FIXING SOFTWARE VULNERABILITIES AND CONFIGURATION ERRORS

by

Zhen Huang

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Fixing Software Vulnerabilities and Configuration Errors

Zhen Huang

Doctor of Philosophy

Graduate Department of Electrical and Computer Engineering

University of Toronto

2018

With the rise of mobile devices such as smart phones and IoTs and emerging new application areas such as fitness and sport aid, smart home, and augmented reality, computer systems have become a critical part of our daily lives. Our reliance on computer systems make software security and reliability extremely important. However, software security and reliability are threatened by software vulnerabilities and configuration errors.

Manually fixing software vulnerabilities and configuration errors is a tedious and time-consuming task. Automating the task has gained intense interest. This dissertation addresses three challenges in automating the task: 1) mitigating software vulnerabilities rapidly and safely, 2) generating sound security patches and 3) troubleshooting complex configuration errors that involve dependent configuration settings. We make the following contributions.

First, we consider mitigating software vulnerabilities. Inspired by configuration workarounds, a fast alternative of security patches, we design Security Workaround for Rapid Response (SWRR) that works similarly to configuration workaround but has substantially larger coverage than configuration workarounds. We implement a prototype Talos that automatically produces SWRRs and instruments SWRRs into applications. SWRRs generated by Talos can cover 2.1× software vulnerabilities than configuration workarounds.

Second, we consider generating sound security patches. With a design specifically targeting three of the most common and severe software vulnerabilities: buffer overflow, bad offset, and integer overflow, we combine program analysis techniques to generate semantically correct security patches. Our prototype implementation called Senx successfully generates correct security patches for 76.2% of 42 real-world software vulnerabilities.

Third, we compare the strengths and drawbacks of Talos and Senx qualitatively and quantitatively. On one hand, Senx has the strength in applicability. On the other hand, Talos has the strength in scalability and usability. We find that Talos and Senx have complementary applicability. Combining them, we can address 90.5% of the 42 software vulnerabilities.

Finally, we consider troubleshooting and fixing configuration errors involving dependent configuration settings. We leverage unsupervised machine learning to understand the dependency among configuration settings and use automated GUI testing to enable regular users to troubleshoot and fix configuration errors with ease. We implement a prototype called Ocasta and conduct a user study on Ocasta. We find that Ocasta can correctly identify 88.6% of dependent configuration settings and significantly save user time and effort in troubleshooting and fixing configuration errors.

Acknowledgements

First and foremost, I would like to express my gratitude to Professor David Lie for his patient supervision, relentless enthusiasm, financial support, and large amount of time spent on discussions for this research. It has truly been a privilege to work with him.

I would also like to thank my thesis committee members, Professor Ashvin Goel and Professor Ding Yuan for their valuable feedback on the work presented in this thesis.

I am very grateful to my family and my parents for their much needed moral support during my research career.

Finally, I would like to thank University of Toronto and the department of Electrical and Computer Engineering for their financial support.

Bibliographical Notes

Chapter 3 and Chapter 4 present work published in IEEE S&P 2016 [62]. The work presented in Chapter 5 is published as an arXiv preprint in 2017 [64]. Chapter 7 presents work published in DSN 2014 [63].

Contents

List of Tables

List of Figures

Chapter 1

Introduction

To date, computer systems have evolved into an indispensable tool to our daily lives. Indeed, almost all aspects of our daily lives such as work, education, communication, entertainment, health and fitness depend on computer systems. Our highly reliance on computer systems makes software security and reliability of paramount importance. Nowadays it is not uncommon for a single security breach or service outage to cause massive personal and corporate loss. Equifax, one of the three largest credit agencies in the U.S, suffered a security breach in 2017 that leaked highly sensitive privacy information such as social security numbers and driver's license numbers of 145.5 million consumers [35]. Also in 2017, Comcast Internet, the largest home Internet service provider in the U.S, had a national wide outage of their Internet service for around 90 minutes due to a configuration error by an enterprise ISP that provides the backbone for other Internet providers [34].

When software security or reliability issues are discovered in computer systems, it is pressing to troubleshoot and fix them to avoid further personal or corporate loss. However, troubleshooting and fixing them is usually a tedious and time-consuming task for software developers or system administrators. To remedy this situation, many techniques have been proposed to aid or automate this task [43, 70, 82, 83, 89, 125, 129–131, 141]. Unfortunately, they have various limitations and are rarely adopted in practice. The best-practice methods still largely rely on the manual effort of software developers and system administrators.

This dissertation focuses on the topic of the automation of troubleshooting and fixing software security and reliability issues. We propose approaches that automatically generate workarounds and patches for software vulnerabilities, and automatically troubleshoot complex configuration errors. We show that our approaches are efficient and applicable to a variety of software vulnerabilities and configuration errors.

1.1 Software Vulnerabilities

Software vulnerabilities is a pressing issue for software security. They play an essential role in the security of computer systems. Indeed, they are software bugs that can be exploited to mount security attacks on computer systems. Since the very early days of Internet, they have been frequently leveraged to comprise computer systems [37]. The recent wide propagation of WannaCry ransomware is also largely attributed to a vulnerability in the Microsoft Windows systems [91].

As a result, it is critical to patch vulnerabilities immediately after they are discovered, in order to prevent attackers from exploiting them to subvert computer systems. However, it commonly takes as long as tens to hundreds of days to release patches to fix software vulnerabilities [80]. To understand the bottleneck of releasing security patches, we study the life cycle of software vulnerabilities and the complexity of security patches. We find that the bottleneck of releasing patches to software vulnerabilities often lies in developing such patches. Similar to other types of software bugs, it is not uncommon for software developers to make even several attempted patches before the correct patch for a software vulnerability [6]. We present this study in Chapter 3. To remedy the delays in releasing patches, a faster alternative called configuration workaround is commonly used to mitigate software vulnerabilities.

1.1.1 From Configuration Workarounds to SWRRs

Configuration workarounds leverage existing configuration options in applications to disable code relevant to software vulnerabilities in order to mitigate them. Because configuration options are readily present in applications, configuration workarounds can be used by users in production without the need for patches. Thereby some large software vendors such as Microsoft and VMWare routinely publish configuration workarounds for newly discovered vulnerabilities to allow users to immediately mitigate the vulnerabilities before software vendors are able to release the patches [31–33, 36].

The security provided by configuration workarounds does not come for free. When they are used, the functionality provided by the code disabled by them are lost regardless whether the inputs to the software will actually trigger the software vulnerabilities. Therefore they are usually used only when the lost functionality is not critical.

However, configuration workarounds can be used to protect only a fraction of software vulnerabilities. Our study shows that only 25.2% of vulnerabilities have configuration workarounds [62]. This is probably because configuration options are usually designed for giving users the flexibility to adapt the functionality of a software to different usages rather than for mitigating software vulnerabilities.

Inspired by the idea of configuration workarounds, we introduce the concept of Security Workarounds for Rapid Response (SWRR). SWRRs mitigate software vulnerabilities in a manner similar to configuration workarounds by disabling executions of vulnerable code. And they can be instrumented into applications before the release of the applications so that they can be activated by users in production just like configuration workarounds. By design, SWRRs can be automatically synthesized and instrumented into applications and they protect aginst more than 2.1x of vulnerabilities than configuration workarounds. We present the design, implementation, and evaluation of SWRRs in Chapter 4.

1.1.2 Security Patch Generation

Automatic Patch Generation (APR) aims to automate the process of generating software patches in order to alleviate the burden on software developers. A large number of APR techniques targeting generic bugs have been proposed in recent years [67, 68, 70, 82, 83, 88, 89, 129, 131].

APR techniques can be broadly categorized as search-based and semantic-based. Search-based techniques search patch candidates within a search space and validate the correctness of the patch candidates with test cases [70, 82, 129]. They can make arbitrary code changes and are applicable to a wide range of bugs. On the contrary, semantic-based techniques leverage semantic information collected from program analysis and follow rules to produce patches [88, 89, 95]. They tend to produce correct patches but are less applicable because they are constrained by the applicability of the program analysis they use.

Unlike patches for generic software bugs, the patches for software vulnerabilities require more strict guarantees on correctness and security. However, the vast majority of APR techniques heavily depend on the comprehensiveness of test cases to generate correct patches, and hence they generate incorrect patches most of the time due to the incompleteness of test cases or error-prone approaches to validating patch candidates [102]. As a result, they are ill-suited for software vulnerabilities.

To address this limitation, we propose a novel approach, called Security Patch Generation, that is specifically designed to generate patches for software vulnerabilities, i.e. security patches. Our approach uses program analysis techniques to produce patches for three of the most common types of software vulnerabilities: buffer overflow, bad offset, and integer overflow. We specifically design program analysis techniques targeting these three types of software vulnerabilities and thereby our design achieves semantically soundness and high applicability. Particularly our design deals with software vulnerabilities involving complex code and data structures. We describe the details of this approach in Chapter 5.

1.1.3 SWRRs v.s. Security Patch Generation

SWRRs and Security Patch Generation address software vulnerabilities with different design goals. SWRRs mitigate vulnerabilities at the cost of functionality loss, so they are intended to be a short-term solution. In contrast, Security Patch Generation produces security patches to fix vulnerabilities without any loss of functionality, which are intended to be a long-term solution.

Because of their different design goals, SWRRs and Security Patch Generation have different applicability to software vulnerabilities. On one hand, SWRRs can work with any type of software as long as there exist error-handling code for them to gracefully abort the execution of vulnerable code, but they can cause obtrusiveness, i.e. major functionality loss, if the vulnerable code happen to implement critical functionality.

On the other hand, Security Patch Generation works only with the software vulnerability types that it is designed to work with, but the patches that it generates will not cause obtrusiveness because they are designed to preserve correct functionality. Unfortunately its applicability to software vulnerabilities is further limited by the applicability of the program analysis techniques it employs.

To understand further the different applicability of SWRRs and Security Patch Generation, we conduct experiments to evaluate their applicability on th same set of real-world software vulnerabilities. We present and evaluate the evaluation results in Chapter 6.

1.2 Configuration Errors

Configuration errors is one of the major causes of software reliability and security issues. Indeed, they are a leading cause of computer system failures and unavailability [55]. Recent studies have shown that the prevalence of configuration errors is mainly due to the increasing complexity of configurations [132] and error-prone configuration design and handling [133].

There has been a large body of work on troubleshooting configuration errors [43, 75, 125, 126, 130]. Using rollback-recovery to diagnose configuration errors is one of the pioneering approaches [130]. It keeps a history of previous configuration states and searches backward in the history to find a working configuration state. But existing rollback-recovery approaches can fail in many situations. First, they cannot fix configuration errors involving dependent configuration settings, where the dependency among configuration settings must be maintained, because they do not take into account the dependency among configuration settings. Second, they can cause unexpected changes to configuration settings irrelevant to configuration errors because they do not distinguish individual configuration settings. And they are only available to expert users because they rely on scripts or programs written by users to determine whether configuration errors are fixed.

Other approaches consider the state of individual configuration settings and uses statistical anomaly detection to identify abnormal configuration settings [75, 125, 126] or white-box dynamic analysis to link erroneous code path to particular configuration settings [43, 141]. However, none of them takes into account the dependency among configuration settings and thus they cannot fix configuration errors involving only one single configuration setting. Unfortunately a significant number of configuration errors (14.9% - 34.7%) can be fixed only by changing all dependent configuration settings together [135].

To tackle configuration errors involving dependent configuration settings, we propose an original approach that leverages unsupervised machine learning to understand the dependency among configuration settings and uses rollback recovery that considers a set of dependent configuration settings as a unit for rollback. It also makes the rollback recovery usable for regular users by employing automated GUI testing facility to remove the burden on users in writing probes. In Chapter 7, we present the details of this approach.

1.3 Dissertation Outline

This dissertation addresses the challenges of the automation of troubleshooting and fixing software security and reliability issues. For software security, we focus on mitigating and fixing software vulnerabilities. For software reliability, we focus on troubleshooting and fixing configuration errors. We illustrate that a novel combination of program analysis techniques can automatically mitigate or fix software vulnerabilities, and unsupervised machine learning coupled with automated GUI testing can automatically fix configuration errors.

In Chapter 2, we discuss related work. We then present our study on the lifecycle and complexity of security patches in Chapter 3, which motivates our work on mitigating and fixing vulnerabilities.

As we have mentioned, Chapter 4 and Chapter 5 respectively present Talos and Senx, two approaches that address software vulnerabilities, followed by Chapter 6 that compares the strengths and drawbacks of Talos and Senx. Chapter 7 then presents Ocasta, our approach that fixes configuration errors. Finally, we conclude and propose directions for future work in Chapter 8.

Chapter 2

Related Work

In this chapter, we discuss previous work related to my thesis. We first discuss work on software vulnerabilities, which is related to my work on mitigating and fixing vulnerabilities. We then discuss work on configuration errors, which is related to my work on troubleshooting and fixing configuration errors. After that, we discuss work on failure recovery, because both my work on vulnerabilities and my work on configuration errors utilize failure recovery techniques. Finally we discuss work on machine learning, because my work on configuration errors leverages unsupervised machine learning.

2.1 Software Vulnerabilities

This section discusses work related to Talos and Senx, the systems that I built to mitigate and fix vulnerabilities, respectively. We first discuss work on characterizing vulnerabilities, because we believe that a good understanding of vulnerabilities is the first step towards addressing vulnerabilities. We then discuss work on mitigating vulnerabilities, which is related to Talos. Finally we discuss work on automatic patch generation, which is related to Senx.

2.1.1 Characterizing Software Vulnerabilities

We consider characterizing vulnerabilities as the first step towards addressing vulnerabilities. Characterizing vulnerabilities allows us to understand how existing best practice methods address vulnerabilities and what major challenges they face and what major limitations they have. To answer these questions, we have conducted a study on the lifecycle of vulnerabilities. We describe this study in Chapter 4. This section discusses other studies on vulnerabilities.

Since software vulnerabilities are such a major source of security threats, there have been a lot of studies on characterizing and understanding software vulnerabilities to determine indicators that might predict their presence [39, 94, 112]. They measured characteristics such as vulnerability density, defect density, vulnerability discovery rate, structural complexity of code, code churns, and developer activities on code; and built models based on the relationships between them. We mention these as they served as an inspiration for us to study configuration workarounds to mitigate such vulnerabilities.

A recent study on the life cycle of software releases [50] indicates that the rapid-release methodology used by Mozilla Firefox does not increase the ratio of vulnerabilities in the code, somewhat contrary to the popular belief that frequent code changes result in less secure software. Another study has measured characteristics such as evolution of vulnerabilities over the years, impacts of vulnerabilities, and access required for exploits over vulnerabilities and their implications on software design, development, deployment, and management [111].

2.1.2 Mitigating Vulnerabilities

This section discusses work related to Talos, the system that I built to mitigate vulnerabilities. We first discuss techniques that rectify or filter malicious inputs, because they have the same goal as Talos of preventing the triggering of vulnerabilities. We then discuss techniques that detect and prevent the execution of exploits. Finally we discuss techniques that try to find configuration workarounds from existing configuration space to avoid failures.

Rectifying Inputs. Talos prevents vulnerabilities from being triggered in the first place. With the same goal, some techniques rectify malicious program inputs so that they cannot trigger vulnerabilities. With taint analysis, SOAP learns constraints on input by observing program executions with benign inputs [81]. From the constraints that it has learnt, it identifies input that violates the constraints and tries to change the input to make it satisfy the constraints. By doing so, it not only renders the input harmless but also allows the desired data in the rectified input to be correctly processed

A2C exploits the observation that exploit code embedded in inputs is often fragile to any slight changes. By encoding inputs with an one-time dictionary and decoding them only when the program execution goes beyond the often vulnerable code, it disables the embedded exploit code and turns it into a program termination [74].

Filtering inputs. An alternative to prevent the triggering of vulnerabilities is to detect and filter malicious inputs. In general, these techniques perform analysis of the application source code to generate a vulnerability-specific input filter that will detect inputs that could reach the specified vulnerability. Some proposals detect and drop such inputs [51, 84, 117, 124], while others convert malicious inputs into benign inputs [81, 107].

For example, Bouncer uses static analysis, combined with dynamic symbolic analysis, on programs to infer the conditions that inputs must satisfy to exploit a vulnerability and then craft filters based on these conditions [51]. HEALERS protects library functions by generating wrappers on them, which intercept malicious inputs and return an error condition instead of executing the vulnerable function [117]. It uses static analysis-guided fault injection to infer predicates on the input arguments to a function that can cause the function to crash. HEALERS only works on libraries with a well defined error specification in their API. In contrast, Talos works on arbitrary internal functions in an application, and thus must infer error paths and values since it does not assume they are specified. Shields [124] uses statically extracted information to generate network filters, which then drop the network packets that might potentially trigger a vulnerability.

Preventing Exploits from Hijacking Program Execution. Rather than preventing the triggering of vulnerabilities, some systems are proposed to prevent exploits to hijack program execution after vulnerabilities are triggered. For example, Software Fault Isolation (SFI) [123] and similar techniques [93, 134] instrument memory operations with bounds checks to make sure even erroneous code cannot corrupt memory. Another approach is to validate every control flow transfer with Control Flow Integrity (CFI) [38, 52, 96, 97, 121, 139, 140]. Comparing to the code instrumentation used by Talos, the code instrumentation used by these systems is either more complex in the case of CFI or needs to be called more frequently in the case of SFI. As a result, these hardening approaches generally have a higher performance overhead.

The major difference between Talos and these approaches is that they all require some malicious input, i.e. proof-of-concept exploit, that can trigger a vulnerability. However, we find that most of times a proof-of-concept exploit is not publicly available for a disclosed vulnerability, probably due to security concerns. On the contrary, Talos does not require a proof-of-concept exploit and requires only the name of a vulnerable function, which is usually publicly available in vulnerability databases.

Finding existing workarounds for failures. A recently emerging area is searching the configuration space of an application for workarounds for a specific failure. REFRACT searches for configuration workaround for program failures [118]. Given a model of the configuration space of a program and strategies to avoid failures, REFRACT tries to find a configuration workaround that can avoid the failures caused by malicious inputs, by repeatedly replaying inputs that trigger the failures to the program using different sample configurations. For 6 of 7 Firefox bugs, it successfully found configuration workarounds. Unlike Talos, REFRACT relies on the existing configuration space of a program to workaround a vulnerability. However, our findings show that configurations often do not provide sufficient coverage to workaround most security vulnerabilities. Talos avoids this limitation by instrumenting a program with SWRRs, which are designed specifically to protect the program from being exploited.

2.1.3 Automatic Patch Generation

This section discusses work related to Senx, the system that I built to generate security patches that fix vulnerabilities. There are three major directions in generating patches: leveraging fix patterns, using program mutations, and learning from correct code. We first discuss work in these major directions. We then discuss work that uses symbolic execution and constraint solvers. Finally we discuss work on honey-patch, which is generated from regular patch but serves the purpose of misleading attackers.

Leveraging fix patterns. Similar to Senx, some automatic patch generation (APR) techniques also leverage fix patterns or models to generate patches.

By observing common human-developer generated patches, PAR generates patches using fix patterns such as altering method parameters, adding a null checker, calling another method with the same arguments, and adding an array bound checker [70]. PAR is able to generate patches for 23% of the 119 bugs from six Java projects. Senx differs from PAR in two aspects. First, PAR is unable to generate a patch when the correct variables or methods needed to synthesize a patch are not accessible at the faulty function or method. Second, PAR uses a trial-and-error approach that tries out not only each fixing pattern upon a given bug, but also variables or methods that are accessible at the faulty function or method to synthesize a patch. On the contrary, Senx employs a guided approach that identifies the type of the given bug and chooses a corresponding patch model to generate the patch for the bug and systematically finds the correct variables to synthesize the patch based on semantic information provided by a patch model.

Focusing on memory leak bugs, LeakFix defines a fix as the only deallocation statement for a memory chunk which must be executed after the allocation statement of the memory chunk and after any use of the memory chunk [57]. By labelling program statements related to memory allocation, deallocation, and usage, and abstracting a program into a CFG containing only those related program statements, LeakFix transforms the problem of finding a fix for a leaked memory chunk into searching for an edge where a pointer expression always points to the memory chunk can be constructed, no execution path covering the edge has a dealloction statement for the memory chunk, and no use of the memory chunk exists on the outgoing path of the edge. LeakFix successfully generates patches for 28% of the 89 reported leaks in the SPEC2000 benchmark.

SPR fixes a defect with transformation schemes [82]. For a defect, it identifies a statement in a target program as a repairing target, then selects a transformation schema, from a list of transformation schemes, to modify the statement by associating an abstract function with it. After that, it repeatedly runs the modified program to discover the correct values that this abstract function should return for both inputs triggering the defect and inputs not triggering the defect. By recording the values of all variables accessible at the identified statement, it tries to find an expression involving a subset of the variables to act as the abstract function. This expression is then used to synthesize a patch.

Using program mutations. GenProg is a pioneering work that induces program mutations, i.e. genetic programming, to generate patches [129]. Leveraging test suites, it focuses on program code that is executed for negative test cases but not for positive test cases and utilizes program mutations to produce modifications to a program. As a feedback to its program mutation algorithm, it considers the weighted sum of the positive test cases and negative test cases that the modified program passes. Treating all the results of program mutations as a search space, its successor improves the scalability by changing to use patches instead of abstract syntax trees to represent modifications and exploiting search space parallelism [78].

Based on an analysis on the patches generated by state-of-the-art generate-and-validate APRs, including GenProg [129], RSRepair [101], and AE [128], Kali generates patches that only delete functionality [102]. The analysis finds that the test suites used by those patch generators to determine whether a patch correctly fixes a bug often fail to do so because the test suites do not even check whether the patched program produces correct output. By augmenting the test suites with checks on program output, they find that the vast majority of the patches claimed to be correct by these patch generators are actually incorrect. This indicates that relying merely on test suites to verify the correctness of automatically generated patches can cause misleading results.

Learning from correct code. Prophet learns from existing correct patches [83]. It uses a parameterized log-linear probabilistic model on two features extracted from the abstract trees of each patch: 1) the way the patch modifies the original program and 2) the relationships between how the values accessed by the patch are used by the original program and by the patched program. With the probabilistic model, it ranks candidate patches that it generated for a defect by the probabilities of their correctness. Finally it uses test suites to test correctness of the candidate patches. Like other generate-and-validate APRs, its effectiveness depends on the quality of the test suites.

Unlike Prophet, CodePhage intends to borrow code from the binary code of a donor program and translate it into a source code patch for a defect in a target program [115]. A donor program is a program that reads the same inputs as the target program, and correctly handles both the input that can trigger the defect in the target program and the input that do not trigger the defect. CodePhage searches for a donor program from its list of donor programs. After finding a donor program for the defect, CodePhage searches for a check in the donor program that returns opposite values for the two inputs and considers it as a candidate check. It then runs the donor program to produce expressions that denote the check as a function that determines the values of some input fields. Finally it searches for locations in the target program where the expressions can be translated to valid source code as a patch for the target program to check against the same input fields.

The way that CodePhage borrows code from one program and translates the code into another program is analogous to the expression translation used by Senx, which instead translates between different scopes of one program.

Using constraint solvers. SemFix uses constraint solving to find the needed expression to repair the right hand side of an assignment statement or the condition checked by an if statement [95]. By executing a target program symbolically with both inputs triggering a defect and inputs not triggering the defect, it identifies the constraints that the target program must satisfy to process both kinds of inputs correctly. It then synthesizes a patch using component-based program synthesis, which combines components such as variables, constants, and arithmetic operations to synthesize an expression that can make the target program satisfy the identified constraints.

Similar to SemFix, Angelix runs the target program symbolically with concrete inputs to discover the constraints that a target program should satisfy to fix a defect and then uses component-based program synthesis to generate a patch [89]. Different from SemFix, it considers more than one statements in a target program so that it is capable of generating a patch that modifies more than one statements in the target program.

Honey-patch. On top of fixing vulnerabilities, a recent work aims to misinform attackers about whether an exploit works or not by transforming a regular patch into a honey-patch, which adds additional logic to redirect malicious inputs to a vulnerable version of a program, so that the exploit targeting the patched vulnerability appears to be successful to attackers [42].

2.2 Configuration Errors

This section discusses work related to Ocasta, the system that I built to troubleshoot and fix configuration errors. We first discuss work on inferring related configuration settings, because Ocasta aims to address configuration errors involving related configuration settings. We then discuss work on diagnosing configuration errors, which shares the same goal as Ocasta. Finally we discuss work on detecting and simulating configuration errors.

2.2.1 Inferring Related Configuration Settings

Few previous studies automatically infer relations among configuration settings. Zheng et al. [142] deduce dependency among configuration settings by experimentally testing the impact of changing configuration settings. Ocasta's clustering algorithm avoids the overhead of experimental tests by using observed application accesses to configuration settings. Glean [75] infers relations among configuration settings by analyzing hierarchical structure of configuration settings, while Ocasta's clustering algorithm does not require the existence of hierarchical structure for configuration settings.

2.2.2 Diagnosing Configuration Errors

There has been a large body of work on diagnosing configuration errors. In this section, we first discuss techniques that identify erroneous configurations with statistics. We then discuss techniques that leverage program analysis to find the cause of configuration errors, after which we discuss techniques that learn from human-generated solutions. Finally we discuss search-based techniques.

Identifying errors with statistics. Of the work that focuses on diagnosing configuration errors, Ocasta is most closely related to Strider [126] and PeerPressure [125]. Both PeerPressure and Strider use a genebank of common configurations and apply statistical methods to determine where the error might lie. These systems assume homogeneity across machines and also have privacy implications as users must share their configurations with the genebank. Ocasta only requires information collected locally from the machine with the error and thus does not have the drawbacks of a genebank.

Leveraging program analysis. ConfAid [43] takes a "white-box" approach by using taint-analysis to try to identify the configuration setting that causes a configuration error. ConfAid ranks configuration settings that affect the path taken to reach the configuration error as more likely to be configuration keys that can fix the error. Another "white-box" approach, Failure-Context-Sensitive analysis [104] extracts the mapping between configuration settings and the source code lines that can be affected by these configuration settings, from the source code of an application. These mappings can be used to identify the configuration setting that causes configuration errors, when the source code lines of the errors are available, for example from an application's error message. More recent work, ConfDiagnoser, combines static analysis of an application's source code and execution profiling to rank configuration settings that causes executions to deviate from pre-generated correct executions [140]. Because these approaches are white-box, they require application source code. In contrast, Ocasta treats applications as black-boxes and only requires access to the application's key-value store.

Learning from human-generated solutions. Kardo [73] and Autobash [116] are both systems that take a human-generated solution for a configuration error, perform analysis on the solution to find the minimum set of actions that make up the configuration fix and generalize it so it can be applied to a wider set of machines. Ocasta does not require human-generate solutions.

All above work focuses on identifying a single configuration setting that causes configuration errors. With the clustering provided by Ocasta, their techniques can be leveraged to diagnose configuration errors caused by more than one configuration settings.

Search-based solutions. Chronus [130] maintains a history of entire system states and focuses on using binary search to find the optimal recovery point in an application's history. Chronus logs at the disk block layer and as a result, many of the historical states it generates can corrupt file systems and thus cannot be used for recovery.

2.2.3 Detecting and Simulating Configuration Errors

Like Ocasta, CODE [137] analyzes the accesses patterns that applications make to the Windows registry. CODE uses a rule learning algorithm to identify normal key access patterns of an application and flags anomalous access patterns as possible configuration errors. CODE detects configuration errors, but unlike Ocasta, it does not fix the errors, nor does it try to identify relationships between keys other than the access patterns.

Conferr is a tool for quantifying system manageability and resilience to configuration errors [47, 69]. It uses simulated human models to try to generate realistic configuration errors. Both CODE and Conferr can be viewed as complementary to Ocasta.

2.3 Failure Recovery

Both my work on software vulnerabilities and my work on configuration errors leverage failure recovery techniques. This section discusses work on the failure recovery techniques that are related to the ones used by my work.

2.3.1 Resuming Execution After Faults

Some techniques try to improve fault tolerance by allowing an application to continue execution after a fault has occurred [49, 56, 85, 100, 106, 113]. In general, these do not have the same level of security as Talos as they cannot guarantee that the recovered application is secure, but they follow the same principle of detecting an erroneous application state and redirecting it to some non-erroneous state that Talos uses with SWRRs.

Failure-Oblivious Computing is proposed to improve the resilience of server applications after an attack has triggered memory errors, by augmenting an application to ignore memory errors [106]. For out-of-bounds memory writes, it simply discards them. For out-of-bounds memory reads, it redirects them to a preallocated buffer that contains pre-defined values that are likely to reduce the possibility of a crash or infinite loop. Recently, this work was followed by RCV, which further limits the propagation of the manufactured values within an application by skipping any system call that tries to use them [85]. Like Talos, these approaches seek to trigger error-handling code in the application. The main difference is that these approaches are simpler in that they guess the values that will cause this to happen, while Talos uses static analysis on the application source code to discover the location of error-handling code and the appropriate place to intercept and redirect execution to the error-handling code. Another difference is that these approaches focus on executing past out-of-bounds memory accesses, while Talos, which disables individual functions, can handle a broader set of software faults.

A technique has been proposed to abort the execution of a function when it overruns a memory buffer, as a consequence of malicious inputs, and resume the execution right after the call to the offending function after making a best effort to undo any side-effect caused by the offending function such as changing global variables [113]. Their evaluation also indicates that the program can continue run in many situations. The challenge of this work is that many times it is difficult if not impossible to infer what side-effects the partial execution of a function has caused and how to correctly undo them. Talos avoids this problem by not executing any part of a function and simply forcing the function to return an error code to its caller.

2.3.2 Time Travel and Roll Back

The concept of time travel and roll back has been used for debugging and system recovery from intrusions. Time-travel virtual machines [72] enables deterministic replay of whole machines to simplify OS debugging. Taser [59] and Retro [71] use system-level tracking and perform selective recovery after an intrusion. Rx [103] uses repeated roll backs to find an execution where bugs do not occur, but does not try to find the root cause or attempt to permanently fix the bug. Like Ocasta, these systems use roll back recovery but focus on fixing other types of faults while Ocasta focuses specifically on configuration errors.

2.4 Machine Learning

My work on troubleshooting and fixing configuration errors leverages machine learning to understand the dependency among configuration settings. In this section, we discuss work that leverages similar machine learning techniques.

2.4.1 Hierarchical Clustering

Many previous studies have used hierarchical clustering for software clustering [40, 108, 122], including program comprehension, reverse engineering, and software reengineering, cluster different levels of abstractions of software artifacts, such as variables, functions, and source files. Prior work has also used hierarchical agglomerative clustering to improve the efficiency of finding software failures during software testing [53] or categorizing software failures [54]. They cluster profiles of an application's executions.

Ocasta uses the maximum linkage criterion, which as been found by other prior work [41, 87] to provide better performance than other linkage criterion. Ocasta augments the hierarchical agglomerative algorithm to be able to partition clusters using an adjustable clustering threshold, which is more flexible and intuitive for our purposes of clustering configuration settings.

Chapter 3

Study on Security Patches

This chapter presents our study on security patches. We focus on the characteristics and cause of pre-patch window, the time between a vulnerability is discovered and the time a patch is issued. This study motivates our work on mitigating and fixing software vulnerabilities. In this chapter, we first study the lifecycle of security patches. We then study the complexity of security patches.

3.1 Lifecycle of Security Patches

We begin with a study of the lifecycle for recent security vulnerabilities. The vulnerabilities used in our study were collected from various sources, including common vulnerability databases [10, 12, 17, 25], vendor-specific security bulletins [4, 11, 16], and software bug

Table 3.1: Security patch statistics.

Application	**#Vulnerabilities**	**Delay (Days)**	**SLOC**	**#Functions**	**#Files**
lighttpd	27	54	49	2	2
apache	30	61	47	2	2
squid	46	73	64	6	3
proftpd	16	9	95	4	2
sqlite	12	62	17	4	3
AVERAGE	26	52	54	4	2

databases [9, 14, 23].

For our study, we need information on the disclosure date of vulnerabilities, the release/-commit date of patches, and the source code of the patches. Hence we choose open-source applications that are popular, reasonably complex, mature, being actively developed and maintained, and have a decent number of vulnerabilities. For each application, we selected as many vulnerabilities as possible that have the required information for manual examination. Our results are shown in Table 3.1. Column "#Vulnerabilities" shows the number of examined vulnerabilities. Column "Delay" shows the average number of days between the disclosure of security vulnerabilities and the release of corresponding software patches. We obtain the date when a vulnerability is disclosed from either an official vulnerability disclosure or the bug report for the vulnerability. For some vulnerabilities, we could not find an official disclosure date, so we approximate this using the earliest dated document in which they are referenced. From the collected data, we can see that it takes considerable time to release a patch and the size of the pre-patch vulnerability window is significant, averaging around 1.5 months after the vulnerability is disclosed.

We find that 43.4% of the vulnerabilities were patched within 7 days after the vulnerabilities were disclosed, 23.3% of them were patched between 7 days and 30 days, and 33.3% of them were patched after 30 days. Similarly, a recent study on the lifecycle of security vulnerabilities in operating systems and web browsers shows that among open source vendors, 65% of the vulnerabilities were patched within 7 days, 9% of them were patched between 7 days and 30 days, and 18% of them were patched after 30 days [111]. Both our study and their study indicate that a significant number of vulnerabilities were patched after 30 days.

To understand the bottleneck in releasing a patch, we break the task of releasing a patch into steps including *vulnerability triage*, *constructing a patch*, and *regression test*. We define constructing a patch as developers' effort in developing a patch after acquiring one or more test cases to trigger a vulnerability. In general, vulnerabilities are reported to software vendors with such test cases, particularly when they are discovered via fuzzing test. We study the time spent in each step by examining the bug report of vulnerabilities. Unfortunately, we were only able to locate the bug reports for 21 of the vulnerabilities that are shown in Table 3.1.

Because most of these bug reports do not contain the time when a developer was assigned or when a tester was assigned, we conservatively assign the period of time between when the bug is reported and when the first patch attempt is created as the time spent for vulnerability triage, the period of time between when the first patch attempt is created and when the last patch attempt is created as the time spent to construct a patch, and the period of time between when the last patch attempt is created and when the patch is committed as the time spent for regression test.

For these vulnerabilities, we find that the time and effort spent in constructing a patch is very significant. For the 8 vulnerabilities that took more than one day to create a patch, 89% of the time was spent in constructing a patch. And 9 of the vulnerabilities took between two to six attempts to patch correctly. Particularly the bug report of one proftpd vulnerability (CVE-2012-6095 [6]) contains five attempts of patch, of which the last patch attempt was created 96 days after the first patch attempt was created, and 29 comments from the developer and the testers, of which the comments from the developer along the time line include "*This updates the previous patch ...*", "*This patch builds on the previous one ...*", "*I've just committed more changes ...*", "*Hopefully the combination of ... addresses the remaining issues.*", "*Unfortunately I don't have a good/easy fix/solution for this yet.*", and one of the very last comments from the testers is still "*I'm afraid I found a bug in ...*".

3.2 Complexity of Security Patches

To understand further why constructing a patch is non-trivial, we further study the complexity of the patches, which are available for all the vulnerabilities shown in Table 3.1. We use column “SLOC” to show the number of lines of source code in patches, column “#Functions” to show the number of functions that are changed by patches, and column “#Files” to show the number of source code files that are changed by patches. We find that on average patches consist of 54 lines of source code and span 4 functions in 2 source code files. This suggests that on average, patches involve non-trivial changes to the application code. As a result, the vulnerability window is likely inherent to patches, as time must be spent by human engineers to understand the vulnerability, design and implement the fix, and finally test and review the patch before release. Due to our need for detailed bug reports and source code patches in performing this study, we were restricted to open-source applications. However, we found no evidence that these conclusions are restricted to open-source projects, and so we believe they should apply equally to both open- and closed-source applications.

Chapter 4

Neutralizing Vulnerabilities with SWRRs

To address the problem of pre-patch window presented in Chapter 3, we propose an approach to preventing vulnerabilities from being triggered. This approach uses a novel mechanism that can be automatically synthesized and integrated into target applications either at the release of these applications or as patches after the release of these applications. Particularly when it is integrated in the former manner, it allows users to prevent vulnerabilities from being triggered without the need of patches and thus substantially reduces or eliminates the pre-patch window. This chapter describes the details of this approach and our prototype implementation.

4.1 Introduction

Patches are the commonly-accepted solution for completely preventing a security vulnerability from being exploited. Patches fix security vulnerabilities and, in most cases, do so with no loss of functionality or performance for the application. However, despite their benefits, patches are not perfect.

An often ignored drawback of patches is the *pre-patch window of vulnerability* that exists between the time a vulnerability is discovered and the time a patch is issued. This vulnerability window is inherent to security patches because patches must be manually created and tested, which takes time and effort to do correctly. Although a large number of techniques have been proposed to automatically generate patches to fix vulnerabilities [78,79,95,100,110,127,129], they have not been widely adopted in practice. As a recent study [111] and our own findings in Section 4.2 demonstrate, the length of this window can be significant, and is unlikely to decrease due to the average complexity of a security patch. While the risk of exploitation during the pre-patch window can be reduced by keeping the vulnerability secret, this is just security-through-obscurity. As the highly active market of zero-day vulnerabilities demonstrates, there is no shortage of instances where attackers may be aware of and able to exploit vulnerabilities during this window [44].

While the period before a vulnerability is known can be reduced by better vulnerability detection and software engineering practices, we believe we can address the window of vulnerability that exists between the time the vulnerability is known to the developer (or to the public) and when the patch is issued. To do this, we take inspiration from configuration workarounds, which are a commonly used mechanism to address the pre-patch window of vulnerability. Configuration workarounds disable functionality related to the vulnerable code so that it cannot be triggered by an attacker. The recent high-profile Android Stagefright bug is a perfect example of this. This remote code execution vulnerability, which affected almost 1 billion devices, was discovered as early as April 2015, although not publicly disclosed until July. A patch was not available until August, several months after the vulnerability was discovered, and even at the time of writing, many smartphones models still do not have a usable patch [98,99]. Fortunately, the worst methods of exploitation via a malicious MMS can be prevented by configuring the MMS client on the phone to not automatically download media files from MMS messages. In exchange for some minor loss of functionality, the configuration workaround protects the user from the exploitation of a very serious vulnerability.

However, configuration workarounds are far from ideal for mitigating security vulnerabilities. Again, as we show in Section 4.2, configuration workarounds have low coverage – there are many vulnerabilities for which no configuration workaround exists because there is no configuration option that can disable the vulnerability. Configuration options are designed to allow users to easily alter the behaviour of a program, and thus only cover functionality that most users would like to enable or disable. Thus it is hardly surprising that very few vulnerabilities have configuration workarounds, and it seemingly becomes a matter of serendipity when a vulnerability can be neutralized with a configuration workaround.

Motivated by the problems of the pre-patch vulnerability window and the low coverage of configuration workarounds, we propose *Security Workarounds for Rapid Response (SWRRs)*, workarounds that can be mechanically generated to address a large percentage of vulnerabilities. The main challenge in designing SWRRs is that they must work in a broad range of circumstances. By their nature, vulnerabilities are not known a priori, and thus SWRRs must work for any vulnerability that can occur. Another challenge is that vulnerabilities can occur anywhere in an application, and be related to almost any type of functionality, but an SWRR must ensure that the application continues to work after the SWRR is applied. Thus, we design SWRRs to be simple and generic, relying on very few assumptions about either applications or vulnerabilities. The cost of this generality is that like configuration workarounds, users must be willing to accept some minor loss of functionality in return for protection from a vulnerability until a patch is issued.

Our key insight for making SWRRs generic and cheap to deploy is that application error-handling code, whose purpose is to gracefully return an application to a good state when an unexpected error arises, can be found mechanically and invoked using static analysis. Based on this insight, we designed and implmented a system called Talos, which detects such error-handling code using static analysis and adds SWRRs into a given application. Each SWRR prevents the execution of code where a vulnerability is located, and calls the error-handling code instead. With Talos, developers can deploy SWRRs either as patches or in-place as part of an application so that they can be activated with run-time loadable configurations.

In summary, SWRRs provide benefits for both software developers and users at a small cost. For the cost of having to run Talos and issue the resulting patch, software developers benefit by having a solution that protects their users; this affords them more time and less immediate pressure to create, test, and deploy a patch. Users benefit by having a solution that protects them during the pre-patch vulnerability window at the cost of having to accept some loss of functionality. In addition, in cases where users cannot install a patch for compatibility reasons or where no patch exists because the software is no longer supported, users can still use an SWRR to protect themselves.

This chapter makes the following contributions:

1. We propose SWRRs, which provide a low-cost way for software developers to quickly protect users during the pre-patch vulnerability window.

2. We design and implement a software tool called Talos to demonstrate that SWRRs can be practically deployed. To safely continue the execution of an application, Talos heuristically identifies error-handling code in the program and transfers execution to those paths to avoid having to execute vulnerable code.

3. We evaluate the effectiveness and coverage of the SWRRs inserted by Talos into 5 popular applications. When tested against 11 vulnerabilities, SWRRs generated by Talos successfully neutralize the vulnerabilities in all cases. Empirical tests on 320 Talos-generated SWRRs show that they can achieve an effective coverage that is 2.1$\times$ that of traditional configuration workarounds.

We begin in Section 4.2 with a study based on data we collected that demonstrates the motivation behind SWRRs. Then we give an overview of SWRRs in Section 4.3 and describe Talos, our tool for automatically inserting SWRRs into application source code, in Section 4.4. We provide details about the implementation of Talos in Section 4.5. We then evaluate the SWRRs that Talos instruments into applications in Section 4.6. We discuss the limitations and other issues of SWRRs in Section 4.7. We then provide a comparison with related work in Section 2.1 and conclude in Section 4.8.

Table 4.1: Configuration workaround statistics.

Application	#Options	#Vulnerabilities	Workaround	Period
lighttpd	88	27	14.8%	2005-14
apache	74	42	16.7%	2002-14
squid	174	30	6.7%	2001-15
proftpd	28	20	20.0%	2004-13
IE	33	31	54.8%	2000-14
Office	325	32	37.5%	2000-11

4.2 Configuration Workaround

Since configuration workarounds represent the current best solution for mitigating the vulnerability window, we present our study of configuration workarounds for recent security vulnerabilities. We define a configuration workaround as any vulnerability mitigation that involves modifying the configuration of the application (i.e., configuration options supported by the application) and exclude many other common fixes such as patching the application binary, disabling the application, or placing the vulnerability out of the reach of attackers (e.g., tightening firewall rules).

4.2.1 Configuration Workaround Coverage of Vulnerabilities

For this study, we use the same set of open-source applications that are used in Chapter 3 and add two popular closed-source applications: Internet Explorer and Microsoft Office. We also exclude sqlite because sqlite does not support any configuration options. For each application, we again randomly select a number of vulnerabilities and search both the software vendors' websites and Internet to determine if a configuration workaround is available. We tabulate the percentage of vulnerabilities examined for which we were able to find a configuration workaround as well as the time period over which the manually examined vulnerabilities were reported. We also tabulate the number of configuration options for each application. Table 4.1 presents the results. Column "#Options" shows the number of configuration options that each application has. Column "Workaround" shows the percentage of vulnerabilities that have configuration workarounds. Column "Period" shows the earliest and latest time when the vulnerabilities are reported. For IE and Office, we cite the number of configuration options measured by Ocasta [63]. For other applications, we obtain the list of their configuration options from their source code using either static analysis, manual examination, or user documentation. While it is difficult to say whether a small number of configuration options indicates that each option covers a large amount of code, in general we can see that the number of configuration options is usually small.

We observe several trends in the results of our study. First, configuration workarounds are listed for every application in our study. This shows that the use and disclosure of configuration workarounds is widespread across software projects. Second, the percentage of vulnerabilities that have workarounds is relatively low – a weighted average (by # of vulnerabilities) shows that only 25.2% of the vulnerabilities have configuration workarounds. As a result, the instances in which a security vulnerability can be neutralized with an existing configuration workaround is low.

Qualitatively, we find that many configuration workarounds disable an entire "module" of functionality that was associated with the vulnerable code. This suggests that many configuration workarounds cause some collateral damage; they not only disable the vulnerable code, but may also unnecessarily disable other non-vulnerable functionality as well. For example, vulnerability CVE-2011-4362 in lighttpd [15] is the result of an incorrect bounds check in the code that is only called during base64 decoding of credentials for HTTP basic authentication. However, the posted configuration workaround disables all types of authentication because it is the only configuration option that can prevent the vulnerable code from being executed. This means that other types of authentication that do not rely on base64 decoding, such as digest and NTLM authentication, are unnecessarily disabled. In general, the coarseness of the configuration options means that the configuration workarounds frequently disable more functionality than is strictly necessary.

Objectively speaking, it is not a complete surprise that configuration workarounds, while widespread in their usage across applications, are generally applicable to a minority of vulnerabilities and might only be able to disable code at a coarse granularity. Having fewer configuration options simplifies testing and generally improves usability, motivating developers to minimize the configurability of their applications. There are likely many regions of code that cannot be disabled by the limited number of configuration options, resulting in many vulnerabilities for which there is no configuration workaround.

4.3 Overview

4.3.1 SWRR Objectives

From our study of configuration workarounds we found that while configuration workarounds are commonly used, they have very low coverage of vulnerabilities thus reducing their utility. Despite this, the reason why configuration workarounds are still used is that they impose no additional effort on the part of the developer. In essence, they provide a small, but tangible benefit for free. While it might seem obvious that a special purpose mechanism like SWRRs can improve on the coverage of configuration workarounds, we remain cognizant that, to be competitive, they must at the same time impose little or no engineering cost. Furthermore, as a temporary alternative to a patch, they must be quick to generate as compared to construct a patch. We achieve low-effort by automatically generating SWRRs with Talos. However, if designed improperly, an automatically generated SWRR may do more harm than a manually created configuration workaround. As a result, we state the following objectives for our design of Talos and the SWRRs it creates:

- **Security:** An SWRR should neutralize its intended vulnerability and, in doing so, it should not introduce new bugs or vulnerabilities.

- **Effective Coverage:** SWRRs should be able to cover many more vulnerabilities than configuration workarounds. Effective coverage is a product of two components: (1)

the number of vulnerabilities whose code SWRRs can disable (which we call "basic coverage"), and (2) the number of SWRRs that, when enabled, result in a minor loss of functionality similar to what would be expected from a configuration workaround.

- **Low Cost:** SWRRs are mechanically inserted into an application using Talos, thus minimizing the engineering effort required to use SWRRs. In cases where a binary SWRR patch cannot be issued, it should be possible to deploy SWRRs in-place like configuration workarounds with minimal performance overhead.

Configuration workarounds are very unlikely to introduce new bugs or vulnerabilities since they have been tested; we expect the same behaviour from SWRRs, however, we limit our security objective to avoiding vulnerabilities that can compromise the confidentiality and integrity of a program. It is possible and acceptable for Talos to create an SWRR that causes the application to terminate, even though this creates a potential denial-of-service vulnerability. We believe this is acceptable because most state-of-the-art vulnerability mitigation techniques (such as Address Space Layout Randomization (ASLR), Control Flow Integrity (CFI), and non-executable stacks) aim to turn memory corruption exploits or malicious control flow transfers into program crashes, which also result in the termination of the program [38, 90, 109, 121, 138, 139]. As a result, our design of SWRRs aims to completely avoid confidentiality and integrity vulnerabilities in exchange for some (small) probability of introducing a denial-of-service vulnerability.

Regarding the effort to generate an SWRR, more effort is always required to generate a full patch than a SWRR. This is because a full patch must preserve all the functionality of the application while SWRRs explicitly allow some loss of functionality. As described in the third paragraph of Section 4.3.2, Talos only requires the function that contains the vulnerable code, which can be obtained from a crash report for example. On the contrary, to create a full patch, a developer needs to understand the exact cause of the vulnerability and all the conditions under which the vulnerability is triggered. In addition, the developer needs to devise new code that retains all desired functionality of the old code but does not contain the vulnerability.

While generating a full patch always takes more effort than generating an SWRR, the difference in effort is dependent on the complexity of the vulnerability. The more complex the vulnerability is, the larger the difference in effort will be. In contrast the amount of effort to use a SWRR is fairly independent of the complexity of the vulnerability, as it just requires knowing the function that contains the vulnerable code. For simple vulnerabilities, the difference in effort might be small, but our result in Table 3.1 indicates that a fair number of vulnerabilities can be quite complex.

4.3.2 SWRR Deployment

There are two possible deployment methods for SWRRs. In the first deployment method of SWRR, which we call *in-place deployment*, Talos is run on the application code base before it is released. Talos inserts an *SWRR check* into every function in the application. Each SWRR check reads and checks a corresponding *SWRR option* in an accompanying *SWRR configuration file*. This allows the application developer to selectively disable code in an application without having to replace the binary by pushing out a new SWRR configuration file instead. Alternatively, the user may change the configuration file to enable the appropriate SWRR if they know which function the vulnerability occurs in. In-place SWRR deployment is useful in scenarios where runtime performance is not critically concerned or where replacing application binaries is difficult, such as in smart phones and other embedded devices.

In the second deployment method of SWRR, which we call *patch-based deployment*, the application developer will run Talos on the application code base when they learn of a new vulnerability, passing Talos the information it requires about the vulnerability. Talos will then insert code that will disable the vulnerable function(s). The application developer will then compile the instrumented code and issue the resulting binary as a temporary patch to users. The application developer can perform minimal testing on the temporary patch as SWRRs are unlikely to cause serious loss of functionality in most cases, which is shown in our evaluation.

Because the SWRR patch is simply a return statement added to the very beginning of a function, Talos can even use binary rewriting to directly overwrite the binary of an application with a return instruction at the start of a function rather than modifying the source code of the application.

Using an SWRR requires that the location of the vulnerability be known. We argue that this is a reasonable requirement – by the time a vulnerability is discovered and confirmed, the location of the vulnerable code is generally known, albeit a proof-of-concept exploit is often not publicly available. For example, many of the CVE vulnerability reports we used in our experiments specifically list the function in which the vulnerability is located.

Each of the two SWRR deployment methods has its own pros and cons. The pros of in-place deployment include no need to generate and roll out an SWRR, and not requiring re-compilation, but its cons include slight increase of code size and very small loss of runtime performance, as we will show in Section 4.6. The pros of patch-based deployment include minimum increase of code size and no loss of runtime performance, but its cons include the need to generate and roll out an SWRR and requiring re-compilation.

As the main goal of SWRR is to provide a rapid response when a vulnerability is newly discovered, we use Figure 4.1 to illustrate the commonalities and differences between the two SWRR deploy methods and the conventional approach of releasing a full patch, when they are used to address a newly discovered vulnerability. In the figure, the workflows of different approaches are distinguished with the use of different types of arrows. The legends used in the figure are explained in the dotted box at the bottom of the figure.

When a vulnerability is newly discovered, each approach starts with vulnerability triage and finding the location of the vulnerability. After that, each approach consists of different steps. First, releasing a patch requires software developers to find the cause of the vulnerability and to construct a patch, which can demand a considerable amount of effort and time, testers to perform regression test to ensure the patch does not break any functionality, the vendor to release the patch, and end-users to apply the patch to their installed application. Second, the in-place SWRR deployment requires developers to identify the SWRR that can mitigate the

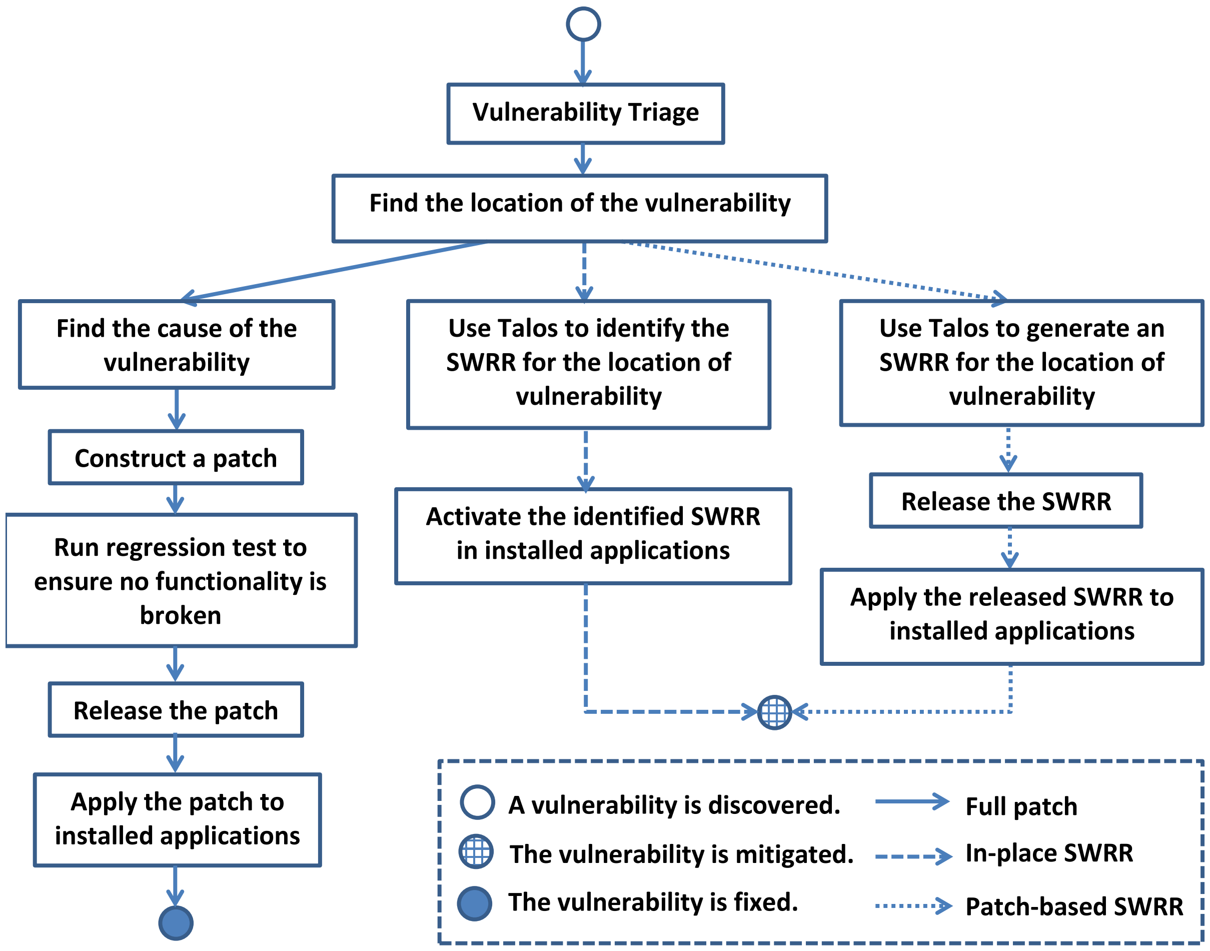

Vulnerability Triage
Find the location of the vulnerability
Find the cause of the vulnerability
Construct a patch
Run regression test to ensure no functionality is broken
Release the patch
Apply the patch to installed applications
Use Talos to identify the SWRR for the location of vulnerability
Activate the identified SWRR in installed applications
Use Talos to generate an SWRR for the location of vulnerability
Release the SWRR
Apply the released SWRR to installed applications
A vulnerability is discovered.
The vulnerability is mitigated.
The vulnerability is fixed.
Full patch
In-place SWRR
Patch-based SWRR

Figure 4.1: Different approaches to addressing a newly discovered vulnerability.

vulnerability, which can be done by simply running Talos, and end-users to activate the SWRR, which is simply to change the application's configuration, because the SWRR has already been built into the application before the application is released. Finally, the patch-based SWRR deployment requires developers to generate an SWRR specifically for the vulnerability, which is also done by running Talos, vendors to release the SWRR as a patch, and end-users to apply the generated SWRR to their installed applications, which is similar to apply a patch. Note that at the end, the conventional approach of releasing a patch will fix the vulnerability, while both SWRR deployment methods only mitigate the vulnerability. However, SWRR deployment methods take less steps and requires much less effort and time.

4.3.3 The Error-handling Code Intuition

Talos must insert SWRRs so that they can neutralize vulnerable functions without violating security and without needing to understand complex program-specific semantics. Talos is almost completely application-agnostic, requiring only a small amount of application-specific information from developers. The key questions are then what application characteristic (1) is present and similar across all or nearly all applications, and (2) can allow Talos to recover from the unexpected redirection of execution away from the vulnerable code?

Our intuition is that code whose purpose is to handle unexpected or abnormal error conditions fits these requirements. First, error-handling is found in nearly every type of application. Essentially any sufficiently complex application that interacts with its environment must gracefully handle unexpected situations such as invalid inputs, inadequate resources, or unexpected delays that it encounters; this is generally accomplished with what we generically refer to as *error-handling code*. Second, error-handling code is designed to be invoked when the application encounters these unexpected or abnormal situations and thus, by nature, it must conservatively return the application back to a known state. In fact, the majority of error-handling code takes great pains to try to avoid violating confidentiality by leaking sensitive information or violating integrity by corrupting data. Instead, most error-handling code remedies an abnormal situation by aborting the current task and cleaning up any intermediate state or, in the worst case, gracefully halting the application if continuation is not possible. As a result, the intuition behind the goals of error-handling code fits well with the security goal of protecting the confidentiality and integrity of applications.

If an application does not have existing error handling code for a vulnerable function, it is possible for Talos to instrument new error handling code designed for the vulnerable function into the application and use the new error handling code in the SWRR for the vulnerable function. However, it will require deeper understanding of the code of the callers of the vulnerable function or even changes to the code of the callers to ensure that the new error handling code does not introduce new bugs.

A keen reader might raise the question on the reliability of existing error handling code. Indeed it is possible that an SWRR happens to use some error handling code that is rarely used in practice and contain some bug. In such case, the bug might manifest when the SWRR is activated. However, our evaluation on the rate of unobtrusiveness for 320 SWRRs across different applications in Section 4.6.2 indicates that the vast majority of existing error handling code is reliable.

4.4 Design

We now describe how Talos inserts SWRRs into application code without introducing new security vulnerabilities. First, we explain how Talos sets about instrumenting an application with SWRRs. Then, we detail the heuristics Talos uses to identify error-handling code within an application for the purposes of SWRR instrumentation.

4.4.1 Inserting SWRRs

There are two design decisions to be made when Talos inserts SWRRs into an existing code base. First, we must decide the granularity of code that each inserted SWRR should enable or disable. The granularity of code that is protected by each SWRR has a bearing on its security and unobtrusiveness. This is because error-handling code can broadly be classified into two categories: intra-procedural error-handlers that operate completely within a function, and inter-procedural error-handlers that are unable to completely handle the error within the function and must expose the error to the function's caller. The error handlers in the former category are difficult for Talos to use as they are tightly coupled with the path within the function used to invoke the error-handling path. For example, they may free memory that they know was allocated on the path leading to the error-handling code, or conversely fail to free memory since they know the paths leading to the error-handling code did not allocate it. If Talos redirects execution to such an error-handling path without understanding the internal semantics of the function, it could result in a double-free bug.

However, error-handling code that exposes the error to the caller must be more conservative because it must be written in such a way that correctness guarantees are met independently of the calling context. As a result, such error-handling code often seeks to ensure that modifications made to application state by the function are undone and that an appropriate value is returned to the caller so that the caller can then handle the failure. For example, an input sanitization function that fails due to an out-of-memory error might free any resources acquired up to that point and then return an error code to the caller so that the caller can conservatively reject the unsanitized input. This intuition implies that functions that contain such error-handling code can safely do nothing as long as the caller is notified that the function has encountered an error. As a result, Talos instruments SWRRs to enable or disable code at the granularity of a function. While there is no guarantee that this intuition is always true, we find that it does hold for a large number of cases allowing Talos to instrument applications with SWRRs that are secure and provide better effective coverage than configuration workarounds as we demonstrate in our evaluation in Section 4.6.

Given that an SWRR option should control the execution of a function, instrumenting a function with an SWRR is fairly straightforward. To instrument a function, Talos adds the code in Listing 4.1 or Listing 4.2 to the function, depending on whether in-place deployment or patch-based deployment is used. For in-place deployment, a check is first performed on line 3 to determine whether the corresponding SWRR option (*SWRR_option*) is enabled; if it is, the entire function body is skipped and the error code (*error_code*) that has been statically extracted from the error-handling code is returned to the caller on line 4. In this section, the text will mostly assume in-place deployment since it is the slightly more complex of the two options, however, we expect that in most cases patch-based deployment will be preferable.

Since a suitable error code must be found for each function instrumented with an SWRR, Talos can only instrument a function if: (1) it can determine if the function has inter-procedural error-handling code, and (2) it can extract the value that the error-handling code returns to

```
procedure FIND_FUNCTIONS($Functions$)
    $to\_instrument \leftarrow \emptyset$
    $SWRR\_map \leftarrow \emptyset$
    for $f \in Functions$ do▷ Apply 2 main heuristics
        if error_logging(f) then
            $to\_instrument \leftarrow \{f, error\_code(f)\}$
            $SWRR\_map \leftarrow \{f, new\_option()\}$
            $remove(Functions, f)$
        else if NULL_return(f) then
            $to\_instrument \leftarrow \{f, NULL\}$
            $SWRR\_map \leftarrow \{f, new\_option()\}$
            $remove(Functions, f)$
        end if
    end for
    for $f \in Functions$ do ▷ Apply 2 extension heuristics
        if $f' =$ propagate$(f, to\_instrument)$ then
            $SWRR\_map \leftarrow f, new\_option()\}$
            $to\_instrument \leftarrow \{f, error\_code(f')\}$
            $remove(Functions, f)$
        end if
    end for
    for $f \in Functions$ do
        if $f' =$ indirect$(f, to\_instrument)$ then
            $SWRR\_map \leftarrow \{f, option(f')\}$
        end if
    end for
     return $\{to\_instrument, SWRR\_map\}$
end procedure
```

Figure 4.2: Talos algorithm for identifying functions to instrument.

```
int example_function(...) {
  /* SWRR inserted at top of function */
  if (SWRR_enabled(<SWRR_option>))
    return <error_code>;

  /* original function body */
  ...
}
```

Listing 4.1: SWRR instrumentation - In-place Deployment

```
int example_function(...) {
  /* SWRR inserted at top of function */
    return <error_code>;

  /* original function body */
  ...
}
```

Listing 4.2: SWRR instrumentation - Patch-based Deployment

be used as the error code. While other work has used dynamic profiling to try to identify error-handling code [114], this requires a comprehensive suite of test inputs to find all error-handling code. We assume this is not always available, so to maintain a low deployment cost, Talos relies exclusively on static analysis. Talos thus uses several heuristics based on common programming idioms that are indicative of error-handling code.

The procedure Talos uses for deciding which functions in an application to instrument has several stages as illustrated in Figure 4.2. The procedure takes as input the set of all functions in the application; it returns a set of functions capable of being instrumented as well as a map of functions to their corresponding SWRR options. Talos first iterates over each function, applying the two main heuristics used to statically detect if the function has error-handling code. If such code is detected, then Talos adds the function along with the *error_code* extracted from the error-handling code to the set of functions it will instrument and removes it from further consideration. In addition, Talos creates a new SWRR option for the function and adds it to the SWRR option-to-function map it maintains. After all functions have been checked with the two main heuristics, Talos then applies the two "extension" heuristics to identify cases where it can extend error-handling code into a function's caller or callee. Talos uses the *error propagation* heuristic to identify cases where the error code for a function can be used in an

SWRR for the callers of the function, even if the callers themselves do not have error-handling code. Finally, Talos also uses the *indirect* heuristic to identify any remaining cases where a function doesn't have error-handling code but can be disabled by a caller (or callers) that have been instrumented by an SWRR. In these cases, the SWRR map is updated so that this function is also associated with the SWRR option of its caller(s).

When a function has more than one piece of error handling code, Talos needs to decide which error handling code to use. To make the decision, Talos assigns different ranks to heuristics and chooses to use the error handling code detected with the heuristic with the highest rank. Talos assigns the main heuristics with the highest rank, the error propagation heuristic with the second highest rank, and the indirect heuristic with the lowest rank. If more than one piece of error handling code are identified with heuristics of the same rank, Talos chooses to use the error handling code closest to the entry of the function.

4.4.2 Main Heuristics

We first describe the two main heuristics Talos uses to identify error-handling code in functions. We will then describe the two extension heuristics.

Error-logging function heuristic

The first heuristic is used to identify program paths that call error-logging functions. Error-logging functions are called to log information when the application encounters an error. To use this heuristic, Talos requires developers to specify the error-logging functions in an application. For each of the surveyed applications, Table 4.2 lists the total number of functions and the number of error-logging functions, where we have manually identified the latter by inspecting the source code. We can see that, even in fairly large applications with hundreds or thousands of functions, many applications have very few and, in many cases, only one error-logging function. Anecdotally, we also find that if there is more than one function, they are still often easy to find because they are all declared within a single header-file in the application source

Table 4.2: Number of functions and number of error-logging functions.

Application	#Functions	#Error-logging Functions
lighttpd	665	1
apache	2,082	4
squid	1,346	1
proftpd	1,092	1
sqlite	1,562	3

code. Thus, we feel that the effort required for developers to specify the error-logging functions in an application is quite reasonable.

The presence of an error-logging function is indicative of error-handling code. However, recall that Talos requires the error-handling code to be inter-procedural, which means that it must also signal the error to the function's caller. Thus, to identify such code using an error-logging function, Talos requires the following: (1) the error-handling code must call an error-logging function, (2) the error-handling code must return a constant value, and (3) the error-logging function and return statement must be guarded by a conditional branch. A conditional check that dominates the error-logging function indicates that the path will only be taken under specific circumstances and a constant return value is required for the current function to signal to its caller that it terminated with an abnormal condition. Listing 4.3, which shows error-handling code in Apache 2.2.19, illustrates how real code fits this heuristic. The error-handling code is only executed when condition `(name == NULL)` is true. It then calls the logging function `ap_log_error()` and returns the constant value `APR_EBADF` to its caller, which indicates that it cannot proceed because of a bad filename.

Talos instruments such functions to always return a constant return value consistent with the error-handling code when the SWRR is activated (i.e. in place of *error_code*) in Listing 4.1.

NULL return heuristic

If the error-logging heuristic does not identify the presence of error-handling code, Talos next uses the NULL return heuristic. The intuition behind this heuristic is that when a function

```
if (name == NULL) {
  /* Apache's error logging function */
  ap_log_error(APLOG_MARK, APLOG_ERR, 0, NULL, "Internal error: pcfg_openfile()
      called with NULL filename");
  return APR_EBADF; /* indicates to caller that error occured */
}
```

Listing 4.3: Error-logging code example from Apache

that normally returns a pointer returns NULL instead, it indicates that the function could not successfully perform its normal operation. This may happen due to an unexpected error or due to an invalid input.

Talos would instrument such functions with an SWRR that returns NULL as its error code. However, Talos must be conservative because not all functions can legally return a NULL to their callers. If an SWRR were to force such a function to return NULL, the caller may dereference the value without checking for NULL and crash the program. To infer whether a function can return NULL or not, Talos checks that there is at least one instance of a call to the function where the caller checks the return value against NULL. The reason Talos does not do this for all call sites is that in some cases, the check for NULL may be hard to detect. For example, consider the case where the caller writes the returned pointer value to a linked list and then the value is only checked against NULL when it is dequeued from the linked list.

Due to this limited check, it is possible for an SWRR returning NULL to crash the program if no code ever checks for NULL after executing the SWRR. In this case, the SWRR essentially turns the vulnerability into a denial-of-service, but still prevents more severe consequences of exploiting the vulnerability such as elevating privilege or hijacking program execution.

4.4.3 Extension Heuristics

We now discuss the two heuristics that Talos uses to extend coverage from functions that have identified error-handling code to those that do not.

Error propagation heuristic

This heuristic is based on the observation that many times the error code returned by a function is used as a return value by the caller of such functions. This has the effect of propagating error codes up the call chain and, as a result, can be used to detect the correct error codes for both callees and callers of a function.

This error propagation manifests in three ways. First, we find that some functions have an execution path that calls another function and simply uses the return value of the function call as their own return value. As illustrated by a simplified code snippet from lighttpd in Listing 4.4, `config_insert_values_global` calls `config_insert_values_internal` and uses the return value of the callee as its own return value at line 3. As a consequence, the error code of `-1` for `config_insert_values_internal`, identified by Talos using the error-logging heuristic at line 10, can be used as the error code for `config_insert_values_global`.

Second, the error code can also be translated before it is propagated up the call chain, as illustrated again in Listing 4.4. Here, `mod_secdownload_set_defaults` checks the return value of a call to `config_insert_values_global` at line 17 and returns a constant value `HANDLER_ERROR` at line 18 if the return value from `config_insert_ values_global` indicates an error. Unlike in the first case, `mod_secdownload_set_defaults` does not use the return value of `config_insert_values_global` directly, but translates it to its own error code if the callee returns an error. To identify this kind of error propagation, Talos looks for a statement that returns a constant and is control-dependent on the return value of a function call. Talos then checks: (1) whether the function in the predicate has been previously identified as having error-handling code, and (2) whether the identified error code can satisfy the predicate of the control dependency. If so, the returned value becomes the error code for the function and Talos marks the function as eligible for SWRR instrumentation. In the example, Talos identifies `HANDLER_ERROR` as the error code for `mod_secdownload_set_defaults`.

Third, the error code can be inferred down the call chain as shown in the code of `http_request_parse` in Listing 4.4. `http_request_parse` has an error path that calls an error logging function

```
int config_insert_values_global(....) {
....
    return config_insert_values_internal(....);
}

int config_insert_values_internal(....) {
....
    if (....) {
        log_error_write(....); // error logging
        return -1;
    }
....
}

SETDEFAULTS_FUNC(mod_secdownload_set_defaults) {
....
    if (0 != config_insert_values_global(....)) {
        return HANDLER_ERROR;
    }
....
}

int http_request_parse(....) {
....
    if (0 != request_check_hostname(....)) {
        log_error_write(....); // error logging
        return 0;
    }
....
}
```

Listing 4.4: Error propagation example from lighttpd

when the return value of the call to `request_check_hostname` is not zero. From this, Talos infers that the error code of `request_check_hostname` must be a non-zero value. To identify this kind of error propagation, Talos checks if any identified error path is control dependent on the value of a predicate involving the return value of a function call. If it is, Talos tries to find a constant value that can satisfy the predicate and then uses that constant value as the error code of the callee of the function. For this example, Talos identifies `1` as the error code for `request_check_hostname`.

Indirect heuristic

If a function is only called by functions that have been identified as eligible for instrumentation, Talos takes advantage of the fact that by disabling all the callers of the function, the function itself can be disabled by SWRRs. In these cases, Talos does not insert any instrumentation into these functions, but simply updates the SWRR map to indicate that the function in question

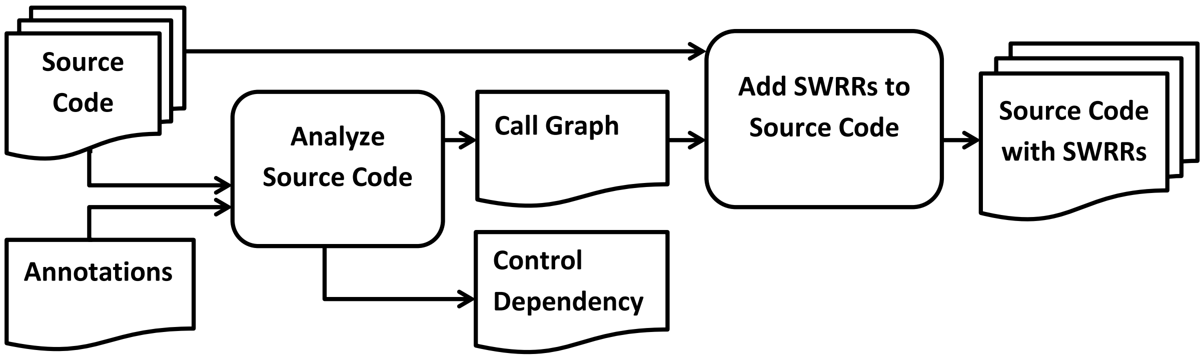


Figure 4.3: Workflow of Talos

can be disabled by activating one or more other SWRRs.

4.5 Implementation

We have implemented a prototype of Talos. Due to the fact that Talos needs a program's call graph to find locations for SWRR insertion, our prototype instruments a program in two phases, as shown in Figure 4.3. The first phase analyzes the source code of the program and is implemented as an analysis pass of LLVM using 1,823 lines of C/C++ code, while the second phase adds SWRRs to the source code and is implemented using 1,852 lines of Python code. In the first phase, Talos takes as input the source code of a program and the annotation of the error logging functions of the program, analyzes the source code using static analysis, and outputs: the program's call graph, the control dependency of each statement of the program, whether each statement is followed by a return, the start line number of each function, and the line number of each statement. In the second phase, it adds SWRRs to as many functions as possible in the source code based on the output of the first phase.

During our implementation, we found that function calls using function pointers are frequently used by applications, particularly to invoke the functionality of loadable modules. Loadable modules are often used as a configuration workaround for vulnerabilities, so we expect that SWRRs should work for these as well. We note that these kinds of function calls usually use function pointers embedded as fields of some C/C++ structures. To identify the caller and callee of a function call using a function pointer, we match a call to a function pointer field of a structure, by identifying the assignment or initialization of the same field. This method is imprecise, but we did not notice any issue with it in practice.

To identify error-handling code that can be used for SWRRs, we need to find out whether a call to an error logging function is followed by a return statement. At first, we tried to label such cases when a call is followed by a return statement within the same basic block of the call. However, we found that LLVM merges all occurrences of return statements within a function into a single return at the end of the function and replaces all other return statements with branch statements. Sometimes a return is translated into a chain of unconditional branch statements that lead to the only return statement of a function. Hence a call to an error logging function and the return statement following it sometimes do not belong to the same basic block. Furthermore, some applications' error logging function is actually a macro defined as an `if` statement, so the call to the error logging function and the return statement belong to two different basic blocks. As a consequence, we label a call as being followed by a return when the call is on a path that unconditionally leads to a return statement.

4.6 Evaluation

We evaluate how well SWRRs created by Talos meet the three objectives we laid out in Section 4.3. First, we evaluate the security of SWRRs, i.e. whether SWRRs can successfully prevent exploits of real-world vulnerabilities. We search among the vulnerabilities that we study in Table 3.1 for those that have public proof-of-concept exploits. And we find 11 vulnerabilities that can be exploited in our testing environment. The information on the official patches for these vulnerabilities are shown in Table 4.3. The average pre-patch window for these vulnerabilities is 54 days. We then evaluate the effective coverage by measuring the basic coverage of SWRRs and the rate of unobtrusive SWRRs. We define "unobtrusive SWRR"

Table 4.3: Evaluated vulnerabilities.

CVE ID	Delay (Days)	SLOC	#Functions	#Files
lighttpd-CVE-2011-4362	25	2	1	1
lighttpd-CVE-2012-5533	2	4	1	1
lighttpd-CVE-2014-2323	N/A	21	2	2
apache-CVE-2014-0226	223	20	2	2
squid-CVE-2009-0478	5	23	2	2
squid-CVE-2014-3609	105	6	1	1
sqlite-CVE-2015-3414	N/A	24	4	4
sqlite-OSVDB-119730	N/A	N/A	N/A	N/A
proftpd-OSVDB-69562	4	4	1	1
proftpd-CVE-2010-3867	13	74	1	1
proftpd-CVE-2015-3306	N/A	104	5	1
AVERAGE	54	28	2	2

as an SWRR that only disables minor application functionality while leaving the majority of an application's functionality intact, much like a configuration workaround, and an "obtrusive SWRR" as an SWRR that disables the majority of an application's functionality, making it unusable. Thus, the basic coverage of SWRRs is reduced to their effective coverage by the percentage of SWRRs that are obtrusive. Finally, we evaluate the performance cost of SWRRs when using in-place deployment.

All our evaluations were conducted on a 4-core 3.4GHz Intel Core i7-2600 workstation, with 16GB RAM, 3TB of SATA hard drive and running 64-bit Ubuntu 12.04.

4.6.1 Security

This evaluation answers the question: **"Do SWRRs successfully neutralize vulnerabilities without introducing new vulnerabilities?"** To test an SWRR, we need one vulnerability that is covered by the SWRR; to check whether an SWRR neutralizes a vulnerability, we also need an exploit for that vulnerability. These two requirements are challenging to meet for many SWRRs and it also requires non-trivial manual effort to check whether new vulnerabilities are introduced. Nevertheless, we make our best effort to find as many vulnerabilities as possible that could be used for this evaluation. The resulting 11 vulnerabilities, disclosed between 2010 and 2015, are used to evaluate the five popular applications as shown in Table 4.4.

To validate security, we check if the SWRR neutralizing the vulnerability successfully thwarts an exploit of the vulnerability. To test whether the exploit is neutralized or not, we either use a published exploit or create a proof-of-concept exploit if no published exploit is available. We verify that the exploit works on the unprotected application and then enable protection using the appropriate SWRR option and try the exploit again. If the exploit fails, we say the SWRR has protected the security of the application.

We can also test whether an SWRR is unobtrusive or not. To do this, we first classify the functionality of each application into two categories, major and minor, by studying its user documentation. We then design two sets of test inputs, major and minor, to exercise as much the application's major functionality and minor functionality as possible. For each application, we make use of the existing test suite of an application if such a test suite is available. Otherwise we make our best effort to create our own sets of test inputs and test suite. We then use this test suite to determine if no or only minor functionality is lost, in which case the SWRR is unobtrusive; if major functionality is also lost, the SWRR is obtrusive.

Our results are summarized in Table 4.4, which also gives the heuristic used to instrument the SWRR that neutralizes the vulnerability, as well as whether availability is violated. Column "Security?" shows whether the exploit against a vulnerability is successfully neutralized by SWRR without introducing new vulnerabilities. Column "Unobtrusive?" shows whether the SWRR is unobtrusive. SWRRs successfully neutralize the exploits for all 11 vulnerabilities and in 8 cases there is no or only minor loss of functionality, making these SWRRs unobtrusive. We provide details on all 11 cases below. For the 3 cases where a posted configuration workaround also exists for the vulnerability, we compare the SWRRs unobtrusiveness with that of the configuration workaround.

lighttpd-CVE-2011-4362. This vulnerability allows a remote attacker to cause an out-of-bounds memory error [15]. The function `base64_decode` takes an untrusted `char*` and per-

Table 4.4: Security of SWRRs.

CVE ID	Heuristics	Security?	Unobtrusive?
lighttpd-CVE-2011-4362	NULL Return	Yes	Yes
lighttpd-CVE-2012-5533	Indirect	Yes	No
lighttpd-CVE-2014-2323	Error-Propagation	Yes	No
apache-CVE-2014-0226	Error-Logging	Yes	Yes
squid-CVE-2009-0478	Indirect	Yes	No
squid-CVE-2014-3609	Error-Logging	Yes	Yes
sqlite-CVE-2015-3414	Error-Propagation	Yes	Yes
sqlite-OSVDB-119730	Error-Logging	Yes	Yes
proftpd-OSVDB-69562	Error-Propagation	Yes	Yes
proftpd-CVE-2010-3867	Error-Logging	Yes	Yes
proftpd-CVE-2015-3306	Error-Logging	Yes	Yes

forms a base 64 decode during HTTP basic authentication by using each character in the untrusted string as a lookup into a table in memory. As `char*` is signed, an attacker could specify negative values and read memory from outside of the table. `base64_decode` has error-handling code that returns NULL, so Talos instruments the function with an SWRR that returns NULL, which successfully neutralizes the vulnerability. Since base64 decoding is disabled, all requests for basic HTTP authentication fail as if the password failed to decode properly. However, lighttpd functions completely normally (including other forms of authentication) as long as basic HTTP authentication is not used. This imposes less loss of functionality than the posted configuration workaround, which disables all forms of authentication. Thus, Talos provides security and provides an unobtrusive SWRR for the vulnerability.

lighttpd-CVE-2012-5533. This vulnerability allows a remote attacker to cause an infinite loop via a specially crafted HTTP connection header. The function `http_request_split_value` splits the fields in an HTTP connection header into an array, but can get into an infinite loop due to the vulnerability. `http_request_split_value` does not have error-handling code, but its caller does have error-handling code that returns `0`; Talos instruments the caller and successfully neutralizes the vulnerability, however, the side-effects of this are severe, as it causes all HTTP requests to be denied, because the caller is the main function that processes

HTTP requests. As a result, while the SWRR provides security, because the SWRR is enabled for all HTTP requests, lighttpd is unable to respond to any HTTP request so there is a major loss of functionality.

lighttpd-CVE-2014-2323. This vulnerability allows a remote attacker to execute an arbitrary SQL command via a specially crafted hostname in the host header of an HTTP request. The vulnerable function `request_check_hostname` checks the validity of hostnames, but it fails to deny hostnames that contain SQL commands. The caller of the function has an error path that calls an error logging function when the return value of the function is not zero, so Talos instruments the function with an SWRR that returns `1`, which successfully neutralizes the vulnerability. As a side-effect of activating the associated SWRR, any HTTP request that specifies a hostname (as opposed to an IP address) will receive a “400 - Bad Request” error response. While the SWRR provides security, because the vulnerable code is used for all HTTP requests with a hostname, which is in most cases the vast majority of requests, there is a major loss of functionality.

apache-CVE-2014-0226. A race condition in the `mod_status` module of apache httpd server allows an attacker to retrieve sensitive information [5]. The function `status_handler` displays administrative information about a web server, such as the web server’s performance and overhead, as a web site. It does not synchronize the use of data that can be modified concurrently by a different thread. `status_handler` has error-handling code that calls an error logging function and returns `HTTP_INTERNAL_SERVER_ERROR`, so Talos instruments the function with an SWRR that returns the error code, which successfully neutralizes the vulnerability. As a side-effect, all requests to the `mod_status` module return an error because `status_handler` is called in response to all requests to the module, but the application will continue to execute and respond to other requests normally. This vulnerability has a posted configuration workaround, which disables the entire `mod_status` module, with the exact same loss of functionality as Talos’ automatically generated SWRR. As a result, Talos provides security with an unobtrusive SWRR for this vulnerability.

squid-CVE-2009-0478. An integer overflow vulnerability allows a remote attacker to cause a denial-of-service by sending an HTTP request with a crafted HTTP protocol version number [27]. The function `httpMsgParseRequestLine` converts the HTTP version number of an HTTP request from a string to an integer, but it uses a signed integer to store the converted version number. As a result, a very large version number will cause an integer overflow and crash the server. `httpMsgParseRequestLine` does not have error-handling code, but its caller does (returns NULL); Talos instruments the caller, which successfully neutralizes the vulnerability. However, the side-effects of this are severe, as it causes all HTTP requests to be denied as every request must be parsed by `httpMsgParseRequestLine` and calls to this function always generate an error with the SWRR enabled. While the SWRR provides security, because the vulnerable code is used for all HTTP requests, squid is unable to respond to any HTTP request so there is a major loss of functionality.

squid-CVE-2014-3609. A missing validity check on the byte range specification of an HTTP request allows a remote attacker to cause a denial-of-service by sending an HTTP request with a specially crafted byte range specification [28]. The function `httpHdrRangeSpecParseCreate` parses the byte range specification of HTTP requests, but it does not correctly check the validity of the length calculated from certain byte range specifications and can cause the server to crash. `httpHdrRangeSpecParseCreate` has error-handling code that calls an error logging function and returns NULL, so Talos instruments this function with an SWRR that returns NULL, which successfully neutralizes the vulnerability. This causes the server to ignore the byte range specification from the client and always serve the full-length of the content. No confidential information is leaked since the client would have received the full-length content anyways if it had not specified a byte range. This vulnerability has a posted configuration workaround, which implements a filter that rejects requests with suspicious byte ranges. The loss of functionality is similar to the SWRR – only requests that specify byte ranges are affected in either case. Talos preserves security in this case with an unobtrusive SWRR.

sqlite-CVE-2015-3414. A vulnerability in the code that parses collation-sequence names in

SQL commands allows an attacker to cause memory corruption. The function `sqlite3ExprAddCollateStri` allocates memory for parsed collation-sequence names, but may use uninitialized memory when parsing a specially crafted collation-sequence name. `sqlite3ExprAddCollateString` does not have error-handling code and simply uses the return value of function `sqlite3ExprAddCollateToke` as its own return value. Due to imprecise static analysis, Talos incorrectly identifies that `sqlite3ExprAddCollateToken` could return NULL, although it is carefully written to always return a valid pointer. As a consequence, Talos instruments the function with an SWRR that returns NULL. Since `sqlite3ExprAddCollateString` should not be able to return NULL, the caller does not check the return value before dereferencing it causing sqlite to crash. If collation is not used, sqlite continues to operation normally, and since collation is not part of the core functionality of sqlite, we call this a minor loss of functionality. If restarted, sqlite continues to function normally.

sqlite-OSVDB-119730. An attacker can cause a memory error in sqlite with the meta command `trace`, which turns on or off the tracing of the execution of commands. The function `do_meta_command` processes all meta commands, which allows users to specify different settings when executing commands. It does not set a pointer to NULL after the memory which it references has been deallocated, and thus can cause a use-after-free memory error. `do_meta_command` has error-handling code that calls an error logging function and returns `1`, so Talos instruments the function with an SWRR that returns `1`; this causes sqlite to return an error to the meta command request. As a result, Talos protects the security of sqlite against this vulnerability. However, because `do_meta_command` is disabled, all other meta commands will also return an error, and thus the availability of all meta commands is violated. However, because this is only confined to meta commands, which are not part of the core functionality of sqlite, this SWRR is unobtrusive.

proftpd-OSVDB-69562. A backdoor that allows a remote attacker to access a root shell was planted into the source code of ProFTPD when ProFTPD's FTP server and mirrors were compromised [21]. The backdoor was added to function `pr_help_add_response`, which creates responses to HELP command, so that a HELP command with a specific argument would cause ProFTPD to execute a shell that can be accessed remotely. The caller of the function has error-handling code that calls an error logging function when the return value of the function is not zero, so Talos instruments the function with an SWRR that returns 1, which successfully neutralizes the vulnerability. As a result, the security and availability of the application are preserved. However, as in the previous case, ProFTPD will respond to all HELP commands with the error message “Unknown command" thus impacting the availability of the HELP facility. However, all other FTP commands continue to function normally. As a result, this is considered an unobtrusive SWRR.

proftpd-CVE-2010-3867. Multiple vulnerabilities in the mod_site_misc module allow a remote attacker to perform various directory and file operations using mod_site_misc commands without authentication. All vulnerable functions, such as site_misc_mkdir that creates a directory on the server upon users’ requests, have error-handling code that calls an error logging function and returns NULL; Talos instruments each of these functions with an SWRR that returns NULL and when all of the SWRRs corresponding to these functions are enabled, ProFTPD returns an error for all the vulnerable mod_site_misc commands. Other than this side-effect, users can continue to use all other FTP commands and thus the SWRRs provide security and are unobtrusive.

proftpd-CVE-2015-3306. Multiple vulnerabilities in the mod_copy module allow a remote attacker to read and write arbitrary files with mod_copy commands without authentication. Similar to CVE-2010-3867, all vulnerable functions (such as copy_copy, which copies files between different locations on the server) have error-handling code that calls an error logging function and returns NULL; Talos instruments each of these with an SWRR that returns NULL when enabled. Again, when the SWRR is activated, ProFTPD returns an error in response to all the vulnerable mod_copy commands. There are no other side effects and ProFTPD continues to work as expected, thus the SWRR provides security and is unobtrusive.

Table 4.5: Basic coverage of SWRRs.

Application	**Protected**	**Error Logging**	**Return Pointer**	**Propagation**	**Indirect**
lighttpd	89.8%	23.6%	1.5%	17.6%	47.1%
apache	77.5%	14.0%	11.9%	20.7%	30.9%
squid	76.6%	18.1%	5.6%	6.3%	46.4%
proftpd	86.1%	32.7%	13.6%	12.9%	26.9%
sqlite	45.3%	2.0%	6.5%	14.4%	22.4%
AVERAGE	75.1%	18.1%	7.8%	14.4%	34.7%

4.6.2 Effective Coverage

In this section, we aim to answer the question **"What is the percentage of vulnerabilities that can be mitigated with an unobtrusive SWRR?"** To answer this question, we perform a quantitative measurement of the two components that make up the effective coverage of SWRRs: the basic coverage and the rate of unobtrusive SWRRs.

Basic Coverage. To evaluate basic coverage, we measure the number of functions where Talos can find an error-handling path and identify an error-handling code to return, which is used to insert an SWRR. This measurement across the five applications is shown in Table 4.5. The first "Protected" column shows the total percentage of functions that are protected by SWRRs in each application. The remaining four columns then provide a breakdown by the percentage of functions that are protected by each of the four heuristics. If we assume that potential vulnerabilities are uniformly distributed across functions in the application, then the percentage in the Protected column gives the basic coverage for the application, which is the likelihood that a potential vulnerability can be disabled by an SWRR.

As Talos uses error-handling to infer the value that should be returned by an activated SWRR, the coverage depends very heavily on how much error-handling code is present in the application and how well Talos' heuristics can identify the error-handling code. Among the five applications, sqlite has the lowest basic coverage of 45.3% as well as a very low percentage of error-logging paths. In addition, sqlite has the lowest percentage of functions that can be protected indirectly. This is likely because sqlite has a simpler call graph than the other applications.

On the other hand, lighttpd has the highest basic coverage of 89.8% because it has a particularly high percentage of error logging paths as well as a high percentage of functions that can be protected indirectly. Unlike lighttpd, proftpd (the application that has the second highest coverage) has a high percentage of error-logging paths and NULL-returning functions, but has a lower percentage of functions that can be protected indirectly.

Overall, we can see that Talos has a basic coverage of 75.1% across all applications and that each technique used by Talos plays an essential role in achieving the high coverage, although each one might have a different impact on the coverage for different applications. We also find that the majority of the functions can be directly protected by Talos.

Rate of unobtrusive SWRRs. We wish to evaluate the unobtrusiveness of SWRRs over a large number of SWRRs. To do this, we perform an experiment where we enable a large number of SWRRs and test whether they result in minor, major, or no loss of functionality. To make it easy to test a large number of SWRRs, we instrument each application for in-place deployment so that we can activate each SWRR simply by changing configurations. To ensure that all the SWRRs under our test are indeed executed, we first find out which functions are executed for the major and minor functionality test inputs used in Section 4.6.1, and then randomly choose approximately 25% of the SWRRs corresponding to the executed functions to focus on in the interests of time. In total we choose 320 SWRRs across all of the applications, as shown in Table 4.6. We then individually enable each of the selected SWRRs and run the test suite for the application. If the application passes both sets of test inputs or passes the major test inputs but fails the minor test inputs, we consider that the SWRR is unobtrusive. Otherwise, we consider the SWRR is obtrusive.

The results are tabulated in Table 4.6. Column "#SWRRs" shows the number of tested SWRRs for each application. Column "Unobtrusiveness" shows the percentage of tested SWRRs that are unobtrusive. A weighted average shows that 71.3% of the SWRRs tested are

Table 4.6: Rate of unobtrusive SWRRs.

Application	**#SWRRs**	**Unobtrusive**
lighttpd	40	70.0%
apache	85	88.2%
squid	65	69.2%
proftpd	90	64.4%
sqlite	40	55.0%
AVERAGE	64	71.3%

unobtrusive, and thus preserve the major functionality of the application. No application had a rate of unobtrusive SWRRs below 50% indicating that the majority of SWRRs are unobtrusive.

While one might believe that the rate of unobtrusive SWRRs is a function of the choice to use SWRRs to disable entire functions or the use of indirect protection, our analysis of some of the results indicates that this is not a major factor. Rather, if the vulnerability is located in the core functionality of an application, it is unlikely that disabling code, even at a finer granularity, will preserve the major functionality of the application. Thus, the main factor for unobtrusiveness is the location of the vulnerability, which is out of Talos' control. Essentially, our findings indicate that commonly executed code tends to have a higher rate of error-handling code, meaning there are more SWRRs located in commonly executed code with major functionality.

In combining the average basic coverage with the average rate of unobtrusive SWRRs, we arrive at an effective coverage of 53.5%, which gives the percentage of potential vulnerabilities that have an unobtrusive SWRR. This is a significant 2.1$\times$ improvement over the 25.2% coverage currently offered by configuration workarounds.

4.6.3 Overhead

When SWRRs are instrumented for in-place deployment, they can incur runtime overhead because they will check whether their corresponding configuration is activated at runtime every time the function into which they are instrumented is executed. When SWRRs are instrumented

Table 4.7: Overhead of SWRR.

Application	**LOC**	**Added LOC**	**#Files**	**#Modified Files**	**Overhead**
lighttpd	46,792	1.9%	79	92.4%	0.6%
apache	135,856	2.2%	191	75.9%	2.3%
squid	70,407	2.4%	119	84.0%	1.5%
proftpd	69,808	2.9%	64	93.8%	1.2%
sqlite	153,020	0.8%	2	100%	1.0%
AVERAGE	95,176	2.0%	91	89.2%	1.3%

for patch-based deployment, there is no additional runtime overhead because there is no such check. Table 4.7 gives the overhead of SWRRs for in-place deployment, measured by the number of lines of source code added by Talos and the number of corresponding source files modified by Talos. Column "App." shows the name of the application. Column "LOC" and "#Files" show the number of lines of code and the number of original source files, respectively. Column "Added LOC" shows the percentage of the lines of source code added by Talos, and column "#Modified Files" shows the percentage of corresponding source files modified by SWRRs. Column "Overhead" shows the runtime performance overhead of SWRRs. The last row shows the average for all columns.

On one hand, we can see that Talos adds on average 2% more lines of source code to implement SWRRs in applications. Given the high coverage achieved by Talos, this indicates that Talos has a very small footprint for each SWRR. On the other hand, the percentage of source files changed by Talos in order to add SWRRs, is on average 89%. This indicates that the functions protected by SWRRs are distributed among most of the source files.

To measure the runtime performance overhead of SWRRs, we use standard benchmarks for each application if a standard benchmark is available, otherwise we write our own benchmark. For each application, we compare the performance of a version of the application that is hardened by SWRRs with a version that is not. We run each benchmark three times for each application and use the average of the three measurements. To have a fair comparison, we run the hardened version of each application with all SWRRs disabled, which has the same functionality of the original application but with the added execution of the SWRRs.

For web servers including lighttpd and apache, we use ApacheBench [3]. For the squid cache proxy, we also use ApacheBench, but we enable the use squid as web proxy in its settings. We use the throughput as the performance metric for these three applications. Roughly SWRRs reduce their throughput by 2%.

For ftp servers including proftpd, we use the ftp benchmark included with pyftpdlib [22], which measures the transfer rate for both file uploads and downloads. SWRRs reduce the transfer rate for file uploads by only 1.2%, and have a negligible impact on file downloads.

For sqlite, we created our own benchmark, which is based on the description of a series of SQL database performance tests on sqlite's official web site [13]. It consists of over 70,000 SQL commands to create table, drop table, insert data, update data, query data, delete data, and perform database transactions. The benchmark measures the total execution time of all these SQL commands on sqlite database tables containing from 10,000 to 25,000 records of data. SWRRs incur a performance overhead of 1.0% on sqlite. On average SWRRs have a very small runtime performance overhead of 1.3% for all five applications.

4.7 Discussion

We begin by discussing the the limitations of SWRRs and then other operational issues associated with the deployment of SWRRs.

4.7.1 Limitations

The ability of SWRRs to neutralize vulnerabilities without security violations is limited by the assumption that applications correctly implement error-handling code. Naturally, this is not the case – applications developers may fail to identify and handle errors, or even if they do handle them, they may handle them incorrectly, as previous work has shown [61, 136]. Unfortunately, there is little that Talos can do if the error-handling code it calls contains bugs. We hope, as previous work has also implored, that developers should pay more attention to the correctness of error-handling code. While it is not invoked very often, when unexpected errors arise, it is the last defense the application has against catastrophic failures.

As an alternative to using existing error handling code, it is possible for Talos to add a new error type exclusively for SWRRs and instrument new error-handling code into applications to handle this new error type. However, it is a more intrusive approach as the new error handling code will need to be instrumented for any caller functions on the call chain to a function instrumented with an SWRR to handle the new error type properly. It will also need information from application developers on how to handle the new error type. We leave this as a future work.

Another obvious limitation is that Talos has no control over where vulnerabilities occur. As illustrated in lighttpd (CVE-2012-5533) and squid (CVE-2009-0478), if the vulnerability occurs in a key function that is used in many operations, then the availability of the application will be severely impacted. Fortunately, this appears to be the less common case (only 3 out of 11 cases in our experiments). We speculate that this is likely due to bugs and vulnerabilities occurring in less commonly executed code, as that code receives fewer opportunities for testing and has less chance of having a bug triggered in production use.

Currently Talos does not leverage the structured exception handling that is used in programming languages such as C++ and Java. However, Talos can be easily extended to do so, because it is even easier for Talos to locate error handling code utilizing structured exception handling. Talos can look for a type of exception that can be safely used to abort the execution of a function and generate an SWRR that throws the exception as the mechanism to prevent the execution of the function. To identify the exception that can be used, Talos can examine which exception is caught by existing exception handlers in the function or which exception is thrown by the function. If Talos cannot locate this kind of exception in the function itself, it can look for it in the callers of the function.

4.7.2 Other Issues

Another question is whether SWRRs and their use can decrease the security of an application in other ways, or whether the SWRRs themselves can be circumvented by an attacker even when activated. For example, even if the user activates an SWRR, an attacker can still corrupt the value of an SWRR option and re-enable the vulnerable code. While this is possible, we believe it sufficiently raises the bar for the attacker, as she must have a memory corruption vulnerability that is not in the function(s) disabled by the activated SWRR(s). In those cases, it is likely a zero-day unknown vulnerability, which requires more effort for an attacker to find and exploit. Given the nature of most memory corruption vulnerabilities, it would be likely that an attacker who has access to such a vulnerability would just use it to compromise the application directly rather than use it to disable an SWRR.

In the rare instance that a memory corruption vulnerability doesn't allow remote code execution but can still corrupt an SWRR option, the attacker now has the ability to activate or deactivate SWRRs, allowing them to re-enable disabled functions or disable enabled ones. As discussed above, they could thus silently re-enable vulnerabilities, or they could prevent code from being executed if the application has no known vulnerabilities. However, as we have shown in this chapter, activated SWRRs generally do not cause security vulnerabilities, and only impact availability. Thus, the most the attacker can do is to cause a denial of service attack with a memory corruption vulnerability – which is something they could likely already do even if SWRRs were not present.

4.8 Summary

This chapter describes the design and implementation of Talos, a system that enables safe and precise SWRRs to protect software vulnerabilities from being exploited by attackers. Our main conclusion is that SWRRs are a rapid, secure, and low-cost solution to enable applications to continue to be used until a patch becomes available. To arrive at this conclusion we test 320

SWRRs in five real world applications and find that the majority of them are unobtrusive and that 75.1% of potential vulnerabilities can be disabled by an SWRR. This indicates that SWRRs can be effective in 2.1$\times$ more vulnerabilities than traditional configuration workarounds. We also reproduce 11 vulnerabilities and their exploits and try them on the applications with and without SWRRs instrumented by Talos. We find that in all 11 cases, the security of the application is upheld and that in 8 cases, the applications retains either all or most of its functionality (with the exception of the vulnerable code).

We view Talos as a first step towards addressing the pre-patch vulnerability window. Given its simple implementation and conservative assumptions, we find these results encouraging. We believe the best avenue for improving the effectiveness of SWRRs is improving the identification of error-handling code or other safe code paths that SWRRs can redirect execution to, which will give SWRRs better basic coverage and thus also increase their effective coverage.

Chapter 5

Semantically Correct Patch Generation for Vulnerabilities

Chapter 4 presents SWRR, a mechanism to preventing vulnerabilities from being triggered. SWRR is designed as a rapid response to vulnerabilities, in order to address the problem of pre-patch window described in Chapter 3. SWRR trades off rapid response with functionality loss. To tackle its drawback of causing functionality loss, we propose an approach to automatically generating security patches that fix vulnerabilities without causing functionality loss. We consider this approach and SWRR complement to each other because they have different applicability on vulnerabilities. In this chapter, we present the details of the approach and our prototype implementation.

5.1 Introduction

Fixing security vulnerabilities in a timely manner is critical to protect users from security compromises and to prevent vendors from losing user confidence. A recent study has shown that creating software patches is often the bottleneck of the process of fixing security vulnerabilities [62]. As a result, an entire line of research inquiry into automated patch generation has arisen to try to address this challenge [57, 70, 82, 83, 89, 95, 101, 102, 128, 129].

```
char *foo_malloc(x,y) {
  return (char *)malloc(x * y + 1);
}

// print a flattened array in
// 2-dimensional format
int foo(char *input) {
  // input format: RRCCDDD....DDD
  //   RR: number of rows
  //   CC: number of columns
  //   DDD...DDD: flattened array data
  //
  // benign input: 0203123456
  // output:
  //      1 2 3
  //      4 5 6

  int size = strlen(input);
  char *p = input;
  int rows = extract_int(p);
  p +=2;
  size -= 2;
  int cols = extract_int(p);
  p +=2;
  size -=2;
+ if ((double)(cols+1)*(size/cols) >
+       rows * (cols + 1) + 1)
+   return -1;
  char *output = foo_malloc(rows, cols + 1);
  if (!output)
    return -1;
  bar(p, size, cols, output);
  printf("%s\n", output);
  free(output);
  return 0;
}

void bar(char *src,int size,int cols,char *dest) {
  char *p = dest;char *q = src;
  while (q < src + size)  {
    for (unsigned j = 0; j < cols; j++)
      *(p++) = *(q++);
    *(p++) = '\n';
  }
  *p = '\0';
}
```

Listing 5.1: A buffer overflow CVE-2012-0947 with a patch (prefixed with '+').

Automatic patch generation approaches broadly break down into two categories: search-based and semantics-based. Search-based approaches try various arbitrary code changes and use a battery of test cases to check whether any of the changes succeeded in fixing the bug [70, 82, 129]. Because search-based techniques can generate unconstrained code changes, they are applicable to a wide variety of bugs, but the correctness of the generated patches can be uneven, and depend heavily on the comprehensiveness of the test cases [92, 102]. In contrast, semantics-based techniques use code analysis (e.g. static analysis and symbolic execution) to produce patches that attempt to address the underlying bug rather than change the code just enough to satisfy the test cases [89, 95]. As a result, semantics-based techniques are more likely to produce correct patches, but are less applicable, since they are constrained to cases where the code analysis is able to analyze the defective code.

When used to fix security vulnerabilities, the requirements that patches work correctly is even more pressing, as falsely believing that a vulnerability has been addressed when in fact it has not, may lead a user to disable other mitigating protections, such as removing configuration workarounds or firewall rules. In this chapter, we propose Senx, which aims to create semantically correct patches for common security vulnerabilities, such as buffer overflows, integer overflows and memory corruption due to bad-offset calculations using semantics-based analyses. However, many of these vulnerabilities involve complex code structures that code analysis techniques traditionally find challenging. For example, buffer overflows often involve complex loop structures. In addition, formulating a check to test a memory access is within the memory-range of the data-structure, may require the patch generation system to synthesize interprocedural code if the allocation of the memory and the faulty memory access occur in different functions.

To concretely illustrate these challenges, consider the buffer overflow vulnerability CVE-2012-0947 [24] from libav in Listing 5.1. The vulnerability arises because the code copies user-provided data into a buffer whose allocation size is computed based on the number of rows and columns specified in the input, but the amount of data copied is based on the size of the data provided. For reference, the correct patch is provided on lines 26-29, which consists of checking if the amount of data to be copied (`(cols+1)`$\times$`(size/cols)`) by the nested loop in `bar` is greater than size of the buffer (`rows`$\times$`(cols+1)+1`) allocated by `foo_malloc`, in which case the patch returns an error to `foo`'s caller. To generate this patch, Senx must correctly identify the pointer `p` used to write to the buffer `dest` in `bar` and infer the memory access range of `p` from the nested loops. Further, Senx must detect that the allocation of `dest` is actually performed in another function `foo_malloc` and symbolically compute its allocation size. Finally, Senx must identify `foo` as the common caller of both `bar` and `foo_malloc` and translate the expression for both the memory access range of `p` from `bar` and the allocation size of `dest` from `foo_malloc` into the scope of `foo` so that the patch can be generated.

Senx accomplishes this with the introduction of three novel patch generation techniques. First, rather than try to statically analyze arbitrarily complex loops, Senx attempts to clone the loop code to be used in the patch predicate, but slicing out any code that may have side effects from the cloned loop so that the loop only computes the memory access range of the pointers in the loop. We call this technique *loop cloning*. For loops where code that has side effects that cannot be safely removed – for example, the loop execution depends on the result of a function call, which Senx cannot safely slice out – Senx falls back on a symbolic analysis technique we developed, called *access range analysis*. Finally, to identify and place the patch in a function scope where all expressions required in the predicate are available, Senx uses *expression translation*, which uses the equivalence between function arguments and parameters to generate a set of equivalent expressions. This enables Senx to generate patches where the defective code is spread across multiple functions. This overcomes a limitation of all previous semantics-based patch generate systems that we are aware of, which can only generate patches for defects that are enclosed entirely within in a single function [70, 82, 89, 95].

Because Senx creates patches for security vulnerabilities, Senx places a higher emphasis on correctness than previous systems. Unlike previous patch generation systems that rely on test cases to determine the correctness of the patch, Senx explicitly identifies instances where its analysis may be incorrect and aborts patch generation in those cases. In our experiments, Senx generates 32 correct patches out of 42 vulnerabilities, and in the remaining 9 cases, correctly detects that it will not be able to generate a patch and aborts instead of generating a patch that does not actually fix the vulnerability. We call this property of Senx's patches *semantic correctness*.

This chapter makes the following main contributions:

- We describe the design of Senx, an automatic patch generation system for buffer overfow, integer overflow and bad-offset vulnerabilities.
- We propose three novel program analysis techniques: loop cloning, access range analysis and expression translation.
- We describe a systematic approach to extract source code expressions of a program by decompiling the binary of the program dynamically. Senx implements its own symbolic ISA that is optimized for translating low level instructions back into source code expressions, so that source code patches can be easily generated. Particularly, Senx supports decompilation of complex expressions involving array indicies and C/C++ structs and classes.
- We describe the implementation of a Senx prototype on top of the LLVM [120] framework and KLEE [46] symbolic execution engine. We plan to make Senx open-source so that others may build on our work or deploy it to automatically create security patches.
- We evaluate Senx on a corpus of 42 vulnerabilities across 11 popular applications, including PHP interpretor, sqlite database engine, binutils utilities for creating and managing binary programs, and various tools or libraries for manipulating graphics/media files. Senx generates correct patches in 32 of the cases and aborts the remainder because it is unable to determine semantic correctness in the other cases. The evaluation demonstrates that all three techniques are required to generate patches, and that failure to find a common function scope in which to place a patch is the most frequent reason for failure.

5.2 Problem Definition

We begin by defining the types of vulnerabilities Senx can currently handle below, as well as what a Senx patch is and what semantically correct means.

Integer overflow. An integer overflow occurs when an integer is assigned a value larger or smaller than can be represented. This manifests as a large value that is increased and becomes a small value, or a small value that is decreased and becomes a large value. Integer overflows usually become vulnerabilities when the integer is subsequently used as an index into an array, which enables an attacker to corrupt or access arbitrary memory locations. Senx handles integer overflows that lead to memory corruptions.

Buffer overflow. A buffer overflow occurs when series of memory accesses traversing a buffer in a loop crosses from an allocated buffer to a memory location outside of the buffer. We use the term *buffer* in the broad sense to refer to either a bounded memory region (such as a struct or class object) or an array. The memory access can be the result of an array dereference or pointer dereference. Senx handles both the case where the memory access exceeds the upper range of the buffer and when falls below the lower range (sometimes called a buffer underflow).

Bad Offset. A bad offset vulnerability occurs when a memory access is computed off a base pointer, and exceeds the upper bound of the object the base pointer points to. Some causes of bad offset vulnerabilities include an incorrect calculation of an array index or a casts of a pointer to the incorrect object type. A bad offset vulnerability may also be called a non-linear buffer overflow in the literature.

Patch. Senx uses vulnerability condition [45] to capture the semantic of correct patches. A vulnerability condition denotes the program state when a vulnerability will be triggered. Senx generates patches in one of two forms: a) a single `if` statement that checks a *predicate*, which evaluates a vulnerability condition and is satisfied if and only if the vulnerability is about to be triggered or b) an added *data type cast* to an evaluation such as addition or multiplication that could otherwise lead to an integer overflow. In the first form, if the predicate evaluates to true, control is transferred to error handling code that avoids executing the vulnerable code, treats the current input as an error, and returns the application back to a known state. In the second form, the cast avoids integer overflows by ensuring that the evaluation is performed using the correct data type.

Semantic correctness. We say a patch is semantically correct if it prevents the execution of the vulnerable code if and only if inputs that will trigger the vulnerability is given to the program. In other words, the patch predicate must only evaluate to true on all vulnerability-triggering inputs while evaluating to false on all other inputs. For a given vulnerability condition, the patch generated by Senx is both sound and complete.

5.3 Design

5.3.1 Overview

Senx generates a patch in the following five steps. First, Senx perform a dynamic decompilation of the program using the vulnerability-triggering input. During this decompilation, Senx uses a custom *expression builder* to generate expressions involving program variables.

Second, Senx uses the results of the execution to classify the vulnerability according to the three definitions given in Section 5.2. Specifically, if any variable had an integer overflow during the execution, then the vulnerability is classified as an integer overflow. If it is not an integer overflow, Senx inspects the symbolic variables to see if an out-of-bounds memory access occured outside of an allocated region. If the out-of-bounds access occurs in a loop and is dependent on the number of iterations of the loop, it is classified as a buffer overflow. Otherwise, Senx checks if the out-of-bounds access is an offset of a base pointer, and if so classifies it as a bad offset.

Third, based on the type of vulnerability, Senx either generates the patch predicate that will detect if the input given to the program will trigger the vulnerability or identifies the correct data type to cast to a evaluation to avoid integer overflow. The predicate or cast is generated from a template specific to the type of vulnerability identified in the previous step.

Fourth, Senx finds a location in the source code where error handling code exists and all the terms of the predicate are available. In some cases, the predicate may contain variables from different function scopes, so that a single statement generated using those variables could not be evaluated at any one place in the program. For example, in Listing 5.1, the size of buffer `dest` is defined by the an expression over the variables `x` and `y` in `foo_malloc`, which are not available in the scope of function `bar` where the loop that overflows occurs. In these cases, Senx uses *expression translation* to translate those variables into expressions that are in the scope of a common caller or callee of the functions. For example, in Listing 5.1, Senx recognizes that funciton `foo` is a common caller of both `foo_malloc` and `bar` and translates all the terms of the predicate into expressions over variables available in the scope of `foo`.

Finally, Senx synthesizes a patch that checks the patch predicate and calls the error handling code if the predicate evaluates to true. Senx uses Talos [62] to find and select the error handling code to call. In each of steps, Senx will not generate a patch if it cannot guarantee semantic-correctness. For example, if Senx is unable to properly identify the vulnerability, cannot generate a predicate, or cannot successfully place the patch it will halt and not generate a patch. During decompilation, it also performs a reachability analysis combined to ensure that the definitions checked in the predicate also reach the point where the vulnerability occurs. If this is not the case, Senx cannot be sure that the predicate checked at the patch holds at the point of the vulnerability and also does not generate a patch. Senx's reachability analysis is enhanced with DSA alias analysis [77] to ensure that there are no aliases that might alter a reaching definition. The pointer analysis returns a confidence category that can be either "must alias", "must not alias" or "may alias". To be conservative, Senx treats "may alias" the same as "must alias".

Table 5.1: Pseudo instructions supported by Senx.

Instruction	**Semantic**	**Description**
val = Load var val = Load $*p$	$val \leftarrow var$ $val \leftarrow *p$	read from var read via pointer p
Store $var, val1$ Store $*p, val1$	$p \leftarrow val1$ $*p \leftarrow val1$	write $val1$ to var write via pointer p
val = GetElement $var, field$ val = GetElement $var, index$	$val \leftarrow$ StructOp$(var, field)$ $val \leftarrow$ ArrayOp$(var, index)$	read from a struct field read from an array element
val = BinOp $val1, val2$ val = CmpOp $val1, val2$	$val \leftarrow$ BinOp$(val1, val2)$ $val \leftarrow$ CmpOp$(val1, val2)$	binary operations comparisons
val = Allocate $size$	$val \leftarrow$ Allocate$(size)$	allocate a local variable
Branch $label$ Branch $cond, label1, label2$	$PC \leftarrow label$ $PC \leftarrow label1$ if $cond$ $PC \leftarrow label2$ if $\neg cond$	unconditional branch conditional branch
val = Call $f(a, \ldots)$	$val \leftarrow f(a, \ldots)$	call function f with $a, \ldots$
Ret $val1$	$val \leftarrow val1$ && $caller.val \leftarrow val1$	return *val1* to caller

5.3.2 Expression Builder

We leverage dynamic decompilation to build expressions used for synthesizing a patch for a target program. While we base our decompilation engine on KLEE [46], we do not use the symbolic representation that KLEE uses as it is heavily tied to maximizing path exploration, and does not store enough information to easily translate expressions back into source code to construct patches. As a result, we design our own symbolic representation for a set of pseudo instructions defined in Table 5.1.

The instructions include `Load` and `Store` memory access instructions, `BinOp` binary operations such as arithmetic operations and `CmpOp` comparison operations such as $>$ and $\geq$, `StructOp` struct operations that access a field of a struct, `ArrayOp` array operations that access an element of an array, `Allocate` for local variable allocation, `Branch` for unconditional and conditional branches, `Call` for function calls and `Ret` for function returns. Each instruction can have an optional label denoted as `label`. The decompilation uses a program counter that points to the current instruction, which is referred to as `PC` in the table. For each instruction presented in Column "Instruction", the decompilation interprets it using the semantic operation indicated

Table 5.2: Rules for building expressions.

Instruction	**Rule to build expression**

val = Load var
val = Load $*p$
	RHS := getExpr(var)
	RHS := makeDeref(getExpr(p))

Store $var, val1$
Store $*p, val1$
	RHS := getExpr($val1$), LHS := var
	RHS := getExpr($val1$), LHS := makeDeref(p)

val = GetElement $var, field$
val = GetElement $var, index$
	RHS := makeStructOp($var, field$)
	RHS := makeArrayOp($var, index$)

val = BinOp $val1, val2$
val = CmpOp $val1, val2$
	RHS := makeBinOp(getExpr($val1$), getExpr($val2$))
	RHS := makeCmpOp(getExpr($val1$), getExpr($val2$))

val = Allocate $size$	RHS := getName(val)
Branch $label$ Branch $cond, label1, label2$	N/A N/A
val = Call $f(a, \ldots)$	RHS := makeCall(getName(f), getExpr(a), . . .)
Ret $val1$	RHS := getExpr($val1$) && $caller$.RHS := getExpr($val1$)

in Column “Semantic”. The results of these operations are stored in Single Static Assignment (SSA) form such that each instruction instance has a unique variable associated with it. The execution makes not distinction between registers and memory.

Source code expressions are generated using the rules defined in Table 5.2. Based on the rules presented in Column“Rule to build expression”, the expression builder builds one or more semantic expressions for the corresponding instruction. Each semantic expression is of the form LHS := RHS, where LHS and RHS are the left side and right side of an assignment respectively. An explicit LHS is used only for Store instructions. The LHS for all other instructions is the SSA value associated with each instruction. The decompilation maintains a call stack so that each *Ret* instruction sets a value in its caller, denoted by `caller`, with the return value.

The expression builder uses several helper functions as described in Table 5.3. Each of these functions generate an expression according to their description. For example, `makeDeref("p")` returns `"*p"`, where $*$ represents pointer dereference. In keeping with SSA, the expressions generated for an instruction are stored along with the instruction. In this way, the expressions associated with an instruction can easily be retrieved by referring to the instruction.

Table 5.3: Operations performed by expression builder.

Operation	**Description**
getExpr	get the expression associated with an instruction or the name of a variable
getName	get the name of a variable
makeDeref	build an expression to denote dereference
makeBinOp	build an expression to denote a binary operation including *arithmetic* operations, bitwise *logic* operations, and bitwise *shift* operations
makeCmpOp	build an expression to denote a comparison including $<, >, =, \neq, \geq, \leq$
makeStructOp	build an expression to denote an access to a struct field directly or via a pointer
makeArrayOp	build an expression to denote an access to an array element
makeCall	build an expression to denote a function call including the name of the function and all the arguments

Complex Data Types. Because the patch generated by Senx is in the form of the source code of a target program, the expressions must conform to the proper language syntax of the program.

Expressions for simple data types such as char, integer, or float, are generated in a rather straightforward way. However, expressions for complex data types such as C/C++ structs and arrays are more challenging. For example, a field of a struct must be attached to its parent object, and the generated syntax changes depending on whether the parent object is referenced using a pointer or with a variable holding the actual object. Arrays and structs can also be nested and the proper syntax must be used to denote the level of nesting relative to the top level object.

To address the challenge, we include the `GetElement` instruction, which reads a field from a struct or an element from an array, in Senx's symbolic instruction set. The expression builder leverages the `GetElement` instructions and debug symbols that describe the ordered list of struct fields to construct expressions denoting access to complex data types including arrays and structs. `GetElement` is overloaded, but since the expression builder maintains the data type of each variable, it calls the appropriate version based on the type passed in `var`. To generate valid C/C++ code for a symblic expression, it retrieves the variable expression associated with `var`. If `var` is an array, it uses the helper function `makeArrayOp`, which recursivly generates code associated with the `index` argument. If `var` is a struct, it calls the helper function `makeStructOp` , which returns the name of the field in the struct. To determine whether an access to the struct is via a pointer or directly to an object, it checks witherh `var` is a result of a `Load` instructino or not, and generates the expression accordingly.

In order to build expressions for complex data access involving nested complex data types, both `makeArrayOp` and `makeStrutOp` use the expression for the variable `var`, which can be the result of a previous `Load` instruction or `GetElement` instruction. In this way, expressions for complex data access such as `foo→f.bar[10]`, where `foo` is a pointer to a struct that has a field `f` and `bar` is an array belonging to `f`, can be constructed.

5.3.3 Loop Cloning

To generate a predicate for a buffer overflow vulnerability, Senx must compare the memory range a loop may read or write to with the size of the buffer being read or written. The latter is extracted by the expression builder, so we focus on loop cloning and access range analysis to describe how the memory range of a loop is calculated. Both loop cloning and access range analysis are functions in Senx that take as input a function `F` in the program and an instruction `inst` that performs the faulty access in the buffer overflow and returns the symbolic memory access range $[A_1, A_n]$ of `inst`. This symbolic access range can then be converted into source code and compared with the allocated buffer range in the predicate.

The key idea of loop cloning is to produce new code that can be called safely at runtime to return the access range without causing any side-effects, i.e. changing program state or affecting program input/output. The new code is constructed from existing code, referred to as *cloning*, and will be called at a location where the buffer range is available so that the access range returned by the new code can be compared against the buffer range.

Because the patch must be inserted into a function where both the access range and buffer

```c
int decode(const char *in, char *out) {
  int i;
  char c;
  i = 0;
  while ((c = *(in++)) != '\0') {
    if (c == '\1')
      c = *(in++) - 1;
    out[i++] = c;
  }
  return i;
}

char* udf_decode(const char *data, int datalen) {
  char *ret = malloc(datalen);
  if (ret && !decode(data+1, ret)) {
    free(ret);
    ret = NULL;
  }
  return ret;
}
```

Listing 5.2: A complex loop involving a complex loop exit condition and multiple updates to loop induction variable on multiple execution paths.

```c
1+  void decode_clone(const char *in, char *out, char **start, char **end) {
2     char c;
3+    *start = in;
4     while ((c = *(in++)) != '\0') {
5       if (c == '\1')
6         c = *(in++) - 1;
7     }
8+    *end = in;
9   }
10
11  char* udf_decode(const char *data, int datalen) {
12    char *ret = malloc(datalen);
13+   char *start, *end;
14+   decode_clone(data+1, ret, &start, &end);
15    if (ret && !decode(data+1, ret)) {
16      free(ret);
17      ret = NULL;
18    }
19    return ret;
20  }
```

Listing 5.3: A cloned and sliced loop that no longer contain any statements that have side-effects and returns the number of iterations. Statements prefixed with '+' are added or modified by Senx to count and return the number of loop iterations.

range are available, loop cloning first searches on the call chain that leads to `F` to find such function. The search starts from the immediate caller of `F` and stops at the first function $\texttt{F}_p$ in which the buffer range is available.

If no such function can be found, Senx will not be able to generate a patch. If such function is found, loop cloning then clones each function $\texttt{F}_i$ along the call chain from `F` until $\texttt{F}_p$ into the new code that returns the access range. As a result, each $\texttt{F}_i$ is either a direct or indirect caller of `F` or `F` itself.

Loop cloning needs to satisfy two requirements: 1) `F` must compute the access range and pass the access range to its caller; 2) any direct or indirect caller of `F` must pass the access range that it receives from its callee upwards to the next function along the call chain. Each $\texttt{F}_i$ is cloned using the following steps.

1. Loop cloning clones the entire code of $\texttt{F}_i$ into $\texttt{F}_i\texttt{\_clone}$.

2. Using program slicing, it removes all statements that are not needed in order to computer the access range or pass the access range to $\texttt{F}_p$. If $\texttt{F}_i$ is `F`, it retains statements on which the execution of `inst` is dependent. If $\texttt{F}_i$ is a direct or indirect caller of `F`, it retains statements on which the call to `F` is dependent.

3. It changes the return type of $\texttt{F}_i\texttt{\_clone}$ to `void` and removes any return statement in $\texttt{F}_i\texttt{\_clone}$.

4. It adds two output parameters `start` and `end` to $\texttt{F}_i\texttt{\_clone}$. If $\texttt{F}_i$ is `F`, it inserts statements immediately before the (nested) loops to copy the initial value of the pointer or array index used in the faulty access into `start`, and statements immediately after the loops to copy the end value of such pointer or array index into `end`. If $\texttt{F}_i$ is a caller of `F`, it changes the call statement to include the two output parameters in the list of call arguments.

After cloning each $\texttt{F}_i$, loop cloning inserts a call to the last cloned function into $\texttt{F}_p$, which returns the access range in `start` and `end`. Subsequently a patch will be synthesized to leverage the returned access range.

To see how loop cloning works, consider the example in Listing 5.2, which presents a loop adopted from a real buffer overflow vulnerability CVE-2007-1887 [20] in PHP, a scripting language interpreter. The buffer overflow occurs in function decode. The loop features a complex loop exit condition and multiple updates to loop induction variable in that depend on the content of the buffer that in points to. The result of loop cloning is shown in Listing 5.3.

Loop cloning is invoked with decode as F, and the faulty access at line 5 as inst. It first finds that function udf_decode is on the call chain to decode and in which the buffer range is available. Because udf_decode directly calls decode, it needs to clone decode only.

It then clones function decode into decode_clone, after which it applies program slicing to decode_clone with line 5 and variable c and in that are accessed at line 5 as the slicing criteria. decode also has a potential write buffer overflow at line 8, but in this example, we focus on generate a predicate that will check whether in can exceed the end of the buffer it is pointing to. The program slicing uses a backward analysis and removes all statements that are irrelevant to the value of c and in at line 5, including line 2, 4 and 8.

After program slicing, it changes the return type of decode_clone into void and removes all return statements. And it adds two output parameters start and end to the list of parameters of decode_clone.

Then it inserts a statement at line 3 to copy the initial value of in to start before the loop and a statement at line 8 to copy the end value of in to end after the loop. Finally it inserts into function udf_decode a call to decode_clone at line 14 and a statement to declare start and end at line 13.

5.3.4 Access Range Analysis

Access range analysis takes as input a function f and a memory access instruction *inst* in f, and outputs the range of the memory access $[A_1, A_n]$ as a pair of expressions.

Using LLVM's built-in loop canonicalization functionality [48], access range analysis computes the access range of normalized loops. Loop canonicalization seeks to convert the loop into a standard form with a pre-header that initializes the loop iterator variable, a header that checks whether to end the loop, and a single backedge. Extracting the access range for a single loop in this way is fairly straight forward. The main difficulty is extending this to handle nested loops.

Access range analysis is implemented for nested loops using the algorithm described in Algorithm 1. It analyzes the loops enclosing *inst* starting with the innermost loop and iterating to the outermost, accumulating increments and decrements on the loop induction variables including the pointer used by *inst*.

Since the loop in `bar` of Listing 5.1 can be normalized, we use it as an example of how Algorithm 1 can be applied to a nested loop. So f is `bar` and *inst* is the memory write using pointer `p` at line 42. For each loop, it first retrieves the loop iterator variable and the bounds of it by calling helper function `find_loop_bounds`, and the list of induction variables of the loop along with the *update* to each of them, which we refer to as the fixed amount that is increased or decreased to an induction variable on each iteration of the loop, by calling another helper function `find_loop_updates`. In our example, we have $iter = \texttt{j}, initial = 0, end = \texttt{cols}$ and $\texttt{j} \mapsto 1, \texttt{p} \mapsto 1, \texttt{q} \mapsto 1$ in *updates* for the innermost `for` loop from lines 41-42.

Algorithm 1 then symbolically accumulates the update to each induction variable to a data structure referred to by *acc*, which maps each induction variable to an expression denoting the accumulated update to the induction variable. As for the example, it will store $\texttt{j} \mapsto 1, \texttt{p} \mapsto 1, \texttt{q} \mapsto 1$ into *acc* for the innermost for loop. After that, it synthesizes the expression to denote the total number of iterations for the loop. At line 16 of the algorithm, we will have $count = \texttt{cols}$ which is simplified from `(cols-0)/1`.

Having the total number of iterations, it multiplies the accumulated update for each induction variable by the total number of iterations. So *acc* will have $\texttt{j} \mapsto \texttt{cols}, \texttt{p} \mapsto \texttt{cols}, \texttt{q} \mapsto \texttt{cols}$ after the loop from line 18 to 22 in Algorithm 1.

Once this is done, it moves on to analyze the next loop enclosing *inst*, which in Listing 5.1 is the while loop enclosing the inner for loop. As a consequence, we will have

Algorithm 1 Finding the access range of a memory access.

```
Input: $f$: a function
       $inst$: a memory access instruction
Output: $acc\_initial$: initial address acccessed by $inst$
        $acc\_end$: end address accessed by $inst$
 1: procedure ANALYZE_ACCESS_RANGE
 2:     ▷ $acc$: accumulated updates to induction variables
 3:     $acc \leftarrow \emptyset$
 4:     $innermost\_loop \leftarrow innermost\_loop(inst)$
 5:     $outermost\_loop \leftarrow outermost\_loop(inst)$
 6:     $visited \leftarrow \emptyset$
 7:     for $l \in [innermost\_loop, outermost\_loop]$ do
 8:         $iter, initial, end \leftarrow$find_loop_bounds$(f, l)$
 9:         $updates, visited \leftarrow$find_loop_updates$(l, visited)$
10:         ▷ Symbolically add up induction updates
11:         for $var, upd \in updates$ do
12:             $acc\{var\} \leftarrow$ sym_add$(acc\{var\}, upd)$
13:         end for
14:         ▷ Symbolically denote the number of iterations of $l$ as $count$
15:         $upd\_iter \leftarrow updates\{iter\}$
16:         $count \leftarrow$sym_div(sym_sub$(end, initial), upd\_iter)$)
17:         ▷ Symbolically multiply induction updates by the number of iterations of $l$
18:         for $var, upd \in acc$ do
19:             if ¬is_initialized_in_last_loop$(var)$ then
20:                 $acc\{var\} \leftarrow$sym_mul$(acc\{var\}, count)$
21:             end if
22:         end for
23:     end for
24:     $ptr \leftarrow$get_pointer$(inst)$
25:     $first\_inst \leftarrow$loop_head_instruction$(outermost\_loop)$
26:     ▷ Find the definition of $ptr$ that reaches $first\_inst$
27:     $acc\_initial \leftarrow$reaching_definition$(f, first\_inst, ptr)$
28:     $acc\_end \leftarrow$sym_add$(acc\_initial, acc\{p\})$
29:     return $acc\_initial, acc\_end$
30: end procedure
```

$iter =$ `q`, $initial =$ `src`, $end =$ `src+size` and `p` $\mapsto 1$ in *updates* at line 10 of the algorithm, `j` $\mapsto cols$, `p` $\mapsto cols + 1$, `q` $\mapsto$ `cols` in *acc* and $count =$ `size/cols` at line 17 of the algorithm, and finally `j` $\mapsto$ `cols`, `p` $\mapsto$ `(cols+1)*(size/cols)`, `q` $\mapsto$ `size` in *acc*. Note that the algorithm will not multiply the number of iterations of the loop to `j` because `j` is always initialized in the last analyzed loop, the innermost for loop.

After analyzing all the loops enclosing *inst*, the algorithm gets the pointer *ptr* used by *inst* and performs reaching definition dataflow analysis to find the definition that reaches the beginning of the outermost loop. As for the example, we will have $ptr =$ `p` and the assignment `p=dest` at line 39 of `bar` as the reaching definition for `p`. From this reaching definition, it extracts the initial value of `p`, $acc\_initial =$ `dest`. Finally it gets the end value of `p`, $acc\_end =$ `dest+(cols+1)*(size/cols)` by adding the initial value `dest` to the accumulated update of `p`, `(cols+1)*(size/cols)` from *acc*. Hence it returns [`dest,dest+(cols+1)*(size/cols)`] as the expressions denoting the access range $[A_1, A_n]$.

5.3.5 Expression Translation

When generating predicates, sometimes the buffer allocation and size is computed in one function scope, while the memory access range or bad offset is computed in a different function scope. However, since the patches Senx generates are source code patches, the predicate of the patch must be evaluated in a single function scope. One possible solution is to send the expression valid in a source function scope as a call argument to a destination function scope where the expression is not valid. This approach requires adding a new function parameter to the the destination function, and adding a corresponding call argument at every call site of the destination function. We decided not to use this approach because it requires code changes to any function on the call chain from the source function to the destination function. In addition, unrelated functions that call any of the changed functions will also have to be changed, resulting in a very intrusive patch.

Expression Translation solves this problem by translating an expression exp_s from the scope of a source function f_s to an equivalent expression exp_d in a scope of a destination function f_d. It does not need adding new function parameters nor call arguments like the aformentioned solution. Senx uses expression translation to translate both the buffer size expression and memory access range expression into a single function scope where the predicate will be evaluated. We call this process *converging* the predicate.

At a high level, expression translation can be considered as a form of lightweight function summarization [60]. While function summarization establishes the relations between the inputs to a function and the outputs of a function, expression translation establishes the relations between the inputs to a function and a subset of the local variables of the function. It works by exploiting the equivalence between the arguments that are passed into the function by the caller and the parameters that take on the argument values in the scope of the callee. Using this equivalence, expression translation can iteratively translate expressions that are passed to function invocations across edges in the call graph. Formally, expression translation can converge the comparison between a expression exp_a , the symbolic memory access location in f_a and exp_s, buffer size expression in f_s iff along the set of edges $\mathbb{E}$ connecting f_a and f_s in the program call graph, an expression equivalent to either exp_a or exp_s form continuous sets of edges along the path such that exp_a and exp_s can be translated along those sets into a common scope.

Note that variables declared by a program as accessible across different functions such as global variables in C/C++ do not require the substitution, although the use of such kind of variables is not very common. We refer to both function parameters and these kind of variables collectively as nonlocal variables. And we refer to an expression consists of only nonlocal variables as a nonlocal expression.

The low-level implementation of expression translation in Senx consists of two functions. One, `translate_se_to_scopes`, identifies all candidate functions along the call stack of a function to translate a particular expression to. For example, in Listing 5.1, it would translate the arguments to `malloc` at line 2 to the scope of it's caller `foo` at line 29, and repeatedly do this for `foo`'s caller. `translate_se_to_scopes` relies on a helper function `make_nonlocal_expr`, which for each scope, translates a local expression into an equivalent expression that consists only of references to nonlocal expressions (i.e. global variables or function parameters). Together, these two functions produce equivalent expressions for every caller in a function's call stack.

Function `translate_se_to_scopes` listed in Algorithm 2 is the core of expression translation. It translates an expression *expr* to the scope of each function on the call stack *stack*. We illustrate how it works with the code in Listing 5.1. For simplicity, we use source code line numbers to represent the corresponding instructions.

Algorithm 2 Translating an expression to the scope of each function on the call stack.

Input: *stack*: a call stack consists of an ordered list of call instruction
expr: the expression to be translated
inst: the instruction to which *expr* is associated

Output: *translated_exprs*: the translated *expr* in the scope of each caller function on the call stack

```
1: procedure TRANSLATE_SE_TO_SCOPES
2:     ▷ Translate $expr$ to an expression in which all the variables are the parameters of $func$
3:     $func \leftarrow$get_func($inst$)
4:     $expr \leftarrow$make_nonlocal_expr($func, inst, expr$)
5:     if $expr \neq \varnothing$ then
6:         for $call \in stack$ do
7:             ▷ Substitute each parameter variable in $expr$ with its correspondent argument used in $call$
8:             $expr \leftarrow$substitute_parms_with_args($call, expr$)
9:             $func \leftarrow$get_func($call$)
10:            $translated\_exprs[func] \leftarrow expr$
11:            $expr \leftarrow$make_nonlocal_expr($func, call, expr$)
12:            if $expr = \varnothing$ then
13:                break
14:            end if
15:        end for
16:    end if
17:    return $translated\_exprs$
18: end procedure
```

To translate the buffer size involved in the buffer overflow, Senx finds that the buffer is allocated from a call to `malloc` at line 2 from the call stack that it associates with each memory allocation, and invokes `translate_se_to_scopes` with *stack* =[line 29], *expr* =“`x*y+1`”, *inst* =line 2, $func$ = `foo_malloc`. The function first converts “`x*y+1`” into a definition in which variables are all parameters of `foo_malloc`, which we call a nonlocal definition, if such conversion is possible. This conversion is done by function `make_nonlocal_expr` listed in Algorithm 3, which tries to find a nonlocal definition for each variable in *expr* and then substitutes each variable with its matching nonlocal definition. `make_nonlocal_expr` relies on `find_nonlocal_def_for_var`, which recursively finds reaching definitions for local variables in a function, eventually building a definition for them in terms of the function parameters, global variables or the return values from function calls. Note that a nonlocal definition can only be in the form of an arithmetic expression without involving any functions. In this case, the resulting *expr* is also “`x*y+1`” because both `x` and `y` are parameters of `foo_malloc`.

Algorithm 3 Making a nonlocal expression.

```
Input:  f: a function
        inst: an instruction in f
        expr: the RHS expression associated with inst
Output: nonlocal_expr: the nonlocalized expr
 1: procedure MAKE_NONLOCAL_EXPR
 2:     ▷ mapping stores the nonlocal definition for each variable within expr
 3:     mapping ← $\emptyset$
 4:     for var ∈ expr do
 5:         if ¬ is_var_nonlocal(f, var) then
 6:             def ←find_nonlocal_def_for_var(f, inst, var)
 7:             if def = $\varnothing$ then
 8:                 ▷ We cannot find a nonlocal definition for var
 9:                 return $\varnothing$
10:             else
11:                 mapping[var] ← def
12:             end if
13:         end if
14:     end for
15:     ▷ Substitute the occurrence of each variable with its nonlocal definition
16:     nonlocal_expr ←substitute_vars(expr, mapping)
17:     return nonlocal_expr
18: end procedure
```

It then iterates each call instruction in *stack*, starting from line 29. For each call instruction, it substitutes the parameters in *expr* with the arguments used in the call instruction. For line 29, it substitutes `x` with `rows` and `y` with `cols+1`, respectively, by calling helper function `substitute_parms_with_args`. As a consequence, "`x*y+1`" becomes "`rows*(cols+1)+1`". Hence it associates "`rows*(cols+1)+1`" with function `foo` and stores the association in *expr_translated*, because line 29 exists in function `foo`. After that, it tries to convert "`rows*(cols+1)+1`" into a nonlocal definition in respect to `foo`. At this point, it halts because both `rows` and `cols` are assigned with return values of calls to function `extract_int`. Otherwise, it will move on to the next function on the call stack and continue the translation upwards the call stack. However, in this case, expression translation is also able to translate the memory access range expression from the scope of `bar` into the scope of `foo`. Thus, Senx uses expression translation to place the patch predicate in `foo`.

5.4 Implementation

We have implemented Senx as an extension of the KLEE LLVM execution engine [46]. Like KLEE, Senx works on C/C++ programs that are compiled into LLVM bitcode [120].

We re-use the LLVM bitcode execution portion of KLEE, and as described in Section 5.3.2, to implement our own decompilation engine.

For simplicity and ease of debugging, we represent our expressions as text strings. To support arithmetic operations and simple math functions on expressions, we leverage GiNaC, a C++ library designed to provide support for symbolic manipulations of algebra expressions [58].

We implement a separate LLVM transformation pass to annotate LLVM bitcode with information on loops such as the label for loop pre-header and header, which is subsequently used by access range analysis. This pass relies on LLVM's canonicalization of natural loops to normalize loops [48]. We extend LLVMSlicer [29] for loop cloning. To locate error handling code, we use Talos [62].

Our memory allocation logger uses KLEE to interpose on memory allocations and stores the call stack for each memory allocation. Senx extends KLEE to detect integer overflows and incorporates the existing memory fault detection in KLEE to trigger our patch generation. For alias analysis, Senx leverages DSA pointer analysis [77].

Senx is implemented with 2,543 lines of C/C++ source code, not including the Talos component used to identify error handling code. Half of the source code is used to implement expression builder, which forms the foundation of other components of Senx.

5.5 Evaluation

First, we evaluate the effectiveness of Senx in fixing real-world vulnerabilities. Second, we manually examine the produced patches for correctness and compare them to the developer created patch. For the sake of space, we only describe two of the patches in detail. Last, we measure the applicability of loop cloning, access range analysis, and expression translation using a larger dataset.

5.5.1 Experiment Setup

We build a corpus of vulnerabilities for Senx to attempt to patch by searching online vulnerability databases [10,12,17], software bug report databases, developers' mailing groups [7,19,23], and exploit databases [18]. We focus on vulnerabilities that fall into one of the three types of vulnerabilities Senx can currently handle. We then select vulnerabilities that meet the following three criteria: 1) an input to trigger the vulnerability is either available or can be created from the information available, 2) the vulnerable application can be compiled into LLVM bitcode and executed correctly by KLEE, and 3) the vulnerable application uses `malloc` to allocate memory as Senx currently relies on this to infer the allocation size of objects. Applications that use custom memory allocation routines are currently not supported by Senx. We obtain

Table 5.4: Applications for testing real-world vulnerabilities.

App.	**Description**	**SLOC**
autotrace	a tool to convert bitmap to vector graphics	19,383
binutils	a collection of programming tools for managing and creating binary programs	2,394,750
libming	a library for creating Adobe Flash files	88,279
libtiff	a library for manipulating TIFF graphic files	71,434
php	the official interpretor for PHP programming language	746,390
sqlite	a relational database engine	189,747
ytnef	TNEF stream reader	15,512
zziplib	a library for reading ZIP archives	24,886
jasper	a codec for JPEG standards	30,915
libarchive	a multi-format archive and compression library	158,017
potrace	a tool for tracing bitmap graphics	20,512
Total	N/A	3,817,268

the vulnerability-triggering inputs or information about such inputs from the blogs of security researchers, bug reports, exploit databases, mailing groups for software users, or test cases attached to patch commits [1, 2, 8, 18, 26, 30].

From this, we construct a corpus of 42 real-world buffer overflow, integer overflow and bad-offset vulnerabilities to evaluate the effectiveness of Senx in patching vulnerabilities. The vulnerabilities are drawn from 11 applications show in Table 5.4, which include 8 media and archive tools and libraries, PHP, sqlite, and a collection of programming tools for managing and creating binary programs. The associated vulnerabilities consit of 19 buffer overflows, 13 integer overflows, and 10 bad-offset vulnerabilities.

All our experiments were conducted on a desktop with 4-core 3.40GHz Intel i7-3770 CPU, 16GB RAM, 3TB SATA hard drive and 64-bit Ubuntu 14.04.

5.5.2 How Effective is Senx in Patching Vulnerabilities?

For each vulnerability of an application, we run the corresponding program under Senx with a vulnerability-triggering input. If Senx generates a patch, we examine the patch for correctness. To determine if a patch is correct, we apply the three following tests a) we check for semantic equivalence with the official patch released by the vendor if available and semantic correctness by analyzing the code, b) we apply the patch and verify that the vulnerability is no longer triggered by the input and c) we check as best we can that the patch does not interfere with regular operation of the application by using the application to process benign inputs. If Senx aborts patch generation, we examine what caused Senx to abort.

Our results are summarized in Table 5.5 and Table 5.6. Column "Type" indicates whether the vulnerability is a ① buffer overflow, ② bad-offset, or ③ integer overflow. Column "Expressions" shows whether Senx can successfully construct all expressions that are required to synthesize a patch, as some code constructs may contain expressions outside of the theories Senx supports in its symbolic ISA. "Loop Analysis" describes whether loop cloning or access range analysis (ARA) is used if the vulnerability contained a loop. "Patch Placement" lists the type of patch placement: "Trivial" means that the patch is placed in the same function as the vulnerability and "Translated" means that the patch must be translated to a different function. "Data Access" describes whether or not the patch predicate involves complex data access such as fields in a struct or array indicies. Finally, "Patched?" summarizes whether the patch generated by Senx fixes a vulnerability. The 10 vulnerabilities where Senx aborts generating a patch are highlighted in red.

Over the 42 vulnerabilities, Senx generates 32 (76.2%) patches, all of which are correct according to our 3 criteria. Of the 14 patched buffer overflows, loop analysis is roughly split between loop cloning and access range analysis (6 and 8 respectively). Senx elects not to use loop cloning mainly due to two causes. First, due to an imprecise alias analysis that does not distinguish different fields of structs correctly, the program slicing tool utilized by Senx may include instructions that are irrelevant to computing loop iterations into slices. Unfortunately these instructions calls functions that can have side-effects so the slices cannot be used by Senx. Second, for a few cases the entire body of the loops is control dependent on the result of a call to a function that has side-effects. For example, the loops involved in CVE-2017-5225 are only executed when a call to `malloc` succeeds. Because `malloc` can make system calls, Senx also

Table 5.5: Patches generation by Senx

CVE ID	Type	Expressions	Loop Analysis
sqlite-CVE-2013-7443	③	Determinate	—
sqlite-CVE-2017-13685	③	Determinate	—
zziplib-CVE-2017-5976	①	Determinate	Cloned
zziplib-CVE-2017-5974	③	Determinate	—
zziplib-CVE-2017-5975	③	Determinate	—
Potrace-CVE-2013-7437	②	Determinate	—
libming-CVE-2016-9264	③	Determinate	—
libtiff-CVE-2016-9273	①	Indeterminate	—
libtiff-CVE-2016-9532	①	Determinate	Cloned
libtiff-CVE-2017-5225	①	Determinate	ARA
libtiff-CVE-2016-10272	①	Determinate	ARA
libtiff-CVE-2016-10092	③	Determinate	—
libtiff-CVE-2016-5102	③	Determinate	—
libtiff-CVE-2006-2025	②	Determinate	—
libarchive-CVE-2016-5844	②	Determinate	—
jasper-CVE-2016-9387	②	Determinate	—
jasper-CVE-2016-9557	②	Determinate	—
jasper-CVE-2017-5501	②	Determinate	—
ytnef-CVE-2017-9471	①	Determinate	Cloned
ytnef-CVE-2017-9472	①	Determinate	Cloned
ytnef-CVE-2017-9474	①	Determinate	Failed
php-CVE-2011-1938	①	Determinate	ARA
php-CVE-2014-3670	①	Determinate	ARA
php-CVE-2014-8626	①	Determinate	Cloned
binutils-CVE-2017-15020	①	Determinate	ARA
binutils-CVE-2017-9747	①	Determinate	Cloned
binutils-CVE-2017-12799	③	Determinate	—
binutils-CVE-2017-6965	③	Determinate	—
binutils-CVE-2017-9752	③	Determinate	—
binutils-CVE-2017-14745	②	Determinate	—
autotrace-CVE-2017-9151	①	Indeterminate	—
autotrace-CVE-2017-9153	①	Indeterminate	—
autotrace-CVE-2017-9156	①	Determinate	ARA
autotrace-CVE-2017-9157	①	Determinate	ARA
autotrace-CVE-2017-9168	①	Determinate	Failed
autotrace-CVE-2017-9191	①	Determinate	ARA
autotrace-CVE-2017-9161	②	Determinate	—
autotrace-CVE-2017-9183	②	Determinate	—
autotrace-CVE-2017-9197	②	Determinate	—
autotrace-CVE-2017-9198	②	Determinate	—
autotrace-CVE-2017-9199	②	Determinate	—
autotrace-CVE-2017-9200	②	Determinate	—

Table 5.6: Patches generation by Senx (continued)

CVE ID	Patch Placement	Data Access	Patched?
sqlite-CVE-2013-7443	Failed	—	✗
sqlite-CVE-2017-13685	Trivial	Simple	✓
zziplib-CVE-2017-5976	Translated	Complex	✓
zziplib-CVE-2017-5974	Translated	Complex	✓
zziplib-CVE-2017-5975	Translated	Complex	✓
Potrace-CVE-2013-7437	Trivial	Complex	✓
libming-CVE-2016-9264	Trivial	Simple	✓
libtiff-CVE-2016-9273	—	—	✗
libtiff-CVE-2016-9532	Trivial	Complex	✓
libtiff-CVE-2017-5225	Trivial	Simple	✓
libtiff-CVE-2016-10272	Translated	Simple	✓
libtiff-CVE-2016-10092	Translated	Simple	✓
libtiff-CVE-2016-5102	Trivial	Simple	✓
libtiff-CVE-2006-2025	Trivial	Complex	✓
libarchive-CVE-2016-5844	Trivial	Complex	✓
jasper-CVE-2016-9387	Trivial	Complex	✓
jasper-CVE-2016-9557	Trivial	Complex	✓
jasper-CVE-2017-5501	Failed	—	✗
ytnef-CVE-2017-9471	Trivial	Simple	✓
ytnef-CVE-2017-9472	Trivial	Simple	✓
ytnef-CVE-2017-9474	—	—	✗
php-CVE-2011-1938	Translated	Simple	✓
php-CVE-2014-3670	Translated	Complex	✓
php-CVE-2014-8626	Trivial	Simple	✓
binutils-CVE-2017-15020	Translated	Simple	✓
binutils-CVE-2017-9747	Translated	Simple	✓
binutils-CVE-2017-12799	Trivial	Simple	✓
binutils-CVE-2017-6965	Failed	—	✗
binutils-CVE-2017-9752	Translated	Simple	✓
binutils-CVE-2017-14745	Failed	—	✗
autotrace-CVE-2017-9151	—	—	✗
autotrace-CVE-2017-9153	—	—	✗
autotrace-CVE-2017-9156	Trivial	Simple	✓
autotrace-CVE-2017-9157	Trivial	Simple	✓
autotrace-CVE-2017-9168	—	—	✗
autotrace-CVE-2017-9191	Failed	—	✗
autotrace-CVE-2017-9161	Trivial	Simple	✓
autotrace-CVE-2017-9183	Trivial	Complex	✓
autotrace-CVE-2017-9197	Trivial	Complex	✓
autotrace-CVE-2017-9198	Trivial	Complex	✓
autotrace-CVE-2017-9199	Trivial	Complex	✓
autotrace-CVE-2017-9200	Trivial	Complex	✓

cannot clone the loops.

Senx must place 23.8% of the patches in a function different from where the vulnerability exists. This is particularly acute for buffer overflows (46.2% of cases), which have to compare a buffer allocation with a memory access range. This illustrates that expression translation contributes significantly to the patch generation ability of Senx, particularly for buffer overflows, which make up the majority of memory corruption vulnerabilities. Senx's handling of complex data accesses is also used in 48.5% of the patches, indicating this capability is required to handle a good number of vulnerabilities

Senx aborts patch generation for 10 vulnerabilities. The dominant cause for these aborts is that Senx is not able to converge to a function scope where all symbolic variables in the patch predicate are available. There is also one case (jasper-CVE-2017-5501) where Senx cannot find appropriate error-handling code to synthesize the patch. In these cases, the patch requires more significant changes to the application code that are beyond the capabilities of Senx. In other cases, Senx detects that there are multiple reaching definitions for patch predicates that it does not have an execution input for. Currently, Senx only accepts one execution path executed by the single vulnerability-triggering input. In the future we plan to handle these cases by allowing Senx accept multiple inputs to cover the paths along which the other reaching definitions exist. Finally, Senx aborts for a couple of vulnerabilities because both loop cloning and access range analysis fail.

5.5.3 Patch Case Study

Out of the 32 generated patches, we select 2 patches to describe in detail.

libtiff-CVE-2017-5225. This is a heap buffer overflow in libtiff, which can be exploited via a specially crafted TIFF image file. The overflow occurs in a function `cpContig2SeparateByRow` that parses a TIFF image into rows and dynamically allocates a buffer to hold the parsed image based on the number of pixels per row and bits per pixel. By using an inconsistent bits per pixel parameter, the attacker can cause libtiff to allocate a buffer smaller than the size of the pixel data and cause a buffer overflow.

When Senx captures the buffer overflow via running libtiff with a crafted TIFF image file, it first identifies that the buffer is allocated using the value of variable `scanlinesizein` and the starting address of the buffer is stored in variable `inbuf`. Hence it uses [`inbuf`, `inbuf + scanlinesizein`] to denote the buffer range. Senx then finds that the buffer overflow occurs in a 3-level nested loop and that the pointer used to access the buffer is dependent on the loop induction variable. Senx classifies the vulnerability as a buffer overflow.

Loop cloning fails because the loop slice is dependent on a call to `_TIFFmalloc`, which subsequently calls `malloc`. Thus, Senx applies access range analysis. Access range analysis detects that only the outer and inner-most loops affect the memory access pointer and from the extracted induction variables, computes the expression [`inbuf`, `inbuf+spp*imagewidth`] to represent the access range.

Because both the buffer range and the access range starts at `inbuf`, Senx synthesizes the patch predicate as `spp*imagewidth > scanlinesizein`. Senx then finds that `cpContig2SeparateByRow` contains error handling code, which has a label `bad`, and generates the patch as below. As the buffer allocation and overflow occur in the same function, Senx puts the patch immediately before the buffer allocation.

```
if (spp*imagewidth > scanlinesizein)
  goto bad;
```

The official patch invokes the same error handling and is placed at the same location as Senx's patch. However, the official patch checks that "`(bps != 8)`". From further analysis, we find that both patches are equivalent, though the human-generated patch relies on the semantics of the libtiff format, while Senx's patch directly checks that the loop cannot exceed the size of the allocated buffer.

libarchive-CVE-2016-5844. This integer overflow in the ISO parser in libarchive can result in a denial of service via a specially crafted ISO file. The overflow happens in function `choose_volume` when it multiplies a block index, which is a 32-bit integer, with a constant number. This can exceed the maximum value that can be represented by a 32-bit integer and overflow into a negative number, which is then used as a file offset.

Senx detects the integer overflow when it runs libarchive's ISO parser with a crafted ISO file. It generates an expression of the overflown value as as the product of 2048 and `vd→location`. Further Senx detects that the overflown value is assigned to a 64-bit variable `skipsize`, thus classifying this as a repairable integer overflow. Senx patches the vulnerability by casting the 32-bit value to a 64-bit value before multiplying:

```
– skipsize = LOGICAL_BLOCK_SIZE * vd->location;
+ skipsize = 2048 * (int64_t)vd->location;
```

The official patch is essentially identical to the patch generated by Senx. The only difference is that the official patch uses the constant `LOGICAL_BLOCK_SIZE` rather than its equivalent value 2048 in the multiplication.

5.5.4 Applicability

We evaluate how applicable of loop cloning, access range analysis and expression translation are across a larger dataset. To generate such a dataset, we extract all loops that access memory buffers and the allocations of these buffers from the 11 programs in coreutils, regardless of whether they contain vulnerabilities or not. We then apply Senx's loop analysis to all loops and find that loop cloning can be applied to 88% of the loops and access range analysis can be applied to 46% of the loops. This is in line with our results from the vulnerabilities. We measure how often expression translation is able to converge the memory access range and buffer allocation size into a single function scope, and find that it is able to do so in 85% of the cases.

We use 11 programs from the coreutils as listed in Table 5.7 to evaluate the applicability of our analysis techniques. The most common reasons for Senx's access range analysis to be aborted is that loops cannot be normalized by LLVM. For example, the number of times a loop that parses string input iterates depends on the content of the string. Such a string cannot be

Table 5.7: Programs for evaluating applicability.

Program	**Type**	**SLOC**	**LLVM bitcode**
sha512sum	data checksum	581	135KB
pr	text formatting	1,723	194KB
head	text manipulation	761	109KB
dir	directory listing	3,388	418KB
od	file dumping	1,368	237KB
ls	directory listing	3,388	418KB
base64	data encoding	238	91KB
wc	text processing	784	120KB
cat	file concatenating	495	182KB
sort	data sorting	3,251	433KB
printf	format and print data	694	198KB
AVG	N/A	1,516	230KB

symbolically analyzed by access range analysis.

To understand the reasons that can cause expression translation to abort, we try to converge the buffer size and memory access range for the loops that we could successfully analyze and tabulate the results in Table 5.8. The "Access Range" column tabulates the average percentage of functions in the loop's call stack that expression translation could translate the memory access range into and "Buffer Range" tabulates the average percentage of functions in the buffer allocation's call stack that expression translation could translate the buffer allocation size into. Finally "Converged" indicates out of all loops, what percentage could expression translation find a common function scope in which to place the patch. As we can see, it seems that the buffer allocation size frequently takes parameters that are calculated fairly close in the call stack to the allocation point, and those values are not available higher up in the call chain, thus limiting the functions scopes many of these cases could be converged to.

Table 5.8: Convergence of expression translation.

Program	**Access Range**	**Buffer Range**	**Converged**
pr	100%	10%	100%
head	100%	25%	100%
tr	86%	36%	100%
od	54%	16%	58%
cat	100%	33%	100%
dir	71%	14%	57%
ls	42%	33%	34%
base64	100%	33%	100%
md5sum	100%	33%	100%
sha512sum	97%	80%	97%
sort	91%	10%	90%
AVG.	85%	29%	85%

5.6 Summary

This chapter presents the design and implementation of Senx, a system that automatically generates patches for buffer overflow, bad offset, and integer overflow vulnerabilities. Senx can synthesize patches in one of two forms.

For a program that manifests a buffer overflow or a bad offset, Senx synthesizes a patch in the first form that uses a predicate to check whether a faulty memory access is about to occur and prevents the faulty memory access by steering the program execution to error handling code, similar to a patch written by human developers. For a program that manifests an integer overflow, Senx can synthesize a patch in the second form that adds a data type cast to avoid the integer overflow or a patch in the first form which checks if the integer overflow is imminent and invokes error handling code.

Senx leverages three novel techniques, expression translation, loop cloning, and access range analysis, to construct a patch. Enabled by the three techniques, Senx generates patches correctly for 32 of the 42 real-world vulnerabilities.

Chapter 6

SWRR v.s. Security Patch Generation

6.1 Overview

In this Chapter, we compare the two approaches that we propose to addressing software vulnerabilities: Talos, which produces and instruments SWRRs as described in Chapter 4, and Senx, which generates security patches as described in Chapter 5. We first compare the advantages and disadvantages of them qualitatively. Then we quantitatively measure their applicability on addressing software vulnerabilities.

6.2 Qualitative Comparison

With different design goals, Talos and Senx have different advantages and disadvantages. We compare their advantages and disadvantages in five criteria: security, unobtrusiveness, usability, robustness, and scalability.

Security. We define the security of SWRRs produced by Talos and security patches generated by Senx as whether they can address software vulnerabilities.

Unobtrusiveness. We consider SWRRs and security patches exhibit unobtrusiveness if they do not affect the application functionality irrelevant to the software vulnerabilities that they

Table 6.1: Advantages and disadvantages of Talos and Senx.

Approach	**Security**	**Unobtrusiveness**	**Usability**	**Robustness**	**Scalability**
Talos	Yes	Low, Medium, High	High	High	High
Senx	Yes	High	Medium	Medium	Low

aim to address. We consider SWRRs and security patches as obtrusive, if they cause loss of functionality irrelevant to the software vulnerabilities.

Usability. Talos and Senx require different sets of inputs to work. We measure their usability based on how hard it is to obtain such inputs. The more difficult to acquire inputs the lower the usability is.

Robustness. Talos and Senx employ different sets of program analysis techniques. Either by their design or the properties of the underlying techniques on which they are built, these techniques have different target problem space. We consider techniques that have larger target problem space as having higher robustness.

Scalability. We define the scalability of Talos and Senx as the degree of manual effort that is required to enable them to work with new software vulnerability types. The more manual effort is required the lower the scalability.

We summarize the results of our comparison in Table 6.1. For security, we label the result as either "Yes" or "No". For unobtrusiveness, usability, robustness, and scalability, we label the results as "High", "Medium", and "Low" based on an approximate categorization of the results.

Because both SWRRs and security patches effectively prevent software vulnerabilities from being exploited, we consider they all provide security.

For unobtrusiveness, we consider SWRRs can have a varying degree of unobtrusiveness depending on the functionality disabled by them. And we consider security patches to have high degree of unobtrusiveness because they disable vulnerable code only for malicious inputs.

For usability, we consider Talos to have higher usability than Senx because Senx further

Table 6.2: Number of functions and number of error-logging functions.

Application	Functions	Error Funcs.
libtiff-tiffsplit	333	1
libtiff-tiffcrop	622	1
libtiff-gif2tiff	453	1
binutils-nm-new	399	6
binutils-readelf	771	6
binutils-objdump	963	6
autotrace	176	1
zziplib	32	1
ytnef	53	1

requires proof-of-concept exploits or vulnerability-triggering inputs from users. Other than this requirement, Senx does not need human interference so we consider its usability as "Medium".

Besides the program analysis techniques employed by Talos, Senx employs additional program analysis techniques that target narrower problem space so we label its robustness as "Medium" rather than "High" for Talos.

For scalability, we consider Senx to have "Low" scalability because it relies on manual analysis of vulnerabilities to generate security patches. As a result, it might require further manual analysis to work with new vulnerability types. In contrast, Talos is largely vulnerability-agnostic so we consider Talos to have "High" scalability.

6.3 Quantitative Comparison

To quantitatively compare the applicability of Senx and Talos, we focus on the unobtrusiveness of the SWRRs produced by Talos and the security patches generated by Senx. We evaluate them on the same set of real-world vulnerabilities. To make it easier for the comparison, we choose to use the set of software vulnerabilities evaluated in Chapter 5. We list the applications that are introduced in that Chapter in Table 6.2.

We use the same definition of unobtrusiveness and the same experimental methodology used in Chapter 4. Briefly, we use two sets of test inputs, covering major functionality and minor functionality respectively, for each application to measure the unobtrusiveness of each SWRR. If no or only minor functionality is lost after activating an SWRR or applying a security patch, we consider the SWRR or security patch is unobtrusive. If major functionality is lost, we consider the SWRR or security patch is obtrusive.

The results are shown in Table 6.3. We categorize the vulnerabilities into four different types of applicability based on the results: 1) both Talos and Senx are applicable to a vulnerability because both the SWRR produced by Talos and the security patch generated by Senx are unobtrusive; 2) only Senx is applicable because the SWRR produced by Talos is obtrusive but the security patch generated by Senx is unobtrusive; 3) only Talos is applicable because the SWRR produced by Talos is unobtrusive but Senx cannot generate a security patch; 4) neither Talos nor Senx is applicable because the SWRR produced by Talos is obtrusive and Senx cannot generate a security patch. We label the applicability type in Column "Applicability".

As we can see, both Talos and Senx are applicable to 45.2% of the 42 vulnerabilities. For 33.3% of the vulnerabilities, only Senx is applicable. For 14.2% of the vulnerabilities, only Talos is applicable. While Senx is substantially more applicable than Talos, 14.2% of the vulnerabilities can only be addressed by Talos. Combining Talos and Senx, 90.5% of the vulnerabilities can be mitigated or fixed.

6.4 Summary

We compare the advantages and disadvantages of Talos and Senx. Qualitatively we consider that both the SWRRs produced by Talos and the security patches generated by Senx provide security. On one hand, SWRRs have a varying degree of unobtrusiveness, while the security patches consistently have high unobtrusiveness. On the other hand, Talos have higher usability and robustness and particularly higher scalability than Senx.

From a quantitative evaluation on a common set of real-world vulnerabilities, we find that for nearly half of the vulnerabilities both Talos and Senx are applicable with high unobtru-

Table 6.3: Applicability of Talos and Senx

Application	**CVE#**	**Talos**	**Senx**	**Applicability**
sqlite	CVE-2013-7443	No	No	4
	CVE-2017-13685	Yes	Yes	1
zziplib	CVE-2017-5976	Yes	Yes	1
	CVE-2017-5974	No	Yes	2
	CVE-2017-5975	No	Yes	2
Potrace	CVE-2013-7437	No	Yes	2
libming	CVE-2016-9264	No	Yes	2
libtiff-tiffsplit	CVE-2016-9273	No	No	4
libtiff-tiffcrop	CVE-2016-9532	Yes	Yes	1
libtiff-tiffcp	CVE-2017-5225	Yes	Yes	1
libtiff-tiffcrop	CVE-2016-10272	Yes	Yes	1
libtiff-tiffcrop	CVE-2016-10092	Yes	Yes	1
libtiff-gif2tiff	CVE-2016-5102	No	Yes	2
libtiff-tiffcp	CVE-2006-2025	No	Yes	2
libarchive	CVE-2016-5844	No	Yes	2
jasper	CVE-2016-9387	Yes	Yes	1
	CVE-2016-9557	Yes	Yes	1
	CVE-2017-5501	No	No	4
ytnef	CVE-2017-9471	No	Yes	2
	CVE-2017-9472	Yes	Yes	1
	CVE-2017-9474	Yes	No	3
php	CVE-2011-1938	No	Yes	2
	CVE-2014-3670	No	Yes	2
	CVE-2014-8626	Yes	Yes	1
binutils-nm-new	CVE-2017-15020	No	Yes	2
binutils-objdump	CVE-2017-9747	Yes	Yes	1
binutils-objdump	CVE-2017-12799	Yes	Yes	1
binutils-readelf	CVE-2017-6965	Yes	No	3
binutils-objdump	CVE-2017-9752	Yes	Yes	1
binutils-objdump	CVE-2017-14745	No	No	4
autotrace	CVE-2017-9151	Yes	No	3
	CVE-2017-9153	Yes	No	3
	CVE-2017-9156	Yes	Yes	1
	CVE-2017-9157	Yes	Yes	1
	CVE-2017-9168	Yes	No	3
	CVE-2017-9191	Yes	No	3
	CVE-2017-9161	No	Yes	2
	CVE-2017-9183	No	Yes	2
	CVE-2017-9197	Yes	Yes	1
	CVE-2017-9198	Yes	Yes	1
	CVE-2017-9199	Yes	Yes	1
	CVE-2017-9200	Yes	Yes	1

siveness. For the rest of the vulnerabilities, Senx is remarkably more applicable than Talos. However, Talos is applicable to a significant number of vulnerabilities where Senx is not applicable. Combining the strength of Talos and Senx, we can mitigate or fix the vast majority of the vulnerabilities. As a result, we consider that Talos and Senx are complement to each other.

Chapter 7

Clustering Configuration Settings For Error Recovery

Aside from software vulnerabilities, configuration errors are also a major cause of system unavailability. This chapter presents my work on troubleshooting and fixing configuration errors.

7.1 Introduction

Configuration errors are a leading cause of failure and unavailability for desktop applications [55]. Fixing such errors has essentially two steps: identifying the configuration settings causing the error, and replacing the faulty settings with values that fix the configuration error.

To facilitate the first step, proposals in the literature have tried to pinpoint the time the configuration error first appeared [130], used statistical anomaly detection to detect abnormal configuration settings [75, 125, 126], or used white-box dynamic analysis to find the particular configuration setting that causes the application to execute an erroneous code path [43]. Of these three approaches, only the last two try to identify the configuration setting that causes the error and even then, they only work if the error is the result of a single configuration setting. Unfortunately, this can be a serious drawback since a recent study found that a significant number of configuration errors (14.9%-34.7%) require changing more than one configuration

Application: **MS WORD 2010**
…\File MRU\Max Display: **20**

…\File MRU\Item 1: **\Path\To\ Document1**
…
…\File MRU\Item 20: **\Path\To\ Document20**

Description: The "**Max Display**" setting determines the number of recently accessed documents stored in the "**Item**" settings.

(a) MS Word

Application: **Acrobat Reader**
…/InlineAutoComplete: **true**

…/RecordNewEntries: **true**
…/ShowDropDown: **true**

Description: The "**InlineAutoComplete**" setting enables/disables the "auto complete" feature when user fills a form. The other settings specify how the "autocomplete" feature should behave.

(b) Acrobat Reader

Application: **Evolution Mail**
…/mail/display/mark_seen: **true**

…/mail/display/mark_seen_timeout: **15**

Description: When the setting "**mark_seen**" is set to true, Evolution marks an email as "seen" after the email is opened for the time specified in the setting "**mark_seen_timeout**".

(c) Evolution Mail

Figure 7.1: Examples of dependencies among configuration settings

setting to fix [135], because some configuration settings are related.

One example of related configuration settings is illustrated in Figure 7.1a: the number of "Item" settings should never exceed the value of `Max Display` setting. Microsoft Word automatically maintains this relationship. For instance, if a user reduces the maximum number of recently accessed documents from the Preference menu, Microsoft Word not only reduces the value of `Max Display` setting, but also deletes extra `Item` settings. Consequently, if the user wants to undo the effect of reducing the maximum number of recently accessed documents, both the old value of `Max Display` and the deleted `Item` settings need to be recovered.

In this chapter, we present a novel technique that uses hierarchical agglomerative clustering [119] to identify clusters of related configuration settings, relying only on the ability to observe application accesses to its configuration store, and is thus language, binary and OS independent. We implemented this technique in Ocasta, which treats applications as black-boxes and is able to work on a wide range of applications and environments.

To evaluate the effectiveness of Ocasta, we collected traces of application usage from both Windows and Linux machines ranging from 18 to 76 days in length and then use Ocasta to identify clusters of related configuration settings in 11 different application in across 4 different OS flavors. Using this data and 16 real-world configuration errors, we show that Ocasta's clustering is able to accurately identify 88.6% of the clusters of related configuration settings.

To further evaluate Ocasta, we added a simple GUI-based configuration error repair tool that, with user input, uses the clustering information from Ocasta to automatically search for and fix settings causing configuration errors. The Ocasta search tool requires the user to provide a GUI-action script that triggers the error, which it then uses to automatically search historical values of the clusters of configuration settings found by Ocasta for a fix. A screenshot of the result is recorded after each search and the user is asked to select a screenshot that shows that the symptoms of the configuration have been treated.

Configuration error repair in general is very hard and while Ocasta's proof of concept tool is able to fix the symptoms of all of our configuration errors, it cannot guarantee that the selected fix does not introduce new hidden errors, nor can it fix errors that do not have any visible symptoms. In general, studies have shown that even trained humans may fail to fix configuration errors completely, create new errors in the process troubleshooting or fixing an existing error, or have to resort to resetting the application back to its defaults to remove the symptoms of a configuration error [66]. Our evaluation demonstrates that Ocasta's method for inferring related configuration settings broadens the range of errors automated configuration error repair tools can handle by providing with clustering information. We believe that even when automated tools fail, the clustering information provided by Ocasta will still be valuable to human troubleshooters.

Our contributions are:

- We characterize the types and reasons of for relationships between configuration settings by manually inspecting and analyzing over 500 configuration settings across 11 applications.

- We present the design and prototype implementation of Ocasta, which uses black-box statistical clustering of application behavior to identify related configuration settings. Ocasta has been implemented on both Linux and Windows and evaluated on both systems using data collected from machines used by real people.

- We further evaluate the usability of Ocasta's clustering with a proof-of-concept tool that given a set of actions that recreates a configuration error, automatically searches historical values of clusters of configuration settings for a fix. We demonstrate the effectiveness of our tool against 16 real-world configuration errors. We also provide a user study showing the effectiveness of Ocasta's configuration repair tool.

We study relations between configuration settings and defining the problem solved by Ocasta in Section 7.2. We then describe Ocasta's high-level design in Section 7.3 and give implementation details in Section 7.4. We describe how we collected our traces in Section 7.5 and evaluate Ocasta in Section 7.6. Finally, we discuss related work in Section 2.2 and conclude in Section 7.7.

7.2 Problem Definition

Similar to relationships between program variables [86], relationships between configuration settings are a common, though not often documented phenomenon that applications exhibit. We begin by describing 3 representative examples of related configuration settings that we found by manually inspecting over 500 configuration settings that were accessed by 11 different Windows and Linux applications in our traces (trace statistics given in Table 7.1).

In Figure 7.1a, to control the number of documents listed in the recently opened documents list in Microsoft Word, `Max Display` limits the number of document names stored in the `Item` settings (e.g. `Item 1`, `Item 2`). In Figure 7.1b, Acrobat Reader uses `InlineAutoCompelete` to determine whether to enable the "auto complete" feature when user fills a form, while `RecordNewEntries` and `ShowDropDown` specify how the "auto complete" feature works, including whether to record user-entered data and whether to display the list of previously recorded data in a dropdown box. Finally, in Figure 7.1c, Evolution will automatically mark an opened email as "seen" after an email has been opened by the user for the time interval specified by the value of `mark_seen_timeout`, but only when `mark_seen` is set to "true". These examples illustrate that related configuration settings exist when one or more settings controls the validity or meaning of another group of settings.

Because related configuration are designed to work together, applications are likely to update related configuration settings together, in order to satisfy their relation as illustrated in our 3 examples. In addition, users tend to change related configuration settings together. For example, a user will probably set the value of `mark_seen_timeout` and change the value of `mark_seen` to "true" together, in order to enable Evolution to automatically mark an opened email. In contrast, independent configuration settings are unlikely to be changed together.

Table 7.1: Summary of trace statistics.

Trace	Days	Reads	Writes	#Keys	TTKV Size
Windows 7	42	6.76M	67.72K	4,611	85MB
Windows Vista	53	3.46M	20.5K	14,673	29MB
Windows Vista-2	18	15.08M	224.64K	1,123	6.3MB
Windows XP	25	22.80M	311.9K	14,667	24MB
Windows XP-2	32	26.76M	268.96K	19,501	46MB
Linux-1	25	91.52K	3.34K	1,660	6MB
Linux-2	84	8.15K	0.48K	35	0.1MB
Linux-3	46	52.41K	0.44K	706	0.7MB
Linux-4	64	507.07K	5.43K	751	6.4MB

Based on this intuition, Ocasta identifies the relations among configuration settings by observing the access correlations among them and uses hierarchical agglomerative clustering to group together configuration settings based on access correlations.

Limitations Ocasta has several limitations. First, independent configuration settings can be accidentally updated simultaneously and cause the hierarchical agglomerative clustering algorithm that Ocasta uses to incorrectly identify them as dependent. Similarly, partial update of dependent settings may be legal in some cases causing Ocasta to incorrectly infer that related settings should be in separate clusters. Ocasta's clustering can be tuned to handle such cases, but this tuning may require some manual intervention. Ultimately, Ocasta can only perform as well as the quality and amount of data available to it. Second, Ocasta must be able to intercept and record accesses to the individual keys where the application stores its persistent settings. We have implemented and tested such capabilities for OS-provided key-value stores like the Windows Registry and GConf in Linux. While many applications use OS-provided stores, some applications use their own files to store configurations. Thus we have also implemented custom parsers for several common file formats, such as XML, JSON, PostScript, INI and plain text.

Ocasta's proof-of-concept error repair tool has some additional limitations. First, a fix for the configuration error must exist in the application's recorded history. Our tool cannot fix applications that have always been misconfigured or where the configuration error arose due to a change in an external dependency. Second, the configuration error must occur deterministically, because our tool only performs one trial execution per historical cluster value in its search. Finally, because the user must be able to identify a fixed application from its screenshot, the configuration error must be visually observable on the display.

7.3 Overview

7.3.1 Clustering Configuration Settings

Ocasta improves configuration troubleshooting and repair by heuristically identifying clusters of related configuration settings. Ocasta abstracts configurations into key-value pairs, with the key being the name of the configuration setting and the value being the content of the setting. As we see in Section 7.4, many application configurations naturally fit into this abstraction.

It is important that the clusters of configuration settings that Ocasta extracts from observing application behavior be accurate. On one hand, extracting *undersized* clusters can create clusters that do not contain all the configuration keys necessary to fix a configuration error. Even worse, attempting to fix an error with an undersized cluster can, in some cases, break dependencies between configuration settings, leading to a non-working application configuration.

On the other hand, extracting *oversized* clusters causes unrelated configuration settings to be clustered together, and can lead to extraneous configuration changes when trying to repair errors. As an extreme example, repairs that reset an application configuration back to its defaults, or copy a configuration from a previous snapshot or a different user, essentially treat the application's configuration as a single, large, oversized cluster.

Ocasta uses the property that related configuration keys are much more likely to be modified together than unrelated keys to infer which keys are related. To determine whether keys have been modified together, Ocasta uses a sliding time window and considers all keys written within the window to have been modified together. Ocasta uses a default sliding window of 1 second, which can be increased if needed by the user. Some keys are modified very frequently, so the chances of such a key being modified concurrently with unrelated keys is high. Consequently, Ocasta only clusters together keys that are often modified together, but rarely modified individually on their own or with other keys. To do this, we define a *correlation* metric between each pair of keys:

$$Correlation = \frac{|A \cap B|}{|A|} + \frac{|A \cap B|}{|B|}$$

A and B denote the set of all writes to keys `A` and `B` respectively, and the intersection of A and B denotes the set of writes where both keys were written together. The correlation metric is maximized at 2 when both keys are always modified together and minimized at zero when both keys are never modified together. The larger the correlation, the more related the pair of keys. Note that the correlation is only defined when both keys have a non-zero number writes. Since Ocasta assumes that the application worked initially, any key that has not been modified from its initial value cannot cause a configuration error, and is thus excluded from Ocasta's search for a configuration fix.

Hierarchical agglomerative clustering [119] takes as input a set of points, distances between each pair of points, and a linkage criterion that defines how distances between clusters are computed. It then iteratively merges clusters together, forming a hierarchy with larger clusters at the top of the hierarchy. In Ocasta, we use the "maximum linkage criterion", which defines the distance between a pair clusters as the maximum distance between any two keys across the clusters. Hierarchical clustering has the advantage over other types of clustering, such as k-means or centroid-based clustering, in that it does not require the number of clusters to be specified in advance. To perform hierarchical clustering, distances need to be smaller as keys become more related, so we use the inverse of our correlation metric as the distance for Ocasta's clustering. To decide when to stop clustering, Ocasta provides a tune-able *threshold*, which defines the maximum distance between any two clusters. By default, Ocasta uses a threshold equivalent to a correlation value of 2 (i.e. a distance of 0.5), which only clusters keys that are always modified together. If the user finds that configuration repair fails due to undersized clusters, she may decrease the threshold to allow Ocasta to cluster together keys that are modified together most of the time.

Like any black-box heuristic, Ocasta can fail under certain circumstances, particularly for configuration settings that have had very few modifications from which Ocasta can learn. For example, the user may modify several unrelated settings at once, causing the application to store those changes together into its configuration store. Unless, these settings are later modified separately, Ocasta will incorrectly infer that they are related, resulting in an oversized cluster. Similarly, it is possible that a user makes a single change to an application that causes a change to only one level of hierarchically dependent configuration keys. For example, she may disable the feature completely, which would only change the higher-level key, modify the lower-level keys without changing the higher-level key, or only modify a subset of the lower-level keys. Again, if this was the only instance of modifications to the key, then Ocasta may infer an undersized cluster that separates related keys from each other into different clusters. While only using black-box information makes Ocasta more broadly applicable, Ocasta can only work with the information it observes and as a result, can be misled when there is inadequate history for its clustering to work.

7.3.2 Automated Repair

Ocasta's automated repair tool uses the clustering information to aid the user in fixing configuration errors. For example, configuration error #15, described in Table 7.3, causes the menu bar to disappear when certain PDF documents are opened in Acrobat Reader. To use Ocasta, the user must first create a *trial*, which tells Ocasta how to recreate the error and makes the symptoms of the error visible on the screen. For example, in the case of error #15, the user starts Acrobat Reader and uses it to open the PDF document that causes the error. Since the menu bar disappears once the document is opened, the error is visible on the screen. The user thus ends the trial with the menu bar missing and document open on the screen. Ocasta records the UI actions the user made in the trial and automatically extracts the identity of the application or applications that were used.

Ocasta's repair tool then asks the user to specify an optional *start time* and an optional *end time*. The start time is the earliest time the user believes the configuration error could have been introduced, and allows Ocasta to limit how far back in time it searches for the cluster that causes the error, which we call the *offending cluster*. If the user doesn't specify a bound, Ocasta will search all the cluster versions in the recorded history of the application. The end time is the latest time the user believes the configuration error could be introduced and should roughly coincide with time the configuration error is first discovered. This is useful if the user might have tried to fix the error themselves and thus may have made spurious configuration changes that might slow down the search. If the user does not specify an end time, Ocasta uses all recorded values up to the end of the recorded history.

In some cases Ocasta can identify a large number of clusters in an application (as many as 220 in our measurements). As a result, recovery will be significantly faster if Ocasta sorts clusters so that the ones that are likely to be configuration clusters are checked before the ones that are likely to be non-configuration clusters. We use the intuition that changes to configuration settings should be infrequent because for them to change, the user must explicitly modify a configuration setting, which also happens infrequently. Ocasta thus sorts the clusters by the number of times they have been modified over the application's history.

Ocasta then executes the user-provided trial on the historical values of the clusters by rolling back an entire cluster of configuration settings at a time and running the trial in a sandbox, which prevents the execution to leave any persistent changes. Ocasta can be configured to perform either a breadth-first (BFS) or depth-first (DFS) search on the historical values of each cluster. In DFS, Ocasta executes the trial on all the historical values of a cluster before moving onto the next cluster. In BFS, Ocasta executes the latest historical value of each cluster before moving onto the next historical value. DFS works well if Ocasta's sort algorithm successfully prioritizes the offending cluster early in the sort, while the BFS algorithm provides performance that is less influenced by how well the sort worked.

After each trial execution, the tool takes a screenshot. Ocasta discards the screenshot if it is identical to either the erroneous screenshot or any previous screenshots it has recorded. The user can periodically check on the recorded screenshots recorded to see if any of them display a fixed configuration. When she see a fixed configuration, Ocasta permanently rolls back the cluster to its corresponding value and returns back to recording mode. A video demonstrating the use of Ocasta is available online for viewing [1].

7.4 Implementation

In this section we describe implementation details of Ocasta's prototype. Ocasta works on both Windows and Linux. Ocasta supports applications that use the Windows registry or the GConf configuration system, as well as applications that store configuration state in XML, JSON, PostScript, INI and plain text files. We describe the implementation of the Ocasta time travel key-value store, the logger, as well as the clustering and repair components of Ocasta.

7.4.1 Time Travel Key-value Store

Ocasta records configuration key-value activity in a time travel key-value store (TTKV). We implemented Ocasta's TTKV using Redis, a commonly used key-value store [105]. Redis maps each key in the application to a record that contains the number of writes and deletions, as well as a list of historical values of the key including timestamps. A special type of value is used to represent deletions of the key, which are also recorded in the value history.

During regular application use, Ocasta's loggers (described in the next section) intercept accesses by applications to their configuration store and record information about these accesses in the TTKV. Ocasta then uses the information stored in the TTKV to compute the clustering information for the keys. In addition, Ocasta's configuration error repair tool uses historical values in the TTKV when performing its search for a configuration error fix.

[1] `http://youtu.be/aRvJlTj-0F0`

7.4.2 Logger

The primary purpose of the logger is to intercept accesses an application makes to its persistent storage and abstract those into key-values that can be stored into the TTKV. As a result, the logger is necessarily dependent on the way the application stores its application state. Below we detail the implementation of Ocasta loggers for the Windows registry, GConf configuration system, and various file formats used by the applications we tested.

Windows registry

The Windows registry is a key-value store provided by the Windows OS. Applications write keys in the Windows registry using a well-documented API provided by the OS. We implemented the Windows registry logger as a user-space shared library. To intercept registry API calls made by applications, we use the Windows debug APIs to inject the shared library into Explorer, the Windows shell. Once injected into Explorer, the shared library intercepts each Windows registry API by hooking the first five bytes of the instructions of the API call in a way similar to Detours [65]. The shared library also injects itself into new processes created by the process it is loaded into by intercepting the Windows API call that creates new processes. Virtually all regular applications are started via the Explorer shell, which implements all the common methods for starting applications such as the Start Menu, desktop shortcuts, taskbar shortcuts, or double-clicking an executable in a folder. As a result, the Ocasta logger is able to monitor every application a user uses. We note that the Windows registry logger only captures registry activity by user applications, not by system services or the Windows kernel, so our current prototype cannot fix configuration errors in those components.

GConf configuration system

The GConf configuration system, commonly found on Linux systems, implements the handlers for its APIs in a shared library. We used the standard approach of intercepting shared library calls on Linux by using the `LD_PRELOAD` environment variable to load our own shared library into the address space of every process. Our library exports a set of shared library calls that is identical to the set of shared library APIs exported by the GConf shared library. By specifying our library in the `LD_PRELOAD` environment variable, our library is always loaded before the GConf library and thus all calls to those APIs will invoke our functions, which will then subsequently call the real functions in the GConf shared library after logging the events to the TTKV.

Application-specific file formats

Applications that don't use OS-provided key-value storage facilities such as the Windows Registry or GConf generally implement their own file-based key-value store. We conducted a small study on the common file formats used for configuration storage and found applications generally use standard file format: JSON, XML, PostScript, or one of two key-value lists that both had the format "$key = value$", which we called INI if it is hierarchical and plain text if it is flat.

We elide the details of the implementation of our application-specific file parsers for the sake of space. One inherent shortcoming of Ocasta when dealing with application-specific file formats is that applications typically read the entire file into an in-memory key-value store. The applications then perform writes on the in-memory store and flush the in-memory store back to disk. To infer which keys are changed, Ocasta compares the files before and after each flush. In practice, we observe that applications typically flush their in-memory store after each key modification to guarantee persistence, but if they do not, Ocasta will not be able to tell if a key was modified several times between flushes. As shown in Section 7.6, despite the coarser level of information available to Ocasta for applications that use application-specific files, Ocasta is still able to offer good clustering performance for these applications.

7.4.3 Ocasta Clustering and Repair Tool

Ocasta's clustering algorithm is based on an open source clustering library [76]. However, the hierarchical clustering API provided by this library does not allow a cluster threshold to be used to restrict clustering. Hence, we added functionality to prune the results returned by the hierarchical clustering API according to a specified threshold.

Ocasta's repair tool has three main components – a UI record and replay tool, which records the user-provided trial and re-executes it on the application, a screenshot tool, which takes and records screenshots of the application and a controller, which coordinates the entire recovery search. We have implemented the repair tool on both Windows and Linux. To save time and effort, we made judicious use of various open-source libraries and packages for recording UI actions, as well as capturing and manipulating screenshots.

A limitation with our current implementation of the repair tool is that it deterministically replays trials and thus does not guarantee the same trial can be replayed correctly across different configuration settings. A robust adaptive replay can probably address this limitation, but the current focus of our work is to demonstrate the benefits of clustering. Nonetheless, we found our repair tool works well in our evaluation and user study.

7.5 Data Collection

We deployed Ocasta on 24 Linux desktop computers running Debian 6 and 5 Windows desktop computers. Ocasta intercepts and records reads, writes and deletions of settings into application configuration stores such as the Windows registry, GConf database and application configuration files. Configuration settings are abstracted into keys and stored into a key-value store called the Time Travel Key Value Store (TTKV). Table 7.1 summarizes the characteristics of the traces from these deployments, which we use in this chapter. The period of deployments range from one month to over two months. All the computers were actively used during the deployment.

All the Linux desktop computers are from four undergraduate computing laboratories administrated by our department. To reduce bias in the selection of the computers, we choose 6 computers from each laboratory. These computers are used mainly on site by undergraduate students for their course work, and remotely by graduate students and faculty members in our department. This study was approved by our institutional ethics review board.

Because these machines are shared among many users, we link usage of applications by the same user regardless of what machine they are using – traces from one machine by a particular user will be combined with traces from another machine by the same user. Our ethics review board required us to only instrument a fraction of the computers in any one lab to give students who did not wish to participate in the study ample opportunity to select an uninstrumented machine. Unfortunately, this meant that we only got a sampling of user-behavior since a student would not be likely to use an instrumented machine every time they were in the lab.

The 5 Windows desktop computers are personal computers used by four graduate students and one faculty member. They run a variety of Windows OS including Windows 7, Windows Vista, and Windows XP.

7.6 Evaluation

We evaluate 3 aspects of our Ocasta prototype. First, we evaluate the accuracy of the clusters that Ocasta extracts. Second, we evaluate the effectiveness and performance of Ocasta, and the benefits of using clustering at recovering from configuration errors. Finaly, we perform a user study to evaluate how easy it is for a user to generate a trial, identify the screenshot showing a fixed application, and use Ocasta in general. All Windows experiments were performed on an Intel Core Duo Dual-Core laptop with 2 GB of memory running Windows 7 and all Linux experiments were performed on a Intel Core 2 Quad-Core desktop with 4 GB of memory running Debian 6. We used 11 popular desktop applications in our evaluations, as listed in Table 7.2.

Table 7.2: Applications and their clusters Identified by Ocasta.

Application	**Description**	**#Keys**	**#Clusters**	**%Accuracy**
MS Outlook	E-mail Client	182	33/82	97.0%
Evolution Mail	E-mail Client	183	18/65	38.9%
Internet Explorer	Web Browser	33	9/12	66.7%
Chrome Browser	Web Browser	35	1/34	100%
MS Word	Word Processor	143	18/110	100%
GNOME Edit	Word Processor	10	1/7	0.0%
MS Paint	Image Editor	66	2/8	50.0%
Eye of GNOME	Image Viewer	5	0/5	N/A
Acrobat Reader	Document Reader	751	120/550	95.8%
Explorer	Windows Shell	298	32/91	84.4%
Windows Media Player	Media Player	165	21/41	90.5%
Total	N/A	1,871	255/1,005	88.6%

7.6.1 Clustering Analysis

To evaluate the accuracy of Ocasta's clustering algorithm, we manually examined all 255 clusters, each of which contains more than one configuration setting, across all applications used in our evaluations. First, we try to confirm whether configuration settings are correlated by examining their names and values. We identify relations of configuration settings from their hierarchical names [75] and verify their relations from their values. Second, we individually change configuration settings in a cluster and check whether the corresponding application runs properly after the change. We conservatively consider a cluster as correctly identified if and only if there is a dependency among every configuration setting of the cluster.

As a result, we define an *oversized cluster* as a cluster that contains one or more extra configuration settings that are not related with the other configuration settings in the cluster, and an *undersized cluster* as a cluster that does not contain one or more configuration settings that are related with the configuration settings in the cluster.

We show the accuracy of Ocasta's clustering algorithm in Table 7.2. For each application, we compute the ratio of correctly identified clusters with more than one setting over the total number of clusters with more than one setting. The result illustrates that Ocasta has a high accuracy of identifying clusters with more than one setting, 72.3% on average (mean accuracy among all applications) and 88.6% overall (ratio of the total number of correctly identified clusters to the total number of clusters across all applications). Except for four applications (Evolution Mail, Internet Explorer, Text Editor, and MS Paint) that have a very small number of clusters (smaller than 20) and a small number of configuration settings, Ocasta accurately identified clusters with more than one setting in 94% of the cases. We elaborate on our findings below.

Oversized Clusters The majority of the incorrectly identified clusters are oversized clusters, which are caused by two major sources. First, Ocasta is limited to using a minimum of one second as the sliding time window. This is because the trace collection infrastructure only records the update time of configuration settings to the precision of the nearest second. Although the 1-second sliding time window works well for most applications, one second is long enough for an application to update more than one group of dependent configuration settings. For example, one oversized cluster of Evolution Mail contains six groups of dependent configuration settings. Second, some configuration settings may be updated simultaneously as the result of software updates, in which case even independent configuration settings could be updated together.

Oversized clusters can cause unnecessary configuration settings to be changed when attempting to fix configuration errors. As a result, we want to minimize the number of oversized clusters and the number of extra configuration settings in oversized clusters. To achieve that, we examined all 17 oversized clusters of the four applications with the highest ratio of oversized clusters. We found that 11 of the oversized clusters are composed of several groups of dependent configuration settings and that the remaining 6 of them have one extra configuration setting in them. This indicates that most of the oversized clusters are probably caused by using a 1-second sliding time window and could potentially have been eliminated if our trace collection infrastructure had recorded key modification times at a finer granularity.

Undersized Clusters Ocasta's clustering algorithm can also cause undersized clusters if dependent configuration settings are not always updated together. Undersized clusters can cause failures in fixing configuration errors, since dependent configuration settings are not changed together, or leave configuration settings in an inconsistent state that can cause application misbehavior. In the next section, we describe how out of 16 injected errors, Ocasta is able to fix all but 2 using the default clustering threshold of 2 and window size of 1 second. The 2 unfixed errors are a result of undersized clusters, which we were able to correct by tuning of the clustering threshold and window size. We did not observe any application crashes or misbehavior during the hundreds of clusters that were changed during the trials executed by Ocasta to fix these errors.

7.6.2 Configuration Repair

The traces we collected contain realistic application usage, but because they are collected without interacting with the users of the applications, we are unable to confirm if configuration errors occurred during trace creation. In addition, we want to be able to precisely control the time at which the configuration error occurs in each trace. Thus, we simulate configuration errors by injecting a write into the trace at the point in time at which we want the error to occur, that changes the offending setting to the erroneous value. If the configuration error is caused by presence or absence of the offending setting, we insert or delete the setting in the trace. To simulate the recording phase of Ocasta, we populate the TTKV of the test machine with one of the traces that exhibited usage of the same application in the configuration error scenario.

We first evaluate how effective Ocasta is at fixing 16 real-world configuration errors, numbered 1-16 in Table 7.3, which are all configuration errors that were either previously used in the literature [125, 137] or were found via online forums, FAQ documents and configuration documents. To demonstrate the benefit of using clustering, we compare the effectiveness of Ocasta with the effectiveness of a modified version of Ocasta, called Ocasta-NoClust, that does not use clustering and rolls back a single configuration setting at a time when it tries to

Table 7.3: Real-world configuration errors used in our evaluation.

Case	Trace	Application	Logger	Description
1	Windows 7	MS Outlook	Registry	User is unable to use Navigation Panel.
2	Windows 7	MS Word	Registry	User loses the list of recently accessed documents.
3	Windows 7	Internet Explorer	Registry	Dialog to disable add-ons always pops up.
4	Windows Vista	Explorer	Registry	"Open with" menu does not show installed applications that can open .flv file.
5	Windows XP	Windows Media Player	Registry	Caption is not shown while playing video.
6	Windows XP	MS Paint	Registry	Text tool bar does not pop up automatically when entering text.
7	Windows XP	Explorer	Registry	Image files are always opened in a maximized window.
8	Linux-1	Evolution Mail	GConf	Evolution Mail starts in offline mode unexpectedly.
9	Linux-1	Evolution Mail	GConf	Evolution Mail does not mark read mail automatically.
10	Linux-1	Evolution Mail	GConf	Evolution Mail does not start a reply at the top of an e-mail.
11	Linux-1	Image Viewer	GConf	User is unable to print image files.
12	Linux-1	Text Editor	GConf	User is unable to save any document.
13	Linux-2	Chrome Browser	File	Bookmark bar is missing.
14	Linux-2	Chrome Browser	File	Home button is missing from the tool bar.
15	Linux-3	Acrobat Reader	File	Menu bar disappears for certain PDF document.
16	Linux-4	Acrobat Reader	File	Find box is missing from the tool bar.

fix errors.

We use as many complex and real configuration errors as possible for the evaluation. For example, error #12 was found on an internet message board, where the discussion contained 56 messages spanning 3 months. However, we are restricted to only using errors where the offending setting(s) have been modified in our traces – otherwise Ocasta will have no clustering information for them and Ocasta's repair tool will have no values to roll back to. This problem cannot happen in practice because any configuration key that is misconfigured must have a modification history on a particular system. We simulate the configuration error by injecting the erroneous value into the TTKV 14 days before the end of the trace and invoke Ocasta in recovery mode. For each error, we provide a suitable trial and set the start time to 14 days before the end of the trace. We configure Ocasta to use the DFS search strategy.

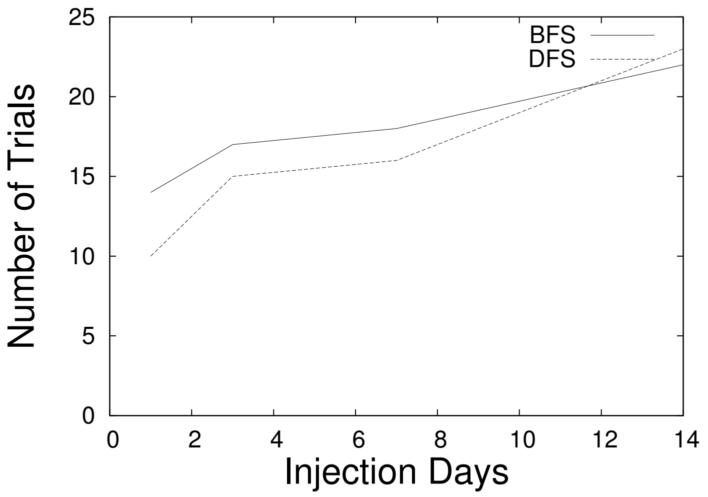
We evaluated Ocasta using the minimum window size of 1 second and the maximum correlation threshold of 2, because these produce smaller clusters and are thus the most likely to lead to invalid configurations or failed fixes. In practice, a user can adjust these settings in case they fail to cluster the configuration settings that cause the configuration problem. With these parameters, Ocasta was able to successfully find the offending cluster and fix the errors in all cases except errors #2 and #4. In both of these cases, the settings that needed to be rolled back were split into several clusters. In error #2, the offending settings consisted of one rarely-changing dominant setting, which controls the validity of another 50 settings that change frequently over a moderate span of time, as we described in Figure 7.1a. When the clustering threshold is reduced to 1, the dominant setting is clustered with 34 of the other settings, but there remain 26 settings that were not clustered together. When we increase the window size to 30 seconds, causing all settings to be clustered together. In error #4, one setting stores an ordered list of names of settings that store applications capable of opening Flash video files. The setting storing the list tends to change even when the setting storing the application name does not change. Reducing clustering threshold to 1 caused both the setting storing the list and the settings storing application names to be clustered together.

Table 7.4: Ocasta recovery performance.

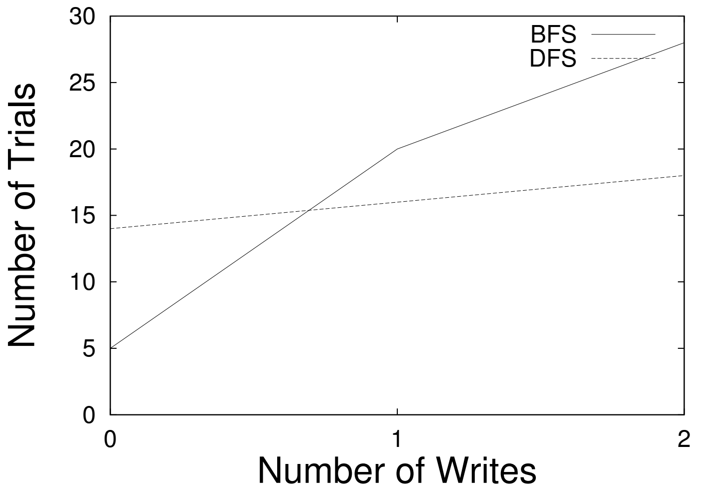
Case	Cl.Size	Trials	Time(mm:ss)	Screens	Ocasta	NoClust
1	2	15	0:30/6:00	5	Y	Y
2	8	2	0:34/1:01	1	Y	N
3	2	14	4:16/5:24	11	Y	Y
4	3	33	3:02/8:57	1	Y	N
5	4	60	5:36/28:40	1	Y	Y
6	8	8	3:04/3:30	1	Y	N
7	2	134	3:30/24:11	2	Y	N
8	2	7	1:46/2:11	2	Y	Y
9	2	9	6:52/8:32	9	Y	N
10	2	12	5:28/6:31	2	Y	Y
11	1	2	0:24/0:56	1	Y	Y
12	1	2	0:20/0:44	1	Y	Y
13	1	7	0:36/3:40	2	Y	Y
14	1	7	0:30/2:58	4	Y	Y
15	1	17	1:05/8:41	2	Y	Y
16	1	157	0:28/57:19	4	Y	Y

Quantitative results are shown in Table 7.4. We can see that Ocasta successfully fixed all 16 configuration errors, but Ocasta-NoClust failed to fix 5 configuration errors, because it requires rolling back more than one configuration settings at a time to fix them. The average cluster size varies between 1 and 8 for our errors, thus effectively reducing the search space by the same factor because Ocasta searches clusters of keys at a time instead of individual keys. The time column gives the time required by Ocasta to find the offending cluster versus the total time for Ocasta to search all cluster versions up to the 14 day start time. This shows that Ocasta's sort is successful at prioritizing the clusters, finding the offending cluster by an average of 78% faster than having to search the entire history. The screenshots column gives the total number of unique screenshots produced by Ocasta, while the trials column indicates the number of trials executed before the offending cluster is found. The user must examine an average of 3 screenshots, with a worst case of 11, indicating a very modest amount of user effort.

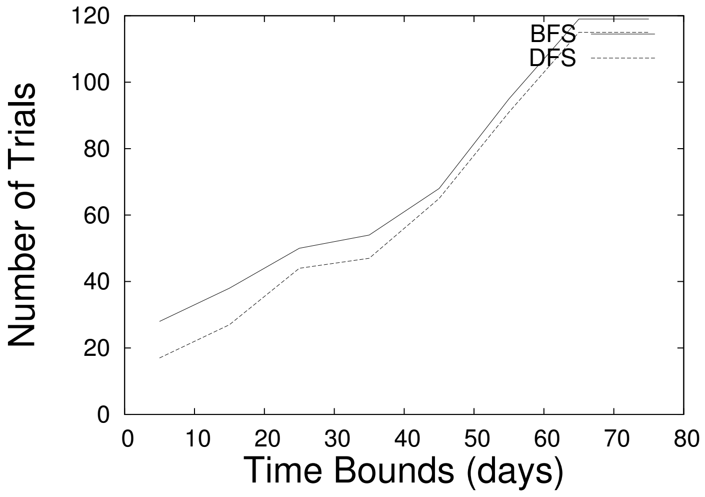
Recall that instead of using DFS, Ocasta can also use BFS as the search strategy. To study the trade-offs we perform searches using both strategies over all 16 errors while varying the number of days in the past when the error was injected, as well as fixing the injection time at



(a) By time of errors



(b) By number of spurious writes



(c) By time length

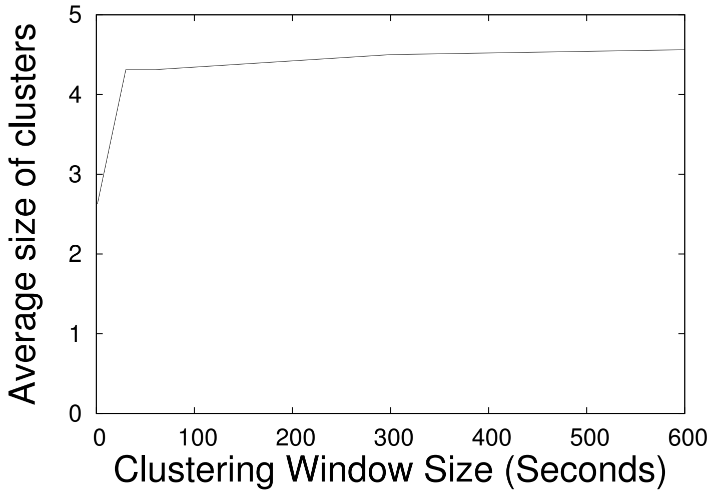
Figure 7.2: Comparison between DFS and BFS.

14 days in the past and adding between 0-2 spurious writes after the initially injected error to simulate the case where the user tried to fix the configuration error for 0-2 times. Figure 7.2a shows the average number of trial executions as a function of error injection time for BFS and DFS. As can be seen, the number of trials by both BFS and DFS increases as the injection time occurs further in the past, as a result of Ocasta's bias towards checking more recently modified clusters first, while DFS provides better performance overall. Figure 7.2b shows the average number of trials as a function of the number of spurious writes after the injected error. BFS search is highly sensitive to this parameter because to search more writes within a cluster, it must try every other cluster as well, so the number of rollbacks increases if there are a lot of clusters.

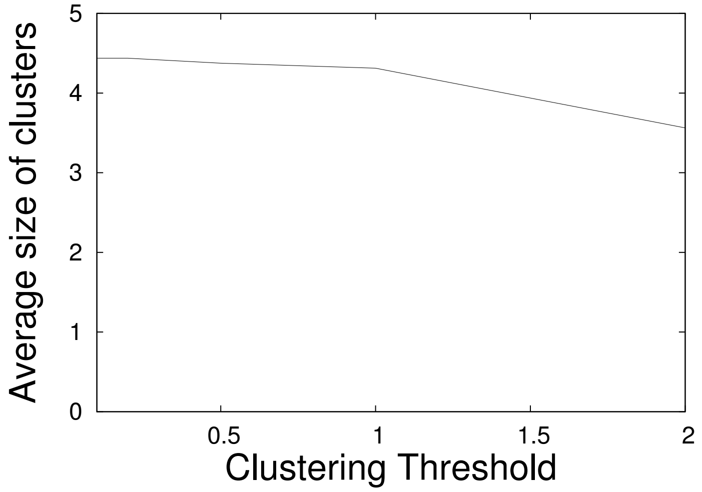
We now evaluate the effect of the start time, which controls the time period Ocasta searches over, on the number of trials Ocasta must execute. Figure 7.2c shows the average number of trials Ocasta perform in its search as start time goes further into the past. As can be seen, the number of trials rises roughly linearly with the length of time the search is conducted over.

7.6.3 Sensitivity

We examine the sensitivity of cluster size to both windows size and clustering threshold. Larger clusters mean fewer trials, but also lead to the potential for more unrelated keys getting changed if the offending cluster grows in size. Figures 7.3a and 7.3b show the growth in average cluster size as a function of both the window size and clustering sensitivity. The sharp drop at the left hand side of Figure 7.3a, is when the window is changed from one second to zero seconds (modifications must have the same timestamp at zero seconds). Since our traces only record key modification times to the nearest second, there is a lot of noise between these two points. With the exception of this artifact, the average cluster is relatively insensitive to either parameter, and ranges between between roughly 3.5 to about 4.5 or 25% of its value. These results indicate that the overall cluster size is relatively insensitive to changes in these parameters, which might suggest that users should tend to prefer smaller thresholds and larger window sizes to minimize



(a) Window size.



(b) Clustering threshold.

Figure 7.3: Average cluster size.

the chances of the offending cluster being undersized.

7.6.4 User Study

To evaluate the effectiveness of the Ocasta repair tool with default settings [2] , we performed a user study on 19 participants with various backgrounds. Because this study contains human subjects, we have obtained a second ethics approval for this study from our institutional ethics review board. The participants include two faculty members from our department, 13 graduate students from four different departments, a system administrator, an administrative assistant, and two software engineers. Six out of the 19 participants of the user study are non-technical users. None of participants were authors of this chapter and none were compensated for this user study. Each participant was given a brief explanation on how Ocasta works and shown a demonstration on a contrived configuration error. The participant then tested Ocasta on a computer setup with configuration error #11, #13, #15 and #16 from Table 7.3. We use only four errors to limit the length of the user study, because it took between 1.5 and 2 hours for each participant to finish the user study. In each case, the participants were first asked to quantitatively rate how familiar were they with the application having the configuration error. Then they were given a description of the error and were asked to use Ocasta to fix the configuration error. We recorded the time the participants took to create the trial. After they finished creating the trial, they were asked to quantitatively rate how difficult it was to produce the trial.

The participant was then shown the set of screenshots Ocasta produces when run on the history from our traces and asked to select the screenshot that showed the fixed application. The time taken for the participant to select the screenshot was also recorded. After the participant selected the screenshot, we recorded whether they selected the right one. We also asked the participant how many of the screenshots they actually examined and to qualitatively rate how difficult it was to find the screenshot.

We then reset the system back to its misconfigured state and asked the participant to try to

[2] 1-second sliding time window, clustering threshold of 2, and DFS search strategy

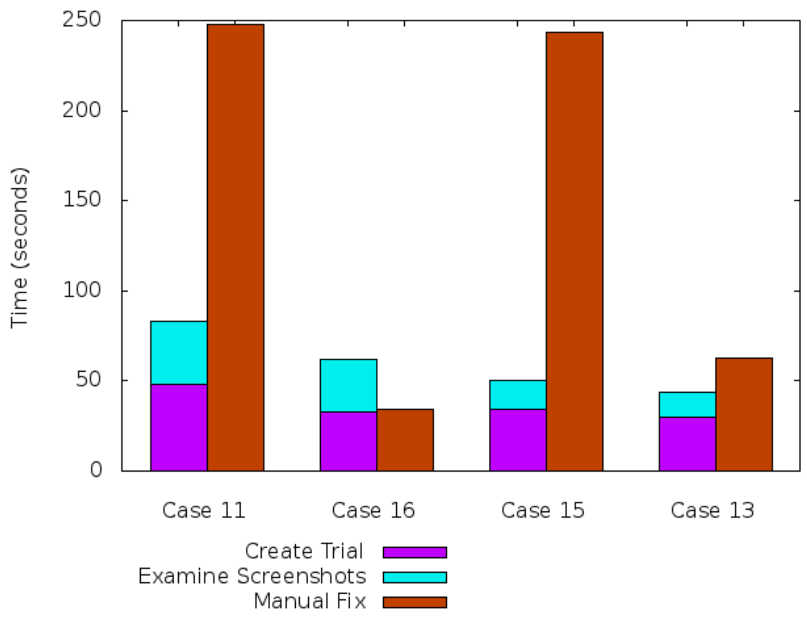


Figure 7.4: Time required to fix the error with Ocasta versus manual fixing.

fix the error manually. The participant was given full control of the computer and was allowed to use Internet to search for possible solutions to the configuration error. To keep the test short, we cut the participants off at 5 minutes. We recorded whether the participant was able to fix the error manually or not and the time it took for them to fix the error. For each error, the participant was finally asked whether they had experienced the particular error themselves before and the steps they took to fix or try to fix the error.

Figure 7.4 shows a comparison between the average time users took to both create the witness and select the screenshot and the average time taken to manually repair each configuration error. If we use the time spent as an indicator of the amount of user effort, we can see that Ocasta saves users a significant amount of effort to repair configuration errors. Only in case 16 were the majority of participants able to fix the configuration error manually and this significantly lowered the average time for the a manual fix. Qualitatively on a difficulty scale of 1 to 5, with 1 being the easiest, across the 4 errors, the participants rated the creation of the trial as 1 74% of the time, 2 21% of the time and and 3 5% of the time. For selecting the correct screenshot, participants rated the difficulty as 1 80% of the time, 2 11% of the time, 3 8% of the time and 4 1% of the time.

Our user study has several sources of bias. First, selection of participants was not completely random, but consisted of colleagues and acquaintances of the authors. Second, the administration of the study was single blind and the person administrating the test knew the correct answer. To minimize this effect, we tried to minimize interaction with the participant and communicated using written materials as much as possible. Third, the participants were cut off at 5 minutes when they tried to fix the error manually, while no cut off was used for generating the Ocasta trial or selecting the screenshot. Thus, the time measurements for some of the manual fixes represent a lower-bound while the time measurements for Ocasta usage are precise. Finally, we selected errors that tended to be simple. This made it easier to explain the errors to users who might be unfamiliar with the applications. In addition, simple errors make manual fixing easier and thus make it more difficult for Ocasta to have a significant advantage over manually searching for the fix.

7.7 Summary

This chapter describes the design and implementation of Ocasta, a system that enables configuration recovery systems to handle multi-configuration setting errors by identifying clusters of related configuration settings using statistical clustering. We have evaluated Ocasta over several months on both Windows and Linux machines and find that Ocasta's clustering accurately identifies about 88.6% of clusters on average. Our evaluation of Ocasta in fixing configuration errors shows that Ocasta successfully fixed all 16 real world configuration errors used in our evaluation, 5 of which require changing more than one configuration setting together to fix, by utilizing the identified clusters of related configuration settings,

Chapter 8

Conclusions

With the wide deployment of computer systems particularly mobile devices, almost every aspect of our daily lives has become dependent on computer systems. As a result, the security and reliability of computer systems is increasingly crucial to us. However, troubleshooting and fixing software security and reliability issues is still a largely time-consuming manual task of software developers, system administrators, and computer users. Thereby there is a growing interests and need to automate this task. This dissertation addresses the challenges in the automation of troubleshooting and fixing software vulnerabilities and configuration errors: 1) generating correct security workarounds and patches; 2) finding the root cause of configuration errors.

For software security, this dissertation presents Security Workarounds for Rapid Response (SWRR) that mitigates software vulnerabilities. By leveraging existing error handling code in applications, SWRRs gracefully prevent software vulnerabilities from being exploited. By its design, SWRR is vulnerability-agnostic and is thus applicable to arbitrary software vulnerabilities. To produce and instrument SWRRs, we implement a tool called Talos that employs novel program analysis techniques to automatically identify existing error handling code in applications. This work illustrates that SWRRs can be automatically produced and instrumented and they effectively mitigate real-world software vulnerabilities with remarkable coverage.

As a vulnerability mitigation technique, SWRRs are intended to be a fast and short-term solution. In the long run, security patches are necessary to fix software vulnerabilities. This dissertation further presents our security patch generation technique and its implementation called Senx. Targeting three of the most common and severe software vulnerabilities, Senx uses original program analysis techniques to carefully generate correct security patches. We find that Senx can generate security patches for the majority of the software vulnerabilities that we evaluate.

As Talos and Senx have different design goals, we compare their strengths and drawbacks. We find that they complement each other while in general Senx can apply to substantially more vulnerabilities than Talos. Combining them together, we can address the vast majority of software vulnerabilities.

For software reliability, we present Ocasta, a tool that automatically troubleshoots and fixes configuration errors. Ocasta targets complex configuration errors involving dependent configuration options. It uses machine learning to understand the dependency among configuration options. And it employs automated GUI testing facility to significantly reduce human interference in the troubleshooting process. With a synergy of system monitoring, machine learning, automated GUI testing, and rollback recovery, Ocasta automates the troubleshooting and fixing of configuration errors. Our user study of Ocasta shows that it substantially reduces manual effort in this task.

This dissertation demonstrates that it is possible to automate two crucial tasks that improve software security and reliability: troubleshooting and fixing software vulnerabilities and configuration errors, through a combination of original program analysis techniques, machine learning, automated GUI testing, and rollback recovery.

8.1 Future Work

Producing unobtrusive SWRRs. Currently SWRRs disable vulnerable code regardless of whether the inputs to an application can actually trigger software vulnerabilities. This is the main source of the obtrusiveness of SWRRs because the functionality provided by the disabled code is even unavailable to benign inputs. Can SWRRs selectively disable vulnerable code depending on whether the inputs are malicious or benign?

A possible solution to the problem is to use program path constraints to differentiate execution contexts for malicious and benign inputs. This approach however is limited by the fact that traditionally path constraints are encoded as logical forumlas that are computationally hard to solve in the presence of large number of variables. Thus developing a more efficient representation of path constraints will be a significant step towards differentiating execution contexts for malicius inputs and benign inputs. I intend to develop such representation and leverage it in SWRRs to make them unobtrusive.

Generating security patches for new vulnerability types. My work illustrates that it is feasible to automate the development of security patches for certain types of vulnerabilities. However, the automation relies on patch strategies generated using manual analysis of these vulnerabilities. Such analysis provides a deep understanding of the cause and effect of vulnerabilities but is a time consuming task. To be able to automatically generate patches for other existing or new types of vulnerabilities, we need to create new patch strategies. How can we speed up this process?

My vision is that machine learning is the key to accomplish this task. I intend to build systems that automatically create patch strategies. First, I am interested in modeling the cause and effect of vulnerabilities in an efficient way so that they can be understood by a machine learning algorithm. Second, I would like to train the machine learning algorithm to understand the cause and effect of vulnerabilities by learning from both the code relevant to vulnerabilities and the code of their corresponding patches. Third, I plan to build systems that leverages the machine learning algorithm to automatically create patch strategies.

Bibliography

[1] agostino's blog. http://blogs.gentoo.org/ago.

[2] agostino's poc repository. http://github.com/asarubbo/poc.

[3] Apache HTTP server benchmarking tool. http://apache.org/docs/2.2/programs/ab.html.

[4] Apache httpd 2.4 vulnerabilities. http://httpd.apache.org/security/vulnerabilities_24.html.

[5] Apache httpd Vulnerability Exploit. http://www.exploit-db.com/exploits/34133.

[6] Bug 3841 - Possible symlink race when applying UserOwner to newly created directory. http://bugs.proftpd.org/show_bug.cgi?id=3841.

[7] bug-coreutils Archives. http://lists.gnu.org/archive/html/bug-coreutils/.

[8] Bugs.MapTools.Org. http://bugzilla.maptools.org/.

[9] Bugzilla For ProFTPD. http://bugs.proftpd.org.

[10] Common Vulnerabilities and Exposures. http://cve.mitre.org.

[11] Critical Patch Updates and Security Alerts. http://www.oracle.com/technetwork/topics/security/alerts-086861.html#SecurityAlerts.

[12] CVE Details. http://www.cvedetails.com.

[13] Database Speed Comparison. http://www.sqlite.org/speed.html.

[14] Debian bug tracking system. http://bugs.debian.org.

[15] lighttpd Vulnerability Exploit. http://www.exploit-db.com/exploits/18295/.

[16] Microsoft Security Bulletin. http://technet.microsoft.com/en-us/security/bulletin/.

[17] National Vulnerability Database. http://nvd.nist.gov.

[18] Offensive Security's Exploit Database Archive. http://www.exploit-db.com.

[19] PHP Bug Tracking System. http://bugs.php.net.

[20] PHP5 sqlite_udf_decode_binary() Buffer Overflow Vulnerability. http://www.php-security.org/MOPB/MOPB-41-2007.html.

[21] ProFTPD Backdoor Unauthorized Access. http://www.osvdb.org/69562.

[22] pyftpd - Extremely fast and scalable Python FTP server library. http://code.google.com/p/pyftpdlib/.

[23] Red Hat Bugzilla. http://bugzilla.redhat.com.

[24] Security issue in libav/ffmpeg. http://www.openwall.com/lists/oss-security/2012/05/03/4.

[25] SecurityTracker. http://securitytracker.com.

[26] sqlite-users. http://www.mail-archive.com/sqlite-users@mailinglists.sqlite.org/.

[27] Squid Invalid Version Number Vulnerability. http://security-tracker.debian.org/tracker/CVE-2009-0478.

[28] Squid Range Headers Vulnerability Workaround. http://www.squid-cache.org/Advisories/SQUID-2014_2.txt.

[29] Static Slicer for LLVM. http://github.com/jirislaby/LLVMSlicer.

[30] Welcome to Sourceware Bugzilla. http://sourceware.org/bugzilla/.

[31] Microsoft Security Advisory 3009008 - Vulnerability in SSL 3.0 Could Allow Information Disclosure. https://docs.microsoft.com/en-us/security-updates/securityadvisories/2015/3009008, 2015.

[32] Microsoft Security Bulletin MS15-031 - Vulnerability in Schannel Could Allow Security Feature Bypass (3046049). https://docs.microsoft.com/en-us/securitybulletins/2015/ms15-031, 2015.

[33] VMWare - Workarounds for vRealize Orchestrator Address Apache Struts Remote Code Execution Vulnerability (1034175). https://kb.vmware.com/s/article/1034175, 2015.

[34] Comcast's nationwide outage was caused by a configuration error. https://www.engadget.com/2017/11/07/comcast-internet-outage-level-3-route-leak, 2017.

[35] Equifax Breach Impacts 145.5 Million. https://https://www.identityforce.com/business-blog/equifax-breach-impacts-143-million-steps-to-keep-your-identity-protected, 2017.

[36] Samba: Remote code execution from a writable share. https://www.samba.org/samba/security/CVE-2017-7494.html, 2017.

[37] The Robert Morris Internet Worm. http://groups.csail.mit.edu/mac/classes/6.805/articles/morris-worm.html, 2017.

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

[39] O. H. Alhazmi, Y. K. Malaiya, and I. Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Comput. Secur.*, 26(3):219–228, May 2007.

[40] P. Andritsos and V. Tzerpos. Information-theoretic software clustering. *Software Engineering, IEEE Transactions on*, 31(2):150–165, 2005.

[41] N. Anquetil and T.C. Lethbridge. Experiments with clustering as a software remodularization method. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 235–255, 1999.

[42] Frederico Araujo, Kevin W. Hamlen, Sebastian Biedermann, and Stefan Katzenbeisser. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 942–953, New York, NY, USA, 2014. ACM.

[43] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, pages 1–11, October 2010.

[44] Leyla Bilge and Tudor Dumitras. Before we knew it: An empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 833–844, New York, NY, USA, 2012. ACM.

[45] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 2–16, Washington, DC, USA, 2006. IEEE Computer Society.

[46] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[47] George Candea. Toward quantifying system manageability. In *Proceedings of the 4th Workshop on Hot Topics in Systems Dependability*, December 2008.

[48] Cannoicalize natural loops. `http://llvm.org/docs/Passes.html\#loop-simplify-canonicalize-natural-loops`.

[49] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 782–791, Piscataway, NJ, USA, 2013. IEEE Press.

[50] Sandy Clark, Michael Collis, Matt Blaze, and Jonathan M. Smith. Moving targets: Security and rapid-release in firefox. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1256–1266, New York, NY, USA, 2014. ACM.

[51] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 117–130, New York, NY, USA, 2007. ACM.

[52] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 292–307, Washington, DC, USA, 2014. IEEE Computer Society.

[53] W. Dickinson, D. Leon, and A. Fodgurski. Finding failures by cluster analysis of execution profiles. In *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, pages 339–348, 2001.

[54] Nicholas DiGiuseppe and James A. Jones. Concept-based failure clustering. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 29:1–29:4, New York, NY, USA, 2012. ACM.

[55] A. Ganapathi, Yi-Min Wang, Ni Lao, and J.-R. Wen. Why pcs are fragile and what we can do about it: a study of windows registry problems. In *Dependable Systems and Networks, 2004 International Conference on*, pages 561–566, 2004.

[56] Qi Gao, Wenbin Zhang, Yan Tang, and Feng Qin. First-aid: Surviving and preventing memory management bugs during production runs. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 159–172, New York, NY, USA, 2009. ACM.

[57] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for c programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 459–470, Piscataway, NJ, USA, 2015. IEEE Press.

[58] GiNaC is Not a CAS. `http://www.ginac.de/`.

[59] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The Taser intrusion recovery system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 163–176, October 2005.

[60] Denis Gopan and Thomas Reps. Low-level library analysis and summarization. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, pages 68–81, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[61] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: Error Handling is Occasionally Correct. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, CA, February 2008.

[62] Z. Huang, M. DAngelo, D. Miyani, and D. Lie. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 618–635, May 2016.

[63] Zhen Huang and David Lie. Ocasta: Clustering configuration settings for error recovery. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 479–490, June 2014.

[64] Zhen Huang and David Lie. Senx: Semantically Correct Patch Generation for Security Vulnerabilities. *arXiv*, 2017.

[65] Galen Hunt and Doug Brubacher. Detours: Binary interception of Win32 functions. In *Proceedings of the 3rd Usenix Windows NT Symposium*, July 1999.

[66] Lon Ingram, Ivaylo Popov, Srinath Setty, and Michael Walfish. Repair from a chair: Computer repair as an untrusted cloud service. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems*, May 2011.

[67] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 389–400, New York, NY, USA, 2011. ACM.

[68] Guoliang Jin, Wei Zhang, and Dongdong Deng. Automated concurrency-bug fixing. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 221–236, Hollywood, CA, 2012. USENIX.

[69] Lorenzo Keller, Prasang Upadhyaya, and George Candea. Conferr: A tool for assessing resilience to human configuration errors. In *Proceedings of the 2008 International Conference on Dependable Systems and Networks*, pages 157–166, June 2008.

[70] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, Piscataway, NJ, USA, 2013. IEEE Press.

[71] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, pages 1–9, October 2010.

[72] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 Annual Usenix Technical Conference*, pages 1–15, April 2005.

[73] Nate Kushman and Dina Katabi. Enabling configuration-independent automation by non-expert users. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, pages 1–10, 2010.

[74] Yonghwi Kwon, Brendan Saltaformaggio, I Luk Kim, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. A2c: Self destructing exploit executions via input perturbation. In *Proceedings of NDSS'17*. Internet Society, 2017.

[75] Emre Kycyman and Yi-Min Wang. Discovering correctness constraints for self-management of system configuration. In *Proceedings of the 1st International Conference on Autonomic Computing*, pages 28–35, May 2004.

[76] M. J. L. de Hoon, S. Imoto, J. Nolan, and S. Miyano. Open source clustering software. *Bioinformatics*, 20 (9):1453–1454, 2004.

[77] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 278–289, New York, NY, USA, 2007. ACM.

[78] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 3–13, June 2012.

[79] C. Le Goues, ThanhVu Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, Jan 2012.

[80] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2201–2215, New York, NY, USA, 2017. ACM.

[81] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic input rectification. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 80–90, Piscataway, NJ, USA, 2012. IEEE Press.

[82] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015. ACM.

[83] Fan Long and Martin Rinard. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 298–312, New York, NY, USA, 2016. ACM.

[84] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. Sound input filter generation for integer overflow errors. In *Proceedings of the 41st ACM SIGPLAN-*

SIGACT Symposium on Principles of Programming Languages, POPL '14, pages 439–452, New York, NY, USA, 2014. ACM.

[85] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 227–238, New York, NY, USA, 2014. ACM.

[86] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 103–116, New York, NY, USA, 2007. ACM.

[87] O. Maqbool and H.A. Babri. Hierarchical clustering for software architecture recovery. *Software Engineering, IEEE Transactions on*, 33(11):759–780, 2007.

[88] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 448–458, Piscataway, NJ, USA, 2015. IEEE Press.

[89] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM.

[90] Microsoft. Data Execution Prevention (DEP). `http://support.microsoft.com/kb/875352/EN-US/`, 2006.

[91] Microsoft Corp. Microsoft Security Bulletin MS17-010 - Critical, 2012. `http://technet.microsoft.com/en-us/library/security/ms17-010.aspx`.

[92] Martin Monperrus. A critical review of automatic patch generation learned from human-written patches: Essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242. ACM.

[93] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: better, faster, stronger SFI for the x86. In *Proceedings of the 2012 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, PLDI '12, pages 395–404, New York, NY, USA, 2012. ACM.

[94] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 529–540, New York, NY, USA, 2007. ACM.

[95] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[96] Ben Niu and Gang Tan. Monitor Integrity Protection with Space Efficiency and Separate Compilation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 199–210, New York, NY, USA, 2013. ACM.

[97] Ben Niu and Gang Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1317–1328, New York, NY, USA, 2014. ACM.

[98] PC Magazine. Stagefright 2.0 Targets Nearly Every Single Android Device. `http://mobile.pcmag.com/networking/`

60449-stagefright-2-dot-0-targets-nearly-every-single-android-device, 2015.

[99] PC Magazine. There's (Almost) Nothing You Can Do About Stagefright. http://mobile.pcmag.com/news/58468-theres-almost-nothing-you-can-do-about-stagefright, 2015.

[100] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[101] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 254–265, New York, NY, USA, 2014. ACM.

[102] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 24–36, New York, NY, USA, 2015. ACM.

[103] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 235–248, October 2005.

[104] A. Rabkin and R. Katz. Precomputing possible configuration error diagnoses. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 193–202, 2011.

[105] Redis. `http://redis.io/`, 2012.

[106] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[107] Martin C. Rinard. Living in the comfort zone. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 611–622, New York, NY, USA, 2007. ACM.

[108] R.W. Schwanke. An intelligent tool for re-engineering software modularity. In *Software Engineering, 1991. Proceedings., 13th International Conference on*, pages 83–92, 1991.

[109] Hovafv Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, October 2004.

[110] Hossain Shahriar and Mohammad Zulkernine. Mitigating Program Security Vulnerabilities: Approaches and Challenges. *ACM Computing Surveys*, 44(3):11:1–11:46, June 2012.

[111] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 771–781, Piscataway, NJ, USA, 2012. IEEE Press.

[112] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Softw. Eng.*, 37(6):772–787, November 2011.

[113] S. Sidiroglou, G. Giovanidis, and A. Keromytis. Using execution transactions to recover from buffer overflow attacks, 2004.

[114] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: automatic software self-healing using rescue points. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 37–48, New York, NY, USA, 2009. ACM.

[115] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 43–54, New York, NY, USA, 2015. ACM.

[116] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. Autobash: Improving configuration management with operating system causality analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 237–250, October 2007.

[117] Martin Süßkraut and Christof Fetzer. Robustness and security hardening of COTS software libraries. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings*, pages 61–71, 2007.

[118] Jacob Swanson, Myra B. Cohen, Matthew B. Dwyer, Brady J. Garvin, and Justin Firestone. Beyond the rainbow: Self-adaptive failure avoidance in configurable systems. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 377–388, New York, NY, USA, 2014. ACM.

[119] Pang-Ning Tan, Michael Steinbach, Vipin Kumar, and editors. *Introduction to Data Mining*. Pearson Addison Wesley, 2005.

[120] The LLVM Compiler Infrastructure. `http://llvm.org/`.

[121] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, August 2014. USENIX Association.

[122] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 246–255, 1999.

[123] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216, 1994.

[124] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 193–204, New York, NY, USA, 2004. ACM.

[125] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 245–258, December 2004.

[126] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *Proceedings of the 17th Large Installation System Administrator Conference*, pages 159–172, June 2003.

[127] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated Fixing of Programs with Contracts. In *Proceedings of the*

19th International Symposium on Software Testing and Analysis, ISSTA '10, pages 61–72, New York, NY, USA, 2010. ACM.

[128] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 356–366, Piscataway, NJ, USA, 2013. IEEE Press.

[129] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[130] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 77–90, December 2004.

[131] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 416–426, Piscataway, NJ, USA, 2017. IEEE Press.

[132] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 307–319, New York, NY, USA, 2015. ACM.

[133] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In

Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, pages 244–259, New York, NY, USA, 2013. ACM.

[134] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93, 2009.

[135] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 159–172, October 2011.

[136] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 249–265, 2014.

[137] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. Context-based online configuration-error detection. In *Proceedings of the 2011 Annual Usenix Technical Conference*, pages 36–41, June 2011.

[138] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, L. Szekeres, S. McCamant, D. Song, and Wei Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573, May 2013.

[139] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, Washington, D.C., 2013. USENIX.

[140] Sai Zhang and Michael D. Ernst. Automated diagnosis of software configuration errors. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 312–321, Piscataway, NJ, USA, 2013. IEEE Press.

[141] Sai Zhang and Michael D. Ernst. Which configuration option should i change? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 152–163, New York, NY, USA, 2014. ACM.

[142] Wei Zheng, Ricardo Bianchini, and Thu D. Nguyen. Automatic configuration of internet services. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 219–229, New York, NY, USA, 2007. ACM.