'rm file*'* will delete all files with name prefixed by *'file'* from the current directory. The '*' character is a wildcard that can be expanded to any sequence of characters. The expansion of the wildcard is handled directly by the shell, before invoking the *rm* utility. Suppose the current directory has files *'file1'*, *'file2'* and *'file3'* and that the user executes the command *'rm file*'*. The shell will initially parse the command and expand *'rm file*'* into *'rm file1 file2 file3'*. Then the shell invokes *'rm'* with the three command line arguments.

The FSP implementation emulates the regular UNIX globbing behavior. When parsing command-line arguments, the FSP client will first try to expand any *source* file path containing a wildcard. Unlike most shells, however, the FSP globbing does not allow the user to escape any wildcard character. *Destination* file paths, however, are not globbed; for example the *'mv file1* file2*'* will rename any file that matches the pattern *'file1*'* to the literal string *'file2*'*. The FSP server handles wildcards like any regular character.

Trojan messages in FSP can lead to an interesting scenario, where it is possible to create a file called *'file*'* on the server, but it is then difficult to actually remove the respective file. A file called *'file*'* can be created by a user of FSP (e.g., *'mv file file*'*), by a malicious third party that has access to the server, or even by a bit flip that appears on the client during a command execution (a single bit flip can convert the ASCII 'j' character into '*').

Once the file named *'file*'* appears on the server, it is difficult to remove. Calling *'rm file*'* would remove *'file*'*, but would also remove any other file prefixed by *'file'*, including a potentially valuable *'fileWithAllMyBankAccounts'*. Similarly, *'mv file* fileToDelete'* would rename all files prefixed by *'file'* to *'fileToDelete'*, removing all but one of the original files. Calling *'rm file*'* would delete all files prefixed by *'file\'*, since there is no escape character in the FSP globbing.

It is interesting to point out that this bug discovered using Achilles is a semantic bug, which makes it difficult to detect automatically using other approaches. While there are techniques to detect buffer overflows or divisions by zero automatically, the FSP bug cannot be found unless the developers write a complex (and potentially buggy) specification of correct behavior. This is one of the main benefits of finding Trojan messages.

FSP: Mismatched String Lengths. Achilles also discovered that the FSP server does not check whether the file path lengths it receives in commands match the actual lengths. It accepts messages that have the actual file path length smaller than the value in the message's *length* field (i.e., there is a *'\0'* character in the file path). The bug allows malicious users to send an additional arbitrary payload to the server.

While the mismatched length does not have a particularly high impact for FSP, incorrect handling of length can lead to important security bugs, like buffer overflows or information leaks (e.g., the Heartbleed bug [1]). Typically, sender nodes generate string lengths themselves, using functions such as `strlen`, which compute the lengths correctly. Thus, we argue that many remote buffer overflow bugs are triggered by Trojan Messages and, as in the case of FSP, Achilles can find them automatically.

PBFT: The MAC Attack. Achilles discovered that PBFT replicas (servers) accept client requests without checking their authentication code. This leads to a known vulnerability of the protocol, known as the MAC attack [35].

Clients can send messages with incorrect authenticators. The first replica to receive the client request does not verify any of the authenticators. If forwards the message to other replicas, which discover the incorrect authenticator, but cannot know whether the original client or the first replica have corrupted the message. In order to guarantee progress, they initiate an expensive recovery protocol, which impacts performance. This allows incorrect nodes to have a significant impact on the performance of the system. A node whose private key was corrupted due to a memory error will always produce incorrect MAC authenticators and will trigger recovery. Alternatively, a malicious client can also corrupt its own messages in order to trigger the expensive recovery mechanism and slow down the system, affecting the service of other, correct clients.

This vulnerability is an interesting example of the subtle effects that Trojan messages can have

[1] http://heartbleed.com/

on a system.

4.5.4 Effect of Optimizations

In this section, we analyze internal details of Achilles. We look at how the optimizations described in item 4.2.2 enable Achilles to handle large client predicates.

Figure 4.11 shows the number of client path predicates Achilles keeps for each execution path analyzed during the FSP server exploration, as a function of the length of the path. The figure shows results for the paths as they are incrementally generated—many of the points in the figure represent incomplete paths. A point at length 7 with 1,000 matching predicates means that there exists an execution path in the server which encountered 6 branching points that depend on symbolic data, and can be triggered by messages in 1,000 different path predicates. Recall from subsection 4.2.2 that Achilles uses these path predicates to search for any existing Trojan messages—the more path predicates in a state, the more complex the check.

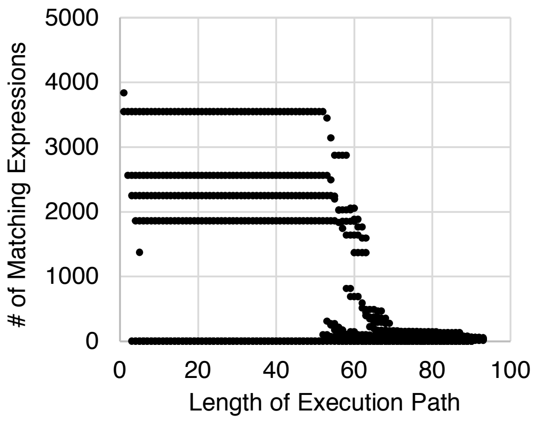

Figure 4.11 – Number of client path predicates that can trigger each execution path in the FSP server, as a function of the length of the path.

The number of path predicates decreases as the length of paths increases—longer execution paths become more and more specialized, and can be triggered by a smaller group of messages. This makes the check for Trojan messages easier, as Achilles makes fewer calls to the solver.

To quantify the effect of optimizations, we compared Achilles to a non-optimized implementation of symbolic constraint differencing. We ran unmodified S2E on both the FSP client and server, and then computed Trojan messages a posteriori. The total execution time of the non-optimized implementation was 2 hours and 15 minutes, compared to the 1 hour and 3 minutes required by Achilles.

5 Elastic Instrumentation

5.1 Intuition

In this chapter, we describe a mechanism for enabling instrumentation in production, in order to reduce the impact of any corner-case bugs lurking in the code. The insight behind our work is that many types of instrumentation can be split in small atoms, and the benefit of the instrumentation is actually the sum of the benefits of each individual atom. Thus, the trade-off between benefit and cost of instrumentation becomes a continuum, rather than a binary choice. Moreover, few execution paths in a program are performance-critical—atoms of instrumentation can be enabled on many execution paths (and reduce the impact of many bugs) with little cost in terms of overhead.

5.1.1 A Survey of System Call Vulnerabilities in the Linux Kernel

To validate the intuition, we present the results of a thorough study of vulnerabilities in the Linux kernel that can be triggered via system calls. We show that:

- there are many vulnerabilities that can lead to a complete compromise of the system;
- more than one third of these vulnerabilities are memory errors, which could be detected by instrumentation tools;
- most vulnerabilities appear in system calls that are rarely used in regular userspace programs.

5.1.2 The Case for Instrumentation

First, we analyzed high-impact vulnerabilities in the Linux kernel. We used the National Vulnerability Database search engine[1] to find vulnerabilities that allow attackers to gain root privileges

[1] https://web.nvd.nist.gov/view/vuln/search

Search Query	# results
linux privileges array index	7
linux privileges buffer overflow	63
linux privileges use after free	7
linux privileges pointer	26
linux privileges length	15
linux privileges address	10
Total	128

Table 5.1 – Search queries in the National Vulnerability Database search engine and their respective number of results.

on a system (thus completely compromising the system). Running the query "linux privileges" yielded 372 such vulnerabilities. We estimated how many of these are due to memory errors by running search queries with keywords common to memory errors, as summarized in Table 5.1.

We manually validated the data by looking at vulnerabilities reported between January 2013 and July 2014. Out of 59 vulnerabilities that allow attackers to gain privileges, 32 are memory management errors.

This data makes apparent once again how vulnerable much of our infrastructure is; considering that the last two years brought 59 newly discovered critical vulnerabilities, it is easy to imagine that hundreds more will be uncovered in the years to come. Yet, a significant fraction of these 59 vulnerabilities could have been mitigated by inserting instrumentation into production code (32 of them are memory safety bugs).

5.1.3 The Case for Elastic Instrumentation

In this part of the survey, we analyzed the distribution of *system call configurations*. We define a system call configuration to be tuples of the form <*name, arg*>, where *name* is a system call name and *arg* is a named constant (defined in Linux headers) that is passed as an argument. Configurations carry more information than the system call name alone and allow a finer partitioning of system calls that corresponds to the sections of kernel code that are eventually run. For example, the *socket* system call can actually instantiate dozens of different types of sockets.[2] The behavior of the socket system call differs significantly depending on its parameters, and our definition captures this as different configurations.

We measured the popularity of system call configurations by analyzing the source code of packages in the Debian repository. Our scripts use the C preprocessor and the M4 macro processor to eliminate comments from source code, perform proper parenthesis matching and search for function calls that match system call names. For each such call, we removed all

[2]Domains are defined at http://lxr.free-electrons.com/source/include/linux/socket.h#L141

parameters that do not match constants defined in the user-facing Linux headers, replacing them with the wildcard *any*. Thus, a function call such as *open(path, O_RDONLY)* becomes *open(any, O_RDONLY)*—the function call is stripped to just the template that identifies that system call configuration. By analyzing the Debian repository packages—more than 100 GB of source files—we found 490,514 function calls that match system calls. They are categorized in 2,492 different system call configurations.

We also analyzed the distribution of known vulnerabilities among system call configurations. Given a vulnerability from the CVE database, we asked which system call configuration would trigger the error. We wrote a script that parses the entire CVE database and analyzes the CVE descriptions to find references to system calls. We manually analyzed over 3,000 matches, trimming the list to 178 Linux kernel vulnerabilities for which we could identify the system call configuration (note that we discarded vulnerabilities that are triggered by user-level programs communicating directly to modules via the `/dev/` interface).

Finally, we matched the data for system call configuration popularity with the data for vulnerabilities. The data shows that most vulnerabilities are in "unpopular" system call configurations. In fact, 36 of the vulnerabilities are in system call configurations that we *never* found in Debian package repositories. One particular example is CVE-2000-0461 [2], in which the "kernel contained an undocumented system call," which had not yet been fully released and therefore was not used by any application. This result gives further evidence for the intuition that unused code contains more bugs than regularly exercised code. These 36 vulnerabilities could have been safely avoided (on a system using only Debian packages) by completely removing the code that contains them; we discuss some approaches that automate this removal in chapter 2. Yet, for the remaining vulnerabilities, removing the code would affect the functionality of some applications.

Figure 5.1 shows the comparison between the number of times system calls configurations appear in the Debian repository and their security exposure. We found that 437,000 of the system call configurations that appear in source code from Debian repositories, which correspond to 90% of all system calls, are configurations that appear more than 300 times each. Instrumenting these configurations would therefore negatively impact the performance of many user-level programs. These configurations are also well exercised—many programs use them in many different ways, effectively testing them and leading to possible bugs being reported early. As the figure shows, only 13% of syscall vulnerabilities that were ever reported to the CVE database are triggered via one of these "hot system calls". Most vulnerabilities are triggered by uncommon system call configurations, with 60% of vulnerabilities appearing in configurations that appear less than 10 times each in Debian source code.

5.2 The Elastic Instrumentation Principle

The survey suggests the high potential of selectively instrumenting system calls in the Linux kernel. Many Linux vulnerabilities lie in rarely used system calls; for these corner case-executions,

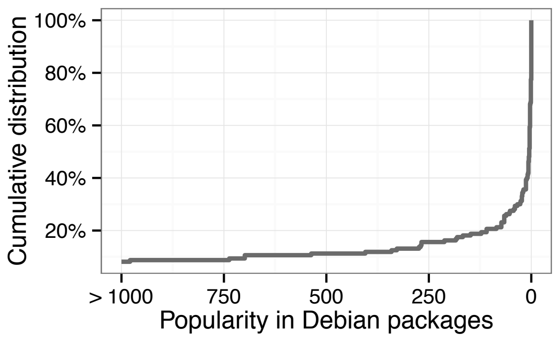

Figure 5.1 – The distribution of vulnerabilities in the kernel as a function of system call configuration popularity in Debian repositories.

the performance overhead of instrumentation has little effect on the overall performance of the kernel and thus powerful instrumentation is no longer inaccessible.

We now introduce the Elastic Instrumentation principle[3]. We first provide a model to clarify why and how instrumentation should be made elastic (§5.2.1) and then explain under which conditions this can be achieved automatically using our framework (§5.2.2).

5.2.1 Model

In our model, we think of instrumentation as a set of atoms—an individual memory safety check, a logging statement, etc. Each such atom consists of a sequence of instructions in the target program.

We say that an atom of instrumentation is *useful* only in program states where its execution is essential for achieving the desired effect of the instrumentation. By this definition, a memory safety check is useful only when it fails and detects a memory corruption, and a logging statement is useful when it distinguishes different program executions.

An atom that is executed in a state where it is useful is a true positive (TP). Otherwise, if it is executed but ends up not being useful, we classify it as false positive (FP). When instrumentation is disabled or used selectively, there are also cases where instrumentation was not used but could have helped (a false negative, FN) and where it was not used and would not have helped (a true negative, TN).

Currently, developers must choose between two extremes: Not use instrumentation at all and maximize the FNs, or use full instrumentation and maximize FPs. False positives lead to overhead

[3]Note that the Elastic Instrumentation Principle and the framework are joint work with another PhD student, Jonas Wagner. As the main author of ASAP [101], Jonas approached the problem from the perspective of selecting atoms of instrumentation statically, at particular lines of code in a given system, while this thesis focuses on selecting atoms of instrumentation dynamically, on particular execution paths.

that can cause SLAs to be violated, drive users away from a website (e.g., because the web page takes more than two seconds to load), cause an interactive application to feel sluggish, etc.

Our approach measures the cost (the number of false positives) and benefit (the likelihood of a true positive) for each atom of instrumentation. It then activates atoms either statically or dynamically.

Static activation decides at compile-time whether to activate an atom or not. It maximizes benefit under a given cost budget by selecting the atoms with the highest overall benefit-to-cost ratio. Dynamic activation can defer this decision, and evaluate cost and benefit for a particular program state encountered at runtime. This incurs a cost to examine the program state, but can reduce the number of false positives by using extra information.

5.2.2 A Recipe for Elastic Instrumentation

We now outline the factors that a user needs to consider when applying Elastic Instrumentation.

1. Identify a Problem We start from a problem that can be solved with instrumentation. A QA team might find that extra logging would help them reproduce bug reports; an engineer might want to track down a performance problem using extra tracing; a developer might want to harden a security-critical system against memory errors. All these cases can be solved through instrumentation, either added manually or by a tool.

At this step, the users focus on solving the problem. The performance optimization is handled by our framework, as long as the instrumentation satisfies the basic properties outlined below.

2. Analyze the Instrumentation Three properties determine how well an instrumentation lends itself to Elastic Instrumentation: *granularity, elasticity,* and *cost/benefit distribution*.

Granularity measures how well the instrumentation code can be subdivided into individual chunks. Log and tracing statements, many types of safety checks, and assertions are examples of code that is very granular: it is composed of many small and independent statements. High granularity makes it possible to precisely select the best benefit/cost trade-off and fine-tune the overhead. In practice, users can measure the granularity using the code analysis tools we provide, by looking at the statistics produced by the instrumentation tool, or manually, e.g., by grepping through source code.

Elasticity quantifies how much the overheads of the instrumentation can be controlled by the tool. Elasticity is related to granularity: instrumentation is fully elastic when each individual instrumentation atom can be enabled individually and does not rely on any other instrumentation code. In practice, many instrumentation tools insert extra code for updating metadata that is used by multiple atoms and thus cannot be removed independently. Users can estimate elasticity by

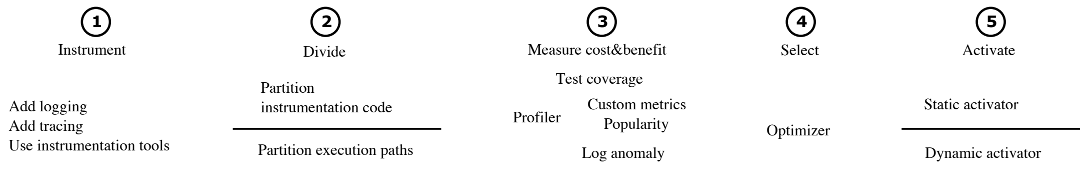

Figure 5.2 – Elastic Instrumentation consists of five steps. The output of the first step is a program augmented with instrumentation. The second step is to divide this instrumentation into atoms that can be individually enabled or disabled. We then measure the cost and benefit of each of these atoms (3), and pass this data to an optimizer that selects which atoms are worth enabling (4). Finally, the fifth step generates a selectively instrumented program. For each step, the figure also shows the individual components that implement the process.

configuring our framework to remove all instrumentation atoms, and then measuring the overhead of the generated program. This measures the cost of logging at the least verbose level, or the cost of memory safety instrumentation with all checks removed. We call this remaining cost the *residual overhead*. For example, memory safety instrumentation tools keep track of allocated objects, regardless of whether checks are enabled or not. The residual overhead is a lower bound on the effectiveness of Elastic Instrumentation.

Cost/benefit measurements give a sense of how much users can gain from the Elastic Instrumentation approach in practice. The key insight is that both cost and benefit are often highly skewed: a security check in a hot loop is much more expensive than a check in rarely-executed error-handling code, and instrumenting a rarely-used system call has a higher expected benefit than instrumenting a system call that is widely used and thoroughly tested. Because of 80/20 rules, users can expect highly skewed distributions, and therefore careful optimization can yield both high benefit and low cost.

3. Apply Elastic Instrumentation By this point, the user has selected instrumentation techniques and obtained evidence that the Elastic Instrumentation principle is applicable. To proceed and apply the principle, the user can now choose the design options that best fit the use case. In particular, the user decides how to estimate the cost and benefit of instrumentation (statically or using profiling data) and how to activate it (statically or dynamically). For each step of the approach, the user selects which of the tools in our framework to run. We discuss these choices and our prototypes in the following sections.

Currently, the choice of tools is guided by the choice between static and dynamic activation. Each activation approach has its own specific tools. We discuss in how we envision static and dynamic activation combined. This thesis focuses on the tools specific to dynamic activation.

5.3 The Elastic Instrumentation Framework

We have built a framework to automate the steps for applying the Elastic Instrumentation principle. As input, it takes a target software system and an instrumentation tool, and processes them over the five steps shown in Figure 5.2. The steps are performed while compiling the target and generate a version of it that activates the optimal set of instrumentation atoms.

5.3.1 Dividing the Instrumentation

The purpose of this step is to identify atoms of instrumentation, that is, the smallest parts that can be individually enabled or disabled. We propose two ways of dividing instrumentation: at the level of program execution paths and at the level of program instructions.

Path-based division classifies related execution paths into groups by examining the program state. For example, examining the kernel system call arguments allows to classify execution paths into groups that are triggered by the same system call configuration. Paths in the same group can be assumed to have similar instrumentation costs and benefits and their instrumentation is activated as a whole. Path-based division is specific to dynamic activation.

Instruction-based division attempts to classify each program instruction as either original program code, or as belonging to a sequence of instrumentation code. Instruction-based division lends itself to static activation.

5.3.2 Estimating Cost and Benefit

This section explains how to obtain cost and benefit information for each instrumentation atom identified in the previous step.

For static activation, our framework provides a custom profiler to measure cost. The profiler runs a user workload several times, with and without instrumentation, and computes the contribution of each instrumentation atom (small sequence of instructions) to the overall overhead of the instrumentation.

For dynamic activation, we built a popularity estimation component for our framework in order to quantify cost and benefit. It counts the number of locations in a large code base where a particular API functionality is invoked. The component distinguishes API calls by their function name and parameters, as described for the Linux system call configurations.

Our popularity estimation component is based on the intuition that popularity is an indication of code reliability. This intuition is similar to the results observed by Mileva et al. [80], who analyzed the popularity of library APIs and proposed a reliability metric for specific API versions based on their popularity. Our study shows that most bugs are in unpopular system call configurations[4] (subsection 5.1.1).

Our metrics are rather simple in comparison to existing work. Software engineering researchers have built tools to predict the location of software defects from version control histories and bug databases [86, 83]. Another approach uses data from program runs to identify statistical bug predictors, i.e., program behaviors that are likely to encounter a failure [20]. For logging, the SherLog project [105] proposed a methodology to estimate how useful a log statement is for finding a bug's root cause. Also, sophisticated commercial tools exist to analyze logfiles and prioritize logfile entries [9, 10, 12].

We believe that our framework is general enough that such techniques could be incorporated. Metrics that apply to static program locations ([86, 83]) are well suited to our instruction-based division of instrumentation and static activation. Metrics that assign benefit to program behaviors ([20]) are useful for path-based division and dynamic activation.

5.3.3 Optimally Selecting Instrumentation

After cost/benefit estimation, we have a set of instrumentation atoms with their benefits and costs, and are facing the problem of selecting a subset to enable. Users can choose to optimize for either cost or benefit.

This is an instance of the well-known Knapsack problem. Despite its NP-hardness, several algorithms can solve practical instances sufficiently quickly for our case ([77]).

For our prototypes, we chose to use a simple greedy algorithm that computes an approximately optimal set of instrumentation atoms. It first sorts the atoms by decreasing benefit-to-cost ratio. The algorithm then iterates through the atoms in this order, adding the current chunk to the set if its cost fits into the budget.

Three reasons made us opt for that choice. Firstly, the main cause for the difficulty of the Knapsack problem is high correlation between cost and benefit. In our experiments, we did not observe such correlation. On the contrary: atoms that are most beneficial tend to be in obscure, cold parts of software, and are thus likely cheap. In such cases, our approximation algorithm produces the optimal result. Secondly, we wanted our prototypes to be simple. Complex decisions can make compilation slow (in the case of static activation) or runtime decisions slow (in the case of dynamic activation). Thirdly, the greedy algorithm lends itself very well to adaptive activation schemes using a runtime monitor.

Consider that the greedy algorithm adds atoms to the selected set as long as the user's constraints

[4]Note that we have not investigated bug density in detail, as we do not have an enumeration of all possible system call configurations. Popularity is also an indicator of overall cost, which contributes to its effectiveness as a metric in our approach. We do not yet know what is the contribution of popularity as a benefit metric and as a cost metric in the results we obtained.

are satisfied. This implies the existence of a threshold cost-to-benefit ratio, t, for which all checks with cost-to-benefit $> t$ are in the set. When using a dynamic activation scheme as described in subsection 5.3.4, we can execute an atom if its cost-to-benefit ratio exceeds the value of a global variable t_{runtime}. A runtime monitor can set t_{runtime} according to the current performance; the monitor can increase it if performance is unsatisfactory, and decrease it if extra resources become available for use by instrumentation.

5.3.4 Activating Instrumentation

When instrumentation atoms have been selected, users of Elastic Instrumentation can choose from two activation mechanisms.

They can *activate instrumentation statically* at compile time using the static activator component of our framework. This is the simplest activation strategy. It has the benefit that there is no runtime overhead due to the activation: the static activator generates a binary where the instrumentation atoms selected in the previous step are present, and the rest are eliminated. Static activation is applicable if cost and benefit values do not depend on parameters known only at runtime.

Secondly, the dynamic activator can generate a program where all the instrumentation is compiled in, but guarded by a decision procedure. This allows *activation based on context-sensitive properties.*

Figure 5.3 describes at a high level how dynamic activation modifies the target system. Our framework duplicates each function in the target system and applies instrumentation to only one of the copies. It then inserts a decision layer at the entry point (or entry points) of the system; the decision of whether to execute the instrumented or non-instrumented version of the code is taken at runtime. Shared data, such as global or static memory is not duplicated. Since instrumentation does not change the semantics of the application, both the instrumented and non-instrumented version of a function can operate on the same shared data, without conflicts. This allows transparent interleaving of instrumented and non-instrumented executions.

We use the dynamic activator to instrument the Linux kernel section 4.5. For each system call configuration, the instrumentation selection step decided whether that configuration should be protected. The activation step generates code that sets or resets the guard flag according to that decision. We inserted the generated decision procedure code manually at the system call entry point. The kernel is particularly suited for dynamic activation: it is a large, monolithic piece of software and thus has many functions whose benefit depends on the context in which they are called. For example, the `copy_from_user` function, which is used to copy data from user space to kernel space, does not check bounds and can corrupt memory if used improperly—the cost and benefit of instrumentation inside this function depends on the calling context.

Dynamic activation is more flexible than static activation. Cost and benefit of a chunk need no longer be constants—they can be context-sensitive. This allows the system to protect code in

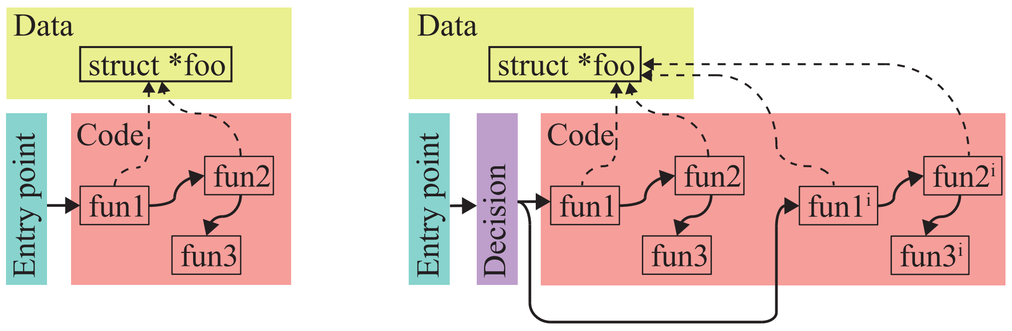

Figure 5.3 – The results of Elastic Instrumentation using dynamic activation to a system. The left side of the figure shows the original system and the right side shows the same system after applying our framework. Functions marked by superscript "i" are instrumented.

some circumstances but not in others. The downside of this scheme is that evaluating the context introduces extra overhead, beyond that of the instrumentation itself.

Dynamic activation can be further enhanced using runtime properties beyond just the calling context of a function. A runtime monitor can execute concurrently to the instrumented software to monitor performance. This monitor can measure request latencies, system load, or SLA compliance, and consider these parameters when deciding whether to enable instrumentation or not. This approach has the advantage that users get tight control over the overhead budget. It is useful in cases where workloads are variable and hard to predict.

5.4 Implementation

We implemented the dynamic activation pass in the *gcc* compiler. The purpose of this pass is to tweak the generated binary of the target in order to allow deciding at runtime whether to enable or disable instrumentation. Thus, unlike the static activation, the dynamic activation compiler pass does not use any decision procedure at compile time, but rather generates a mechanism through which the decision can be taken at runtime.

The dynamic activation pass generates multiple versions of each function in the target program and inserts a dispatching mechanism at the beginning of each function. Our pass runs early in the compilation; it marks each version of the function using function attributes, such that the versions will be handled differently by the compilation passes that add or tweak instrumentation. Currently, our prototype only generates two versions of each function (the original and a clone) and marks the clone using the "no_sanitize_address" attribute, such that the Address Sanitizer pass does not instrument it.

Our pass also inserts a dispatcher at the beginning of each original function. This dispatcher is needed for every function to avoid the execution of a system call unintentionally switching between the instrumented and non-instrumented versions (this can happen, for instance, due to

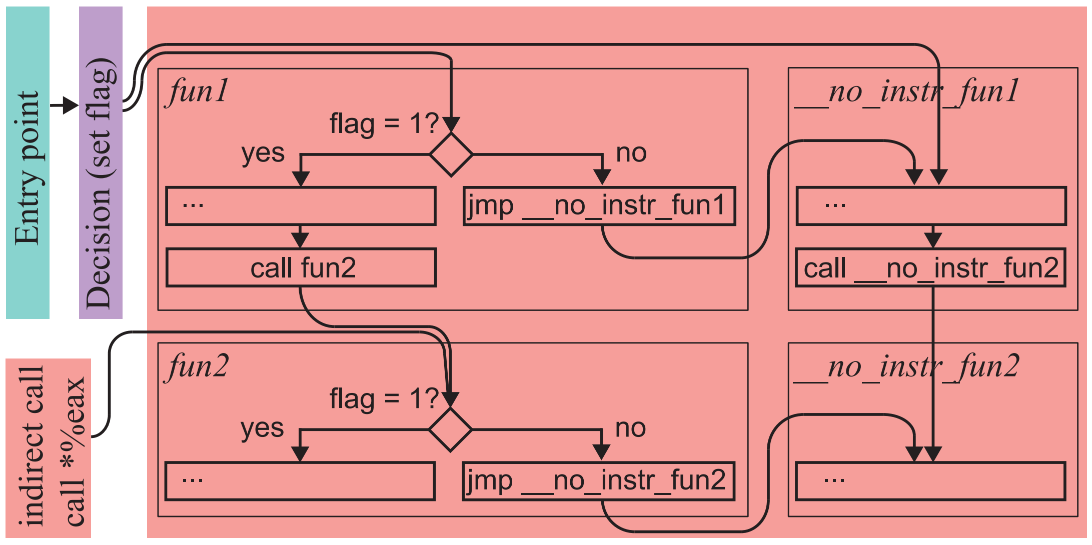

Figure 5.4 – Detailed illustration of the binary generated by the dynamic instrumentation pass.

indirect calls, as we describe in the next paragraph). However, the dispatcher does not need to re-evaluate the context every time; the context is only evaluated once, at the system call entry point, after which a globally-accessible thread-local flag is set accordingly. The dispatchers in the functions only checks this flag and jumps to the clone when the flag is set.

Figure 5.4 shows in more detail the binary generated by our dynamic activation pass. In the figure, functions *fun1* and *fun2* are instrumented by Elastic Instrumentation. One clone is created for each, prefixed by *__non_instr*. The original functions themselves become dispatchers, through the condition inserted at the beginning of each function. As an optimization, our pass changes direct calls from the non-instrumented versions of functions to directly point to other functions with no instrumentation (i.e., without going through the dispatcher again). However, this optimization is difficult to perform for indirect calls, whose targets cannot be determined at compile time[5]. Therefore, indirect calls remain untouched; at runtime, any such indirect call would invoke the original function, which has the dispatcher built-in, so the dispatcher would redirect to the proper version. Thus, the "fast path" through the program, where no instrumentation is applied, will only incur the cost of the flag check once at the program entry point and once for each indirect call that cannot be handled statically by the compiler.

5.5 Evaluation

In this section, we describe how we used our Elastic Instrumentation framework to add memory safety instrumentation to the Linux Kernel. We answer the following questions:

[5]The Split Kernel [65] uses an ingenious trick to handle indirect calls statically, by placing the original and instrumented functions at fixed offsets from each other and using pointer arithmetic before any indirect call. The same trick could be applied to our implementation.

- Would Elastic Instrumentation have helped preventing recent security vulnerabilities (subsection 5.5.2)?
- What overheads and benefits can be achieved when using the Elastic Instrumentation approach in practice (subsection 5.5.3)?

5.5.1 Setup

Our evaluation measures both effectiveness and performance. In our benchmarks, we measure *effectiveness* using a reference set of known bugs and vulnerabilities. These are real-world vulnerabilities, present in past versions of the kernel, that have been reported to the Common Vulnerabilities and Exposures (CVE) database. We re-introduced these vulnerabilities into the kernel, by reverse-applying the patch that fixed them. We also wrote a script to trigger each of these vulnerabilities, either using a real exploit or—if no known exploit was available—using a hook in the kernel itself.

We use security vulnerabilities because they are publicly available, representative of real-world problems, and often in the scope of instrumentation tools. However, it is worth keeping in mind that the instrumentation tool we use (Kernel AddressSanitizer[6]) was built for debugging, rather than security; it reduces the attack surface and can make vulnerabilities harder to exploit, but does not provide a comprehensive security guarantee.

We measure *performance* by measuring the performance of user-level applications running on top of the kernel. We compare overhead relative to the same application running on an uninstrumented baseline version of the kernel.

We use two sets of vulnerabilities as reference sets.

First, we attempt to reproduce recent high-impact vulnerabilities. We analyzed vulnerabilities reported to the CVE database between January 2013 and July 2014 and found eleven that can potentially lead to local malicious applications gaining root access, completely compromising the target machine. Two of them are not due to memory errors, so they are not in the scope of this experiment, but the remaining nine are memory errors. We "ported" the vulnerable code that enables these nine exploits to version 3.19 of the kernel, by analyzing the patch that fixed them and then reversing the patch.

For three of the vulnerabilities, we found public exploits that gain root access; we managed to reproduce these exploits on our kernel. For the remaining six, we wrote our own test cases to exercise the bug that enables them. Our test cases do not attempt to gain root access, but rather just trigger a simple out-of-bounds write. We use this first set to evaluate the usefulness of Elastic Instrumentation (subsection 5.5.2).

Our second reference set is the data we obtained from the survey (subsection 5.1.1): 178

[6]https://code.google.com/p/address-sanitizer/wiki/AddressSanitizerForKernel

vulnerabilities in the Linux kernel for which we could identify the system call configuration. We use these as reference when we evaluate the efficiency of our approach (subsection 5.5.3). Note that we did not reproduce these 178 vulnerabilities, but instead report whether Elastic Instrumentation enables instrumentation for the respective syscall configuration that triggers each vulnerability.

We compiled the kernel with regular Kernel AddressSanitizer (KASan) and with Elastic Instrumentation. The elastic version uses the dynamic activator; it executes code with or without instrumentation based on the popularity of the system call configuration that is being processed. The popularity estimator from our framework measured each system call configuration's popularity, and generated code to evaluate it at runtime—a large nested C *switch* statement that takes as input the system call number and parameters and outputs the popularity. We inserted the generated code in the kernel, at the system call entry point. This required us to write ~10 additional lines of code. We also wrote the decision function—an *if* statement that compares the popularity value with a fixed threshold.

When studying exploits, we noticed that file descriptors in system calls are also highly discriminant for system call behavior. We decided to incorporate file descriptors in the instrumentation activation decision, by keeping track of whether an uncommon system call was ever executed on a file descriptor. This information is stored in the file descriptor metadata. If an uncommon system call was executed having a file descriptor as parameter, then all subsequent system calls on the respective file descriptor will have instrumentation enabled. This change required less than 10 lines of code.

We compared the performance of user-level applications when running on top of the Linux 3.19 kernel. We compiled the kernel with no address sanitization, with full KASan sanitization, and with Elastic Instrumentation using different benefit thresholds. Other than sanitization, all other kernel compilation options were identical. We ran OpenBenchmarking.org benchmarks for SQLite and building PHP, as well as Apache web server version 2.4.6. For Apache, we ran two configurations: one in which the workload generator (ab) runs on the same system as the web server (and is also affected by the overhead in the kernel), and one in which ab runs on a separate machine. Our experiments were performed on a machine with Intel Core i7 CPU (4 cores @ 2.6 GHz) and 16 GB RAM.

5.5.2 Usefulness of Elastic Instrumentation

Kernel root exploits allow non-privileged user-space applications to gain unauthorized root-level access to a system. Some users intentionally execute exploits on their own devices in order to obtain root access and bypass restrictions imposed by the manufacturer. However, such exploits can be silently run by malware in order to compromise the device.

We found publicly-disclosed exploits for three vulnerabilities due to memory errors. KASan flags an error in two of these exploits, at an overhead of 28% for Apache; with Elastic Instrumentation, we can reduce the overhead to 7% while still detecting the same exploits.

The CVE-2013-2094 [6] exploit [5] targets an out-of-bounds access in a global array, which is due to an integer overflow. By carefully controlling the parameters to the *perf_event_open* system call, attackers can increment values at arbitrary memory locations. In order to find the offset of the array in kernel memory and calibrate the arbitrary write primitive it constructs, the exploit first maps a large amount of memory and initializes it to known values. It then triggers the increment once; it "calibrates" the write primitive by checking which value in the mapped region was modified. With this primitive, the exploit can then modify arbitrary kernel memory; in this particular case, it overwrites an interrupt descriptor table entry in order to take control of the kernel control flow. KASan flags the initial "calibration" stage, as the kernel writes directly to user-space-mapped memory. The system call that triggers the bug, *perf_event_open*, is rare, so sanitization is enabled in Elastic Instrumentation when the exploit executes. Both KASan and Elastic Instrumentation detect this exploit and prevent it from executing.

The CVE-2013-1763 [4] exploit [3] targets an array indexing error in NETLINK_SOCK_DIAG sockets. A global array in the Linux kernel is indexed with a one-byte value (the socket family), without first checking whether the index is greater than the maximum number of families. With KASan enabled, the one-byte offset falls into poisoned memory; this bug can no longer be exploited to hijack control flow in the kernel. Since the system call that triggers the exploits is an operation on a rare NETLINK_SOCK_DIAG socket, Elastic Instrumentation enables instrumentation for the respective system call, flagging the exploit.

The CVE-2014-0038 [8] exploit [7] targets a bug where the kernel directly casts a *struct timestruct* pointer received from user space to a kernel pointer. The exploit crafts this pointer such that the kernel overwrites a function pointer, so that a subsequent indirect call is hijacked by the exploit. The size of *timestruct* is 16 bytes, which is larger than the 8 bytes of the function pointer it overwrites. However, KASan does not currently insert poisoned memory in between elements of the same *struct*; since the function pointer is in a larger struct, KASan does not detect this overwrite. Not that the buggy code is in a rarely executed 32-bit compatibility system call; if KASan supported intra-struct poisoning, both KASan and Elastic Instrumentation would detect this exploit.

We also executed our own test cases for the remaining six exploits: CVE-2013-1826, CVE-2013-1827, CVE-2013-1827, CVE-2013-4587, CVE-2013-4588 and CVE-2013-6368. KASan flags five of them. Elastic Instrumentation always enables instrumentation for the system calls that trigger the bug, as they are all uncommon system calls, and flags the same bugs as full KASan.

We observed from this experiment that address sanitization can help prevent some bugs from being exploited. Enabling sanitization in production forces potential attackers to make their exploits more precise, in order to avoid touching poisoned memory. For example, it would not be possible to overwrite a single 8-byte pointer by using a 16-byte *struct*, as the write would also touch some unallocated memory around the target pointer. We argue that this reduces the attack

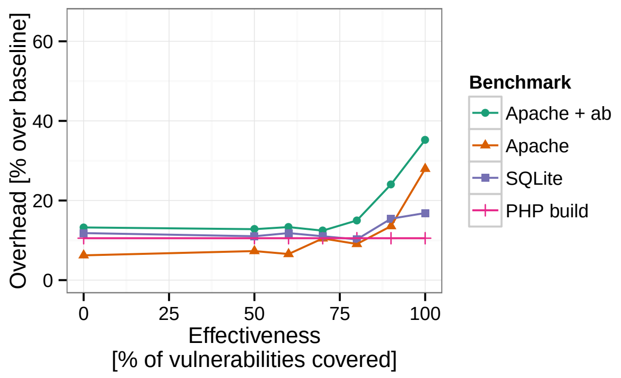

Figure 5.5 – Overhead vs. effectiveness for the Linux kernel

Figure 5.6 – Achievable points in the trade-off between effectiveness and overhead of instrumentation. We use the fraction of vulnerabilities for which the corresponding system call configuration is protected.

surface for existing bugs.

This experiment showed that the Elastic Instrumentation version of the kernel is as protected against recent high-impact as a fully instrumented kernel. Instrumentation can make some bugs unexploitable and others more difficult to exploit. Yet, the Elastic Instrumentation kernel obtains these benefits while incurring 7% overhead for an Apache web server workload, significantly less than the 28% incurred by full instrumentation.

5.5.3 Effectiveness vs. Overhead

We now evaluate the effectiveness and overhead that can be obtained using Elastic Instrumentation. Our approach enables users to control the amount of instrumentation, thus choosing a point in the trade-off between effectiveness and overhead. In this section, we explore the shape of this trade-off curve.

Figure 5.6 shows the levels of effectiveness and overhead that can be achieved for different applications running on top of the Linux kernel. For example, an Apache server running on a fully instrumented kernel will experience a latency increase of 28%. Using Elastic Instrumentation, it is possible to reduce overhead to 9% while enabling instrumentation for 80% of the known vulnerabilities. If the *ab* workload generation tool for Apache is also run on the same machine, the overhead is even higher, as *ab* also makes a large number of system calls and puts pressure on the kernel. SQLite is less kernel-intensive and therefore less affected by the overhead of instrumentation, while building PHP is least affected by the different levels of instrumentation.

Note that, because we do not have exploits for all the vulnerabilities in our survey, we do not know how many vulnerabilities are actually detected by instrumentation; we merely measure the fraction of vulnerabilities for which the buggy system call is instrumented.

The principle of diminishing returns is clearly visible in the convex shape of the curves in this graph. For effectiveness levels from 0 to about 75%, the overhead remains very close to the residual overhead—at least 75% of the vulnerabilities in our experiments lie in cold code where instrumentation is almost free. The remaining 25% get progressively more expensive to instrument.

The graph also shows the residual overhead for Kernel AddressSanitizer. This is the overhead at 0% effectiveness. It is due to various factors, including the overhead of the decision function, changes to the memory allocator, reduced function inlining because instrumentation increases function size, increased instruction cache pressure, bookkeeping to track allocated objects and poisoned areas between objects, and changes to built-in functions. We believe that this residual overhead can be further reduced through engineering efforts.

6 Future Work

In this chapter we discuss several ideas through which the techniques introduced in this thesis can be improved.

One idea that applies to all techniques is better support for developer involvement. This thesis attempts to automate testing and instrumentation as much as possible. Full automation lowers the entry barrier, allowing developers to integrate our techniques with little effort, yet it does not tap in the vast knowledge that human developers have about their systems. Ideally, we would have a system that uses any amount of information the developer is willing to add. Integrating developer knowledge is still an open problem. We discussed in this thesis some isolated examples of ways through which developers' domain-specific knowledge can speed up black-box and white-box testing, but not a principled way of combining human and machine effort throughout the entire development process.

In order to persuade developers to spend effort and add their knowledge to reliability tools, we imagine mechanisms to provide early feedback during code development. Automated tests could run in the background, as developers write new code; by integrating the testing frameworks in development environments (IDEs), the testing frameworks could notify developers immediately when their code is wrong (this idea is explored by other tools as well, e.g. the formal verification tools behind Ironclad Apps [53] integrate with Visual Studio). Since the code is fresh in their mind, it is likely easier for developers to fix bugs.

Achilles can have false positives due to the incompleteness of symbolic execution. To improve on this, we imagine a feedback loop through which a candidate set of Trojan messages is refined over several iterations. Specifically, once a potential Trojan message is discovered, Achilles could re-evaluate the clients, guiding the symbolic execution towards the expression of the Trojan message. As shown by ESD [107] or Demand-driven symbolic execution [18], this focused symbolic execution is significantly faster than "blind" exploration, and can help eliminate false positives in Achilles. This approach is similar in spirit to the abstraction refinement in CEGAR [34].

Our Elastic Instrumentation framework currently has a clear separation between static and dynamic activation. However, the two approaches complement each other, and could be applied together. Static activation has the advantage of performance, as it requires no run-time overhead to decide which version of code to run. Dynamic activation has the advantage of context-sensitivity, which can lead to better decisions. The two approaches can work together: dynamic activation can be used strictly for functions where the cost and benefit of instrumentation is determined to be context-sensitive. The difficulty here is how to determine where cost and benefit are context-sensitive.

Another area in which Elastic Instrumentation can be improved is the granularity of dynamic activation. Currently, the decision of whether to instrument or not is only taken once, at the entry point of a program. The reason for this is that the actually analysis of context and decision taking is an expensive task. However, in some cases it might be worth trading in this overhead for more fine-grained decisions. Dynamic activation currently applies instrumentation at a function-level granularity: either a function is fully instrumented, or not at all. While this worked well for our kernel use-case, other systems might need more fine-level control. This increased granularity could be achieved by cloning a function more than once, with different levels of instrumentation in each clone, or by inserting more decision points in each function.

Finally, we envision a tighter integration between testing and instrumentation. The intuition behind Elastic Instrumentation is that some parts of a system are more reliable than others. By knowing which parts of a system have been more thoroughly tested, Elastic Instrumentation could assess the benefit of instrumenting each part. Conversely, if developers use instrumentation to harden parts of a program, then they can focus their testing and validation efforts on the remaining parts. For example, in the Linux kernel, developers could allocate more resources to the performance-critical system calls that are not instrumented. Since only a minority of system calls are performance-critical, a thorough validation is possible even with limited resources.

7 Conclusion

Modern systems are composed of many components that communicate and coordinate with each other. Corner-case bugs can lurk in the interaction between components. Regular testing techniques cannot find these elusive bugs, so they can remain hidden until software is released in production, with dire consequences. This thesis proposed three techniques to help developers harden their systems against such bugs.

We proposed a black-box fault injection algorithm to search for high-impact failures. By learning and exploiting the inherent patterns of how a system under test handles failures, our algorithm chooses the fault scenarios that are most likely to have high impact. We showed how this approach can find new bugs in mature software, such as the MySQL database management system and the Apache web server.

We also described a white-box approach that can systematically find Trojan messages in the implementation of distributed systems. Trojan messages are a means through which failures propagate among nodes. Thus, they represent the Achilles' heel of distributed system. We showed how our technique found subtle bugs due to Trojan messages in the implementations of the FSP file service protocol and PBFT Byzantine-Fault-Tolerant protocol.

Finally, we proposed the Elastic Instrumentation framework, which enables developers to use instrumentation tools in production in order to harden their systems against remaining corner-case bugs. We showed that instrumentation can often be split into small pieces that can be selectively enabled or disabled in a system. By enabling instrumentation only on corner-case executions, our approach obtains most of the benefit of instrumentation for only a fraction of its performance overhead. We applied our technique to the Linux kernel and showed that Elastic Instrumentation is as efficient as full instrumentation at protecting against recent high-impact vulnerabilities but only incurs a quarter of the overhead.

References

[1] Amazon s3 availability event: July 20, 2008. Retrieved on 2013-07-20. http://status.aws.amazon.com/s3-20080720.html.

[2] Cve 2000-0461 report. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0461.

[3] Cve 2013-1763 exploit. http://www.exploit-db.com/exploits/24555.

[4] Cve 2013-1763 report. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1763.

[5] Cve 2013-2094 exploit. http://timetobleed.com/a-closer-look-at-a-recent-privilege-escalation-bug-in-linux-cve-2013-2094.

[6] Cve 2013-2094 report. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2094.

[7] Cve 2014-0038 exploit. http://blog.includesecurity.com/2014/03/exploit-CVE-2014-0038-x32-recvmmsg-kernel-vulnerablity.html.

[8] Cve 2014-0038 report. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0038.

[9] Logrhythm log analytics. https://www.logrhythm.com/logrhythm-in-action/operations/log-analysis.aspx.

[10] Piwik log analytics. http://piwik.org/log-analytics/.

[11] Random fault injection in linux kernel. http://lwn.net/Articles/209292/.

[12] Splunk log management. http://www.splunk.com/view/log-management/SP-CAAAC6F.

[13] Suse apparmor. https://www.suse.com/support/security/apparmor/.

[14] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE Symp. on Security and Privacy (S&P)*, 2008.

[15] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Annual Technical Conf. (USENIX)*, 2009.

[16] Amazon EC2. http://aws.amazon.com/ec2.

[17] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *Intl. Conf. on Dependable Systems and Networks (DSN)*, 2008.

[18] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. 2008.

[19] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)*, 2001.

[20] P. Arumuga Nainar and B. Liblit. Adaptive bug isolation. In *Intl. Conf. on Software Engineering (ICSE)*, 2010.

[21] L. N. Bairavasundaram, G. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An analysis of data corruption in the storage stack. 2008.

[22] E. Bisolfati, P. D. Marinescu, and G. Candea. Studying application–library interaction and behavior with LibTrac. In *Intl. Conf. on Dependable Systems and Networks (DSN)*, 2010.

[23] P. E. Black, editor. *Dictionary of Algorithms and Data Structures*, chapter Manhattan distance. U.S. National Institute of Standards and Technology, 2006. http://www.nist.gov/dads/HTML/manhattanDistance.html.

[24] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional detection of data races. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)*, 2010.

[25] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *ACM EuroSys European Conf. on Computer Systems (EUROSYS)*, 2011.

[26] J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song. Input Generation via Decomposition and Re-Stitching: Finding Bugs in Malware. In *Proceedings of the 17th ACM Conference on Computer and Communication Security*, Chicago, IL, October 2010.

[27] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Sys. Design and Implementation (OSDI)*, 2008.

[28] G. Candea, S. Bucur, and C. Zamfir. Automated software testing as a service. In *Symp. on Cloud Computing (SOCC)*, 2010.

[29] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Symp. on Operating Sys. Design and Implementation (OSDI)*, 1999.

[30] R. N. Charette. This Car Runs on Code. *IEEE Spectrum*, 2009.

[31] V. Chipounov, V. Kuznetsov, and G. 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 (ASPLOS)*, 2011.

[32] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)*, 30(1), 2012. Special issue: Best papers of ASPLOS.

[33] Clang User's Manual. Undefined behavior sanitizer. http://clang.llvm.org/docs/UsersManual.html.

[34] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *JACM*, 50(5):752–794, 2003.

[35] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2009.

[36] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. Subdomain: Parsimonious server security. In *Proceedings of the 14th USENIX Conference on System Administration*, 2000.

[37] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. 2008.

[38] M. Desnoyers. Using the Linux Kernel Tracepoints. https://www.kernel.org/doc/Documentation/trace/tracepoints.txt.

[39] D. Dhurjati, S. Kowshik, and V. Adve. Safecode: enforcing alias analysis for weakly typed languages. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)*, 2006.

[40] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM Symp. on Operating Systems Principles (SOSP)*, 2001.

[41] H. Etoh and K. Yoda. Gcc extension for protecting applications from stack-smashing attacks, 2000.

[42] File service protocol. Retrieved on 24-Jul-2013. http://fsp.sourceforge.net/.

[43] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Intl. Conf. on Computer Aided Verification (CAV)*, 2007.

[44] C. Giuffrida, A. Kuijsten, and A. Tanenbaum. Edfi: A dependable fault injection tool for dependability benchmarking experiments. In *IEEE 19th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2013.

[45] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *ACM Symp. on Operating Systems Principles (SOSP)*, 2009.

[46] P. Godefroid. Compositional dynamic test generation. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 2007.

[47] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symp. (NDSS)*, 2008.

[48] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, Jan. 2012.

[49] R. Guerraoui and M. Yabandeh. Model checking a networked system without the network. In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2011.

[50] H. Gunawi, T. Do, P. Joshi, P. Alvaro, J. Hellerstein, A. Arpaci-Dusseau, R. Arpaci-Dusseau, K. Sen, and D. Borthakur. Fate and destini: A framework for cloud recovery testing. In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2011.

[51] Hadoop Fault Injection framework, 2010. http://hadoop.apache.org/hdfs/docs/r0.21.0/faultinject_framework.html.

[52] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. *SIGPLAN Not.*, 2012.

[53] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Symp. on Operating Sys. Design and Implementation (OSDI)*, 2014.

[54] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into vhdl models: the mefisto tool. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, 1994.

[55] R. W. Jones and P. H. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Intl. Workshop on Automated Debugging (AADEBUG)*, 1997.

[56] P. Joshi, M. Ganai, G. Balakrishnan, A. Gupta, and N. Papakonstantinou. Setsudo: Perturbation-based testing framework for scalable distributed systems. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, 2013.

[57] X. Ju, L. Soares, K. G. Shin, K. D. Ryu, and D. Da Silva. On fault resilience of openstack. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, 2013.

[58] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2), 1995.

[59] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *Intl. Conf. on Dependable Systems and Networks (DSN)*, 2008.

[60] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2007.

[61] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 1976.

[62] D. E. Knuth. Structured programming with go to statements. *ACM Computing Surveys*, 1974.

[63] K. Krishnan. Weathering the unexpected. *Queue*, 10(9):30:30–30:37.

[64] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza. Attack surface metrics and automated compile-time os kernel tailoring. In *Network and Distributed System Security Symp. (NDSS)*, 2013.

[65] A. Kurmus and R. Zippel. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *ACM Conf. on Computer and Communications Security (CCS)*, 2014.

[66] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)*, 2012.

[67] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A soft way for openflow switch interoperability testing. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, 2012.

[68] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[69] B. W. Lampson. Hints for computer systems design. *ACM Operating Systems Review*, 15(5):33–48, 1983.

[70] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2014.

[71] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics – Doklady*, 10, 1966.

[72] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[73] P. D. Marinescu, R. Banabic, and G. Candea. An extensible technique for high-precision testing of recovery code. In *USENIX Annual Technical Conf. (USENIX)*, 2010.

[74] P. D. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In *Intl. Conf. on Dependable Systems and Networks (DSN)*, 2009.

[75] P. D. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Transactions on Computer Systems (TOCS)*, 29(4), Dec. 2011.

[76] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)*, 2009.

[77] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*.

John Wiley & Sons, Inc., 1990.

[78] M. Maurer and D. Brumley. Tachyon: tandem execution for efficient live patch testing. 2012.

[79] P. McMinn. Search-based software test data generation: A survey. 14:105–156, 2004.

[80] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller. Mining trends of library usage. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, 2009.

[81] Crash due to missing errmsg.sys. http://bugs.mysql.com/bug.php?id=25097, 2006.

[82] Crash due to double unlock. http://bugs.mysql.com/bug.php?id=53268, 2010.

[83] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Intl. Conf. on Software Engineering (ICSE)*, 2006.

[84] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: Compiler enforced temporal safety for C. In *Intl. Symp. on Memory Management (ISMM)*, 2010.

[85] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *POPL*, 2002.

[86] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *ACM Conf. on Computer and Communications Security (CCS)*, 2007.

[87] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein. Analyzing protocol implementations for interoperability. In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2015.

[88] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *ACM SIGSOFT Intl. Symp. on the Foundations of Software Engineering (FSE)*, 2008.

[89] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

[90] J. Samson, J.R., W. Moreno, and F. Falquez. A technique for automated validation of fault tolerant designs using laser fault injection (lfi). In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, 1998.

[91] R. Sasnauskas, P. Kaiser, R. L. Jukic, and K. Wehrle. Integration testing of protocol implementations using symbolic distributed execution. In *Proceedings of the 2012 20th IEEE International Conference on Network Protocols (ICNP)*, 2012.

[92] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer - Data race detection in practice. 2009.

[93] J. Song, C. Cadar, and P. Pietzuch. Symbexnet: Testing network protocol implementations with symbolic execution and rule-based specifications. *Software Engineering, IEEE Transactions on*, 2014.

[94] J. L. Steffen. Adding run-time checking to the portable C compiler. *Software: Practice and Experience*, 1992.

[95] M. Sullivan and R. Chillarege. Software defects and their impact on system availability – a study of field failures in operating systems. 1991.

[96] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *IEEE Symp. on Security and Privacy (S&P)*, 2013.

[97] L. Tahat, B. Vaysburg, B. Korel, A. Bader, L. Technol, and I. Naperville. Requirement-based automated black-box test generation. *Computer Software and Applications Confer-*

ence, 2001.

[98] Library of test and utility APIs, 2010. http://testapi.codeplex.com/.

[99] T. K. Tsai and R. K. Iyer. Measuring fault tolerance with the FTAPE fault injection tool. In *Intl. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, 1995.

[100] A. Turing. Checking a large routine. In *The early British computer conferences*, 1989.

[101] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder. High system-code security with low overhead. In *IEEE Symp. on Security and Privacy (S&P)*, 2015.

[102] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Symp. on Operating Sys. Design and Implementation (OSDI)*, 2002.

[103] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Intl. Conf. on Dependable Systems and Networks (DSN)*, 2009.

[104] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MoDist: Transparent model checking of unmodified distributed systems. In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2009.

[105] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[106] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving Software Diagnosability via Log Enhancement. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[107] C. Zamfir and G. Candea. Execution synthesis: A technique for automated debugging. In *ACM EuroSys European Conf. on Computer Systems (EUROSYS)*, 2010.