Automatic Hardening against Dependability and Security Software Bugs

Dissertation
zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
TECHNISCHEN UNIVERSITÄT DRESDEN
FAKULTÄT INFORMATIK

eingereicht von
Dipl.-Inf. Martin Süßkraut
geboren am 08.01.1979 in Halle (Saale)

Gutachter
Prof. Christof Fetzer, PhD, Technische Universität Dresden
Prof. George C. C. Candea, PhD, Ecole Polytechnique Fédérale de Lausanne

Tag der Verteidigung: 21. Mai 2010

Dresden Juni 2010
Abstract

It is a fact that software has bugs. These bugs can lead to failures. Especially dependability and security failures are a great threat to software users. This thesis introduces four novel approaches that can be used to automatically harden software at the user’s site. Automatic hardening removes bugs from already deployed software. All four approaches are automated, i.e., they require little support from the end-user. However, some support from the software developer is needed for two of these approaches. The presented approaches can be grouped into error toleration and bug removal. The two error toleration approaches are focused primarily on fast detection of security errors. When an error is detected it can be tolerated with well-known existing approaches. The other two approaches are bug removal approaches. They remove dependability bugs from already deployed software. We tested all approaches with existing benchmarks and applications, like the Apache web-server.
Acknowledgements

I am grateful to many people for their help in doing this work. First of all, I wish to thank my family – especially my wife Birgit. Without her support I would not have had the strength and time for writing this thesis. I would like to acknowledge the debt I owe to my Advisor Christof Fetzer. He taught me most of what I know about doing research. I always enjoyed all the discussions with him that inspired me to most of this work. My colleague Ute Schiffel highly improved the quality of this thesis with her tough questions. I also want to thank her and my wife Birgit for their ability to withstand and identify my English mistakes. This thesis is based on several published papers. These papers would not have been possible without my co-authors: Christof Fetzer, Stefan Weigert, Ute Schiffel, Thomas Knauth, Martin Nowack, Diogo Becker de Brum, Stephan Creutz, and Martin Meinhold. I also wish to thank my colleagues at the Systems Engineering Group. I have learned a lot from them that is important in my job. Last but not least, I want to thank my students. By teaching them I also learned a lot. My apologies if I have inadvertently omitted anyone to whom acknowledgement is due. While I believe that all of those mentioned above have contributed to improve this work, none is, of course, responsible for any remaining weakness.
## Contents

3.2 Related Work ................................................. 50
3.3 Deterministic Replay and Speculation ......................... 52
  3.3.1 Interface ............................................. 53
  3.3.2 Implementation ........................................ 55
3.4 Switching Code Bases .......................................... 56
  3.4.1 Example ............................................... 57
  3.4.2 Integration with parexc_chkpnt ......................... 58
  3.4.3 Code Transformations ................................... 59
  3.4.4 Stack-local Variables .................................. 67
3.5 Speculative Variables ......................................... 67
  3.5.1 Interface ............................................. 68
  3.5.2 Deadlock Avoidance .................................... 69
  3.5.3 Storage Back-ends ...................................... 69
3.6 Parallelized Checkers ......................................... 69
  3.6.1 Out-of-Bounds Checks ................................... 70
  3.6.2 Data Flow Integrity Checks .............................. 71
  3.6.3 FastAssert Checker ..................................... 71
  3.6.4 Runtime Checking in STM-Based Applications ............ 72
3.7 Evaluation .................................................. 73
  3.7.1 Performance ........................................... 73
  3.7.2 Checking Already Parallelized Applications .............. 77
  3.7.3 ParExC Overhead ....................................... 78
3.8 Conclusion ................................................ 80

4 Automatically Finding and Patching Bad Error Handling ............ 83
  4.1 Related Work ............................................. 84
  4.2 Overview ................................................ 86
  4.3 Learning Library-Level Error Return Values from System Call Error Injection 89
    4.3.1 Components .......................................... 91
    4.3.2 Efficient Error Injection ............................. 91
    4.3.3 Obtain OS Error Specification ......................... 92
  4.4 Finding Bad Error Handling ................................ 92
    4.4.1 Argument Recording ................................... 93
    4.4.2 Systematic Error Injection ............................ 94
    4.4.3 Static Analysis ....................................... 96
  4.5 Fast Error Injection using Virtual Machines ................... 99
    4.5.1 The fork Approach ...................................... 100
    4.5.2 Virtual Machines for Fault Injection ................. 101
  4.6 Patching Bad Error Handling ................................ 102
    4.6.1 Error Value Mapping .................................. 103
    4.6.2 Preallocation .......................................... 105
    4.6.3 Patch Generation ...................................... 106
  4.7 Evaluation ................................................ 108
    4.7.1 Measurements .......................................... 109
1 Introduction

It is a fact that software is deployed with bugs. Barry Boehm and Victor R. Basili state that: “About 40 to 50 percent of user programs contain nontrivial defects.” These bugs can lead to failures that decrease the dependability, the availability, and the security of a system. Studies have found that 5% to 24% of all failures in deployed high-performance systems can be contributed to software.

Software failures can be catastrophic: “In the last 15 years alone, software defects have wrecked a European satellite launch, delayed the opening of the hugely expensive Denver airport for a year, destroyed a NASA Mars mission, killed four marines in a helicopter crash, induced a U.S. Navy ship to destroy a civilian airliner, and shut down ambulance systems in London, leading to as many as 30 deaths.” Additionally, software failures have an economical impact. The National Institute of Standards and Technology estimates that in the U.S. software failures cost $59.5 billion annually.

The reasons why software is deployed with bugs are many-fold. Most prominent are economical, legal, and technical reasons as well as the human factor. Software makers have the ability to remove bugs in already deployed software by patching. This ability alone is an economical motivation to release early and fix later. It is still an unresolved problem if software makers should be liable for their product’s failures. Up to now it is common practice that software makers explicitly exclude their liability for software failures in their licence agreements.

From a technical point of view a recent study suggests that the likelihood of bugs depends on the used components. Other studies show that certain programmers introduce more bugs than others.

As already stated, the industry standard to deal with bugs is to deploy software patches that remove bugs. But these patches are not sufficient, because first, not every bug is fixed and second, there is a window of exposure between detecting a bug and the application of a patch which removes this bug. The same economical reasons that lead to release a product with bugs encourage software makers to only fix most severe bugs, for example, a security vulnerability is more likely to be removed than a bug that can lead to a crash failure. Even if bugs are removed by patches from the software makers, there is a window of exposure. In 2008 the computer company Apple needed in average 9 days (worst-case 156 days) after the publication of a vulnerability for Apple’s web browser Safari until a patch was publicly available. On the other hand, users do not always apply patches instantaneously. For web browsers not more that 80% of the Firefox users and 46% of the Opera users had the most up-to-date version of their browser running on any day of 2007. A survey showed that 67.5% of Oracle database professionals do not install critical security patches. Reasons for not applying patches are that patches incur the risk of new bugs and that patching is sometimes uncomfortable for the software’s
user [34]. If a system is certified, then applying a patch can mean to lose the certification for this system.

**Automatic hardening** of software gives the user the ability to remove bugs and to tolerate failures independently of the software makers. Two general approaches of automatic hardening exist:

**Automatic Bug Removal** Automatic bug removal uses dynamic and/or static analysis to find and analyze bugs in a given software. The found bugs are removed by automatically deriving patches from the previous analysis. These patches do not only help the user, but can also serve as bug reports to the software makers [131]. We presented automatic bug removal in [117, 119].

**Error Tolerance** Automatic bug removal cannot remove all possible bugs. Hence, it is necessary to tolerate bugs at runtime. There is a lot of active research in that direction, e.g., [96, 94, 124, 84, 108, 50, 99]. The common goal is that the effects of bugs become not visible to the user. All of these approaches have in common that they rely on error detection with low performance overhead. Therefore, they apply light-weight, but incomplete error detection. We will show how to use speculation to implement heavy-weight more complete error detection with low performance overhead [42, 43, 121, 122, 120].

### 1.1 Terminology

This thesis uses the terminology definitions of [8]. Since this thesis focuses on software the following definitions are adapted to software. An **application** is a computer program that provides a service. It interacts with other applications, hardware and humans. The intended behavior of an application is defined by a **specification**.

Important for this thesis is the distinction between **failure**, **error** and **fault**:

**Definition 1.1.** A **failure** is a deviation from an application’s specification. Crash failures, for instance, are an unspecified termination. Another important failure is the successful abuse of an application by an attacker.

**Definition 1.2.** An **error** is the deviation of an application’s runtime state from the intended state. This erroneous state can lead to a failure. However, an erroneous application state does not necessarily imply that the application will fail at some point in time. This fact is exploited to tolerate errors by repairing the application state before the application fails.

**Definition 1.3.** A **fault** is a defect in an application that can lead to an error. Commonly, software faults are called bugs. A security vulnerability is a special kind of bug that can lead to a failure controlled by a malicious attacker. A fault in an application A can be the result of a propagation of a failure in application B into A.

An application interacts with humans. Automatically hardened applications are interesting for three different groups of people:
1.2 Automatic Hardening

**End-User/Administrator** Automatic hardening increases the dependability, availability, and security for the end-user of the application. An administrator who supports users of an application also counts as end-user when considering automatic hardening. We assume that both, end-user and administrator, have neither the skills nor the incentive to actively support the automatic hardening process.

**Developer as User** Applications also provide services to other applications. The developer of application $A$ using application $B$ risks a failure of $A$ even if $A$ is bug-free because $B$ can have bugs. These bugs can lead to failures of $B$ that might propagate to $A$ and, hence, might become failures of $A$. Even if the developer has the skill to review $B$ for bugs, the already introduced economic reasons will most likely prevent that the developer reviews $B$. To reduce the risk caused by using $B$ the developer can use automatic hardening. In general, we assume that the developer has not much more means than the previously introduced end-user and administrator to support the automatic hardening process.

**Application Developer** Automatic hardening can also be used as a debugging tool for the developer of the original application. A generated patch can serve as a template for the developer to remove a bug [131]. Some automatic debugging tools [123] use approaches similar to the ones used for error tolerance [94]. Instead of tolerating a detected failure the failure gets analyzed to pinpoint the underlying bug(s).

Additionally, an application developer can apply automatic hardening to her application to reduce the risk for the user. We assume, that in contrast to the end-user, administrator and developer as user the application developer can support automatic hardening. For instance, it might be acceptable for the application developer to do small code changes to enable error tolerance [42, 43, 121, 122, 120]. Nevertheless, the main focus of automatic hardening is the end-user side.

1.2 Automatic Hardening

Automatic hardening combines error detection with automatic bug removal and error tolerance. The main application area of automatic hardening is already deployed software. Therefore it is a preventive maintenance technique [8]. Since automatic hardening is used at the end-user site it has to be fully automated. End-user or administrator have typically neither the knowledge nor the resource to assist in the hardening process. Automatic hardening can be a service provided by the operation system: even if a user does not apply a patch the automatic hardening protects the user’s system.

Hardening is a best-effort process. That means that there might be always some bugs that cannot be fixed or found. Nevertheless, the goal is to tolerate or to remove most of the found bugs.

Automatic bug removal or tolerance is the primary goal. The following secondary goals are needed to make the approach applicable:

**Hardening should not decrease dependability, availability, and security.** We have already discussed that there is also a risk that a patch introduces new bugs. Of
course, we want that automatic hardening achieves “something good” even if it introduces new bugs. To rephrase this goal: Hardening should increase the dependability, availability, and security of the protected application.

Furthermore, sometimes a trade-off is possible. For example, for security it might be acceptable to trade availability for integrity. A detected malicious control flow manipulation could be mapped to a stop-failure by crashing the application. Thus, at least the injection and execution of the attacker’s code can be prevented.

**Low runtime overhead.** Hardening sometimes involves a substantial runtime overhead. A trade-off between runtime overhead, dependability and security failures exists. To be acceptable hardening should only lead to small performance overheads.

Especially error tolerance is prone to high performance overheads. Therefore, existing tolerance approaches [96, 94, 124, 84, 108, 50, 99] are evaluated with light-weight, but also incomplete error detection. We show in Chapters 2 and 3 how to implement more complete error detection with low performance overheads by using speculation.

**No source code required.** Most commercial software producers deliver their software as compiled binary. Therefore, users have only access to the binary. So, hardening has only access to the binary. Hardening cannot even expect debugging information in the binary.

For an application developer it might be acceptable to make changes to the source code to enable hardening or to increase the chances of success at the end-user site.

### 1.3 Contributions

Figure 1.1 associates the contributions of this thesis with the two aspects of automatic hardening: error toleration and automatic bug removal. The contributions to error tolerance are:

- a) preventing code injection by dynamically updated system call models
- b) parallelizing runtime checks

And for automatic bug removal:

- a) detecting and patching bad error handling
- b) removal of robustness and security bugs in COTS libraries
1.4 Theses

a) **Preventing code injection by enforcing dynamically updated personalized system call models.** As already introduced security vulnerabilities are a special kind of software bug that allow a malicious attacker to take control over a system. We show in Chapter 2 how to prevent a malicious attacker from code injections and control flow manipulations. The main approach is a novel combination of enforcing a fine-grained system call model and dynamic taint analysis. *SwitchBlade* is our implementation of this approach. *SwitchBlade* has a low performance overhead, a low chance of false alarms, and a high probability that an attack is detected.

b) **Parallelization of runtime checks.** Any error tolerance mechanism needs first to detect errors before they can be masked. At runtime, errors are detected by runtime checks. Some of these checks are very lightweight, for instance, a crash detector. Other checks can introduce a substantial performance overhead. We introduce in Chapter 3 the *ParExC* approach. *ParExC* distributes the perceived performance overhead of runtime checks over the cores of modern multi- and many-core CPUs. Therefore, *ParExC* parallelizes the runtime checks. *ParExC* makes several major contributions to the state-of-art of parallelizing runtime checks. Chapter 3 explains these contributions in detail.

Both contributions make use of *speculation* to reduce the perceived performance overhead of error detection.

This thesis makes two major contributions to automatic bug removal:

a) **Detecting and patching bad error handling.** Studies show that error handling code contains a disproportionate large amount of bugs [29, 126, 132]. In Chapter 4 we discuss how to find error handling bugs and how to generate patches that remove at-least some error handling bugs.

b) **Removal of robustness and security bugs in COTS libraries.** Most applications rely on commercial-of-the-shelf (COTS) code. COTS code comes often in the form of binary libraries. We demonstrate in Chapter 5 how to find robustness and security bugs in COTS libraries and how to generate patches that remove some of these bugs.

Both contributions are complementary. The first contribution deals with bugs in an application calling COTS libraries. Whereas, the second contribution deals with bugs in COTS libraries called by an application. The underlying approaches of both contributions are similar but not identical. Both approaches use fault injection to identify bug instances of known bug patterns. Furthermore, the patch generation is based on templates related to these bug patterns.

1.4 Theses

This section presents and discusses the theses derived from the introduced motivation for automatic hardening:
1 Introduction

**Thesis 1.** Speculation can be used to reduce the perceived performance overhead of error detection mechanisms.

Error detection is needed for tolerating errors at runtime. The perceived performance overhead of many error detection mechanisms make them unusable in practice. For example, detecting memory errors with a state of the art bounds checker can lead to a 12x performance overhead [98]. Other approaches to detect memory errors like data-flow integrity (DFI) [22] checking have similar performance overheads, e.g., 2.5x and higher. These performance overheads increase the latency of the checked application. Because errors are unlikely, speculating that no error occurs can be a valid strategy to reduce the perceived performance overhead and to decrease the latency of the checked application. In Chapters 2 and 3 we show two different approaches to speedup the error detection using speculation and therewith reducing the latency.

The SwitchBlade approach (published at EuroSys’08 [42] and ACM SIGOPS Oper. Syst. Rev. [43]) combines a low-overhead checker with possibly false alarms, with a high-overhead checker with low probability of false alarms. SwitchBlade speculates that the low-overhead checker has no false alarms by only running the low-overhead checker. Whenever an alarm is detected, the high-overhead checker evaluates a defined part of the last execution of the application by replaying this part to check if the alarm was a true or a false positive.

ParExC uses current multi-core CPUs to decrease checking overhead (published at (EC)$^2$’09 [21], at SSS’09 [18] and to appear at CGO’10 [20]). It speculatively executes an unchecked variant of the application as predictor. The predictor’s execution is partitioned into epochs. Each epoch is replayed with error checking. The checked replay of different epochs is done in parallel to each other and to the predictor. Thereby, we decrease the runtime of the checked application compared to an execution without ParExC.

**Thesis 2.** The system call model of an application depends on the environment.

System call interposition is a common approach to restrict the power of applications and to detect code injections. It enforces a model that describes what system calls and/or what sequences thereof are permitted. However, there exist various issues like concurrency vulnerabilities and incomplete models that restrict the power of system call interposition approaches. We present in Chapter 2 a new approach that uses randomized and personalized fine-grained system call models to increase the probability of detecting code injections. However, using such a fine-grained system call model, one cannot exclude the possibility that the model is violated during normal program executions. For instance, the system call model of the Apache web-server depends on the clients (web-browser) using it. The system call model of the UNIX tool grep depends on the likelihood of failures in the environment. We show with fault injection experiments that for grep the system call model changes with the number of injected faults.

**Thesis 3.** System call model enforcement combined with taint analysis can detect control flow manipulation with in average low performance overhead, low false positive, and low false negative rate.
To cope with false alarms of system call model enforcement, SwitchBlade uses on-demand taint analysis to update a system call model during runtime.

The approach has the low performance overhead and low false negative rate of system call model enforcement and the low false positive rate of taint analysis.

**Thesis 4.** *Parallelization of runtime checks can (partly) mitigate the performance costs of expensive runtime checks.*

In Chapter 3 we present and evaluate a novel framework, ParExC, to reduce the runtime penalties of compiler generated runtime checks. An obvious approach is to use idle cores of modern multi-core CPUs to parallelize the runtime checks. This could be accomplished by

(a) parallelizing the application and in this way, implicitly parallelizing the checks, or

(b) by parallelizing the checks only.

Parallelizing an application is rarely easy and frameworks that simplify the parallelization, e.g., software transactional memory (STM), can introduce considerable overhead. ParExC is based on alternative (b). ParExC has – in contrast to similar frameworks – two noteworthy features that permit a more efficient parallelization of checks: (1) speculative variables, and (2) the ability to add checks by static instrumentation.

**Thesis 5.** *Parallelizing runtime error checks themselves can scale better than parallelizing an application with runtime checks.*

Chapter 3 compares ParExC with an approach using a transactional memory-based alternative. Our experience is that ParExC is not only more efficient than the STM-based solution but the manual effort for an application developer to integrate ParExC is lower.

**Thesis 6.** *Automatic bug removal based on patch patterns can decrease the number of failures.*

A bug pattern is a template for a given kind of bug. We show how to use fault injection to detect bug instances of two common bug patterns:

1. Error handling bugs in Chapter 4 (published at EDCC’06 [117, 118] and at DSN’07 [119]) and

2. Input validation bugs in Chapter 5 (published at DSN’07 [119]).

Related to the bug patterns are patch patterns that are templates for removing the bug instances of the related bug patterns. We show two general patch patterns:

1. Error code mapping in Chapter 4 and

2. Input filtering in Chapter 5.
1 Introduction

Thesis 7. **Fault injection finds failures even in mature software.**

Bad error handling is the cause of many service outages. We will show in Chapter 4 ten well known GNU coreutils [3] that contain error handling bugs which lead to crashes. We have found these bugs with fault injection.

Commercial-of-the-shelf (COTS) components, like software libraries, can be used to reduce the development effort. Unfortunately, many COTS components have been developed without a focus on robustness and security. In Chapter 5 we evaluate the Apache Portable Runtime (APR) [2], a mature software library used on well-known and well-tested open source projects like the Apache web-server [1] and Subversion [5]. We found a failure for more than every second injection into APR.

Thesis 8. **Automatic error mapping can mask failures.**

Chapter 4 also presents one approach for automatic hardening applications against error handling bugs: *error mapping*. Error mapping maps an error from a site in the application known to have error handling bugs to a site that has no known error handling bugs. The underlying assumption of error mapping is: that the mapped error will be handled at the second site without resulting in a failure. Pre-allocation of memory is one example for error mapping. Error mapping can mask up to 84% of previously detected failures.

Thesis 9. **Automatic filtering can mask a high percentage of failures injected by bit-flip faults.**

In Chapter 5 we introduce a novel approach to harden software libraries to improve their robustness and security. The approach is automated, general and extensible and consists of the following stages. First, we use a static analysis to prepare and guide the following fault injection. Second, in the dynamic analysis stage, fault injection experiments execute the library functions with both usual and extreme input values.

We automatically harden the library by deriving and verifying one protection hypothesis per function (for instance, function foo fails if argument 1 is a NULL pointer). A protection wrapper is generated from these hypothesis to reject un-robust input values of library functions. We evaluate the approach by hardening the APR used by the Apache web-server. 56% of the bit-flips injected at the interfaces of the library could be prevented by the automatically generated protection wrapper.
2 Enforcing Dynamic Personalized System Call Models

Many desktop applications and almost all server applications maintain network connections. Network connections can make applications vulnerable to code injection attacks. For example, small coding mistakes can introduce buffer overflow vulnerabilities that might be exploitable by attackers. Users can harden their applications against code injection attacks by running the applications in a sandbox. The sandbox trades availability for security, i.e., whenever the sandbox detects a code injection, the application is aborted before the attack can do any harm. Dynamic taint analysis [115, 28, 81, 55, 95] is a well-known approach to implement such a sandbox. It detects code injection attacks by checking if the control flow has been altered maliciously, i.e., tainted, by network or other user input. Since all control flow instructions need to be checked and taint information has to be propagated via a dynamic data-flow analysis, taint checking introduces a non-negligible performance overhead. Newer works, e.g., [80], try to restrict this impact by tracking less data and by checking less control flow instructions. For example, by restricting the tracking to detect the exploitation of known vulnerabilities one can restrict the amount of tracking and instrumentation and, hence, the performance impact.

Restricting the taint analysis to known vulnerabilities improves the performance, but it decreases the protection from exploits using so far unknown code vulnerabilities. The underlying hypothesis of our approach is that injected code has some purpose, e.g., getting access to the system, and to achieve this purpose, the attacker has to execute system calls. Hence, one can detect or at least restrict the power of code injections by restricting the system calls that can be performed.

The system calls, an application is permitted to perform, can be restricted along three dimensions:

1. the set of system calls that an application is permitted to execute,
2. the set of permissible arguments to a system call, and
3. the temporal order of system calls.

Several systems [53, 90, 93] use a combination of (1) and (2). Anomaly-based systems [46, 56, 129, 109, 103, 64, 75] use (3) to detect intrusions based on statistics of the sequence of system calls: Given a window of the last $k$ system calls, the system decides if the sequence seems to be normal (no intrusion) or abnormal (intrusion).

Our approach is to enforce a fine-grained system call model that enables us to restrict system calls along all three dimensions: (1) set of system calls, (2) permissible arguments and (3) temporal order.
Both sand-boxing approaches dynamic taint analysis, and system call restrictions have different trade-offs when applied for automatic hardening.

- **Dynamic taint analysis** On one hand, dynamic taint analysis has a low chance of false negatives and false alarms. False negatives are not detected code injection attacks. Hence, dynamic taint analysis makes it difficult for an attacker to inject code and at the same time does not reduce the availability by false alarms. However, the price is a relative high performance overhead.

- **Enforcing a fine-grained system call model** On the other hand, fine-grained system call restriction can have a low performance overhead. But, as we will show, a fine-grained system call model bears the possibility of false alarms. False alarms are an implication from the observation that the system call model of an application depends on the environment (Thesis 2).

Furthermore, system call restriction is prone to mimicry attacks, i.e., attacks that imitate the exact system call behavior and cross system call data flow of the program. An attacker could slightly change the arguments to system calls such that the program behavior fits the needs of the attacker without violating the system call model.

In our approach, called SwitchBlade\(^1\), we make use of the observation of Thesis 2 to reduce the chance of false negatives through mimicry attacks. We can make a mimicry attack arbitrarily difficult by using **personalization** (see Section 2.3.1) and **randomization** (see Section 2.3.2) of the system call model.

SwitchBlade **personalizes** the system call model by learning a minimal system model that describes the execution of an application in a given environment. An attacker could learn the set of accepted system call sequences by analyzing the source or binary code of an application. We try to use the smallest model that describes the sequence of system calls of an application in a given environment. Thereby we want to prevent that an attacker is able to mimic program features that are disabled in a given environment. For example, an editor might be compiled without the feature to spawn shell scripts. Personalization will remove the system call subsequences associated with such features that are statically or dynamically (via a configuration file) disabled. We show in Section 2.4.1 that models depend on various factors such as the client applications accessing a protected server application and errors experienced in a given environment (Thesis 2). Thus, an attacker cannot be sure that all sequences permitted by the code are indeed permitted by a system call model for that specific system he is attacking.

SwitchBlade **randomizes** the model by inserting system calls in the application (and hence, in the model) that are not in the original program. We can increase the difficulty of mounting a successful attack by inserting dummy system calls using, for example, library wrappers or source code transformation tools. An attacker would need to be very careful to call (library) functions in the right sequence, with the right inter-system-call data-flow and with the correct backtrace. Moreover, an attacker needs to write the right return addresses in the backtrace, which are already randomized by virtual memory randomization feature of modern OSs.

\(^{1}\)This chapter is based partly on [42, 43].
Using personalized and randomized system call models means that one has to learn the model dynamically for each installation of an application. We address this problem by combining a taint analysis tool with a model learner. One learns some initial model using the learner and the taint analysis makes sure that one does not learn system calls from injected code during the training period. Our current implementation enforces the (initial) model speculatively by intercepting system calls with the help of a kernel module.

The initial system call model very likely is not complete and will therefore result in false alarms, i.e., we miss-speculate: some system calls performed by the application are not yet part of the model and will hence result in a model violation. Whenever this happens, we switch to a combined taint and learn mode. The last requests before the model violation are replayed in taint/learn mode. If the taint mode does not detect an intrusion, we update the system call model. Like other approaches that use a replay mechanism (e.g., the RX approach [94], Sweeper [124] or recovery-oriented computing [89]), we need an application-specific proxy to perform this replay.

While for stateless (server) applications a proxy is sufficient, for stateful applications we need periodic application-level check-points. Since we do not continuously track taint information in SwitchBlade, we cannot assume that a check-point is not already corrupted, i.e., there is the potential for false negatives – unless we mark the complete check-point as tainted. However, marking everything as tainted leads with a very high likelihood to a false alarm. Application-level check-pointing does not write the complete process state. Only the data that is essential for recovery is stored. Marking the complete application-level check-point as tainted avoids false negatives and in our experiments also false alarms. Note that many modern desktop and server applications support application-level checkpointing. For example, IMAP servers support a CHECK command to write the contents of mailboxes to disk. One can roll back to such an application check-point using a process in an uncorrupted initial state. This permits us to roll back to a point where the taint analysis was switched off and, thus, no taint information is available.

The SwitchBlade approach combines the enforcement of a fine-grained system call model with dynamic taint analysis using speculation. The system call model is enforced speculatively. Whenever an system call model violation is detected, SwitchBlade tries to confirm the speculative alarm by dynamic taint analysis. With SwitchBlade we demonstrate Thesis 1 (“Speculation can be used to reduce the perceived performance overhead of error detection mechanisms.”). One can use speculation to get:

1. the low chance of false negatives of a personalized and randomized fine-grained system call model,
2. the low chance of false alarms of dynamic taint analysis, and
3. the low performance overhead of system call interception.

SwitchBlade has several restrictions. First, it is currently only applicable to stateless applications and stateful applications for which an application-level check-pointing mechanism exists. Second, we require the support for replaying application-level requests with the help of a proxy. Replay mechanisms are technically difficult but can be reused to achieve other objectives like locating bugs [123], tolerating software bugs [89, 94, 124].
Enforcing Dynamic Personalized System Call Models

tolerating server crashes and facilitating load-balancing across machines. Third, our current implementation has several shortcomings, in particular, supporting multithreaded applications and kernel support for deterministic replay of system calls. Fourth, we can only detect code injections that modify the system call behavior. However, using randomization we can make it increasingly difficult for injected code not to modify the system call behavior.

In Section 2.1, we review the related work before we present the SwitchBlade architecture in Section 2.2. In Section 2.3, we introduce our system call model. We demonstrate the need for dynamic model updates and we describe our novel data-flow-based learner in Section 2.4. We describe the taint analysis in Section 2.5 and the model enforcement in Section 2.6. The performance and effectiveness of SwitchBlade is investigated in Section 2.7. Our evaluation shows that SwitchBlade can provide effective and efficient means for detecting and containing code injection attacks.

2.1 Related Work

There exist a variety of runtime tools that address certain classes of vulnerabilities, e.g., PointGuard [26], FormatGuard [25], StackGuard [24], Stack Shield, LibSafe [9], ProPolice [38], and LibSafePlus [7] to name a few. Some of these tools require the recompilation of the source code and might therefore not be applicable to protect third-party software. More generic tools exist that cover a larger class of vulnerabilities, e.g., program shepherding [61], XFI [37], and dynamic taint-based tools like the ones described in [28, 55, 81, 115]. For several runtime tools, researchers found ways to circumvent their detection mechanisms (e.g., [66, 106]).

There exists a significant body of related work in the domain of system call interception and intrusion detection. System call interception to confine the intrusion into applications has been used for many years [53, 90, 93]. Intercepting security related operations is even supported by default in the Linux kernel [135] and is used by security tools such as AppArmor [11, 27].

System call based intrusion detection systems can broadly be classified into misuse-based and anomaly-based systems. Misuse-based systems detect deviations from a usage model while anomaly-based ones detect statistical deviation from safe system call behaviors. There is a rich set of articles about anomaly-based intrusion detection, e.g., [16, 56, 129, 109, 103, 63, 75]. The basic underlying idea is to look at a window of system calls to detect deviations to known good windows of system calls. Newer approaches also use system call arguments [63] and the data-flow between the system calls [14] for the detection. System call based intrusion detection systems are susceptible to mimicry attacks that imitate the statistical system call behavior of the application [127].

Misuse-based detection schemes provide a set of rules that describe which system calls are permitted and which are not [53, 93]. SwitchBlade uses a misuse-based detection scheme and applies several mechanisms to prevent exploits from evading detection (see Section 2.3). Creating a good policy with a low false alarm and false negative rate is in general difficult. [93] points out the difficulties of generating a good policy for system
2.1 Related Work

call interception: (1) during the learning phase one needs to cover all possible code paths to reduce the number of false alarms at deploy time, and (2) one needs to make sure to avoid anomalies (like exploits) during the training phase to reduce the number of false negatives. We address these two main issues in SwitchBlade by combining a novel data-flow-oriented model learner and a dynamic taint analysis tool. The authors of [93] also point out that most of the policy violations they experienced were by a web server caused by the attempted execution of user-created CGI scripts. Therefore, we applied SwitchBlade to two web servers.

The policy used by SwitchBlade is quite different from those used by tools such as SysTrace [93] or AppArmor. SwitchBlade focuses on checking of the sequencing of system calls. We derived the system call model from the model carrying code approach [104] in which the model is used to confine mobile code. We added explicit garbage collection for model variables and we can tag arguments as being constant (see Section 2.3). Also, we learn the system call model via data flow analysis to ensure the accuracy of the data-flow constraints of the models.

Korset [13] also enforces a system call model. In contrast to SwitchBlade and [104] Korset extracts the system call model at compile time with static analysis. Thus, Korest does not support personalisation as every site enforces the same system call model extracted from the application’s source code. Furthermore, we believe that it is difficult to extract a system call model statically for applications that make heavy use of dynamically loaded plug-ins (such as Apache).

Checking the arguments of system calls can introduce races that might permit to circumvent this security mechanism. The Time Of Check To Time Of Use (TOCTTOU) problem is well studied in the literature [51]. Watson [130] shows that several system call interposition frameworks (Systrace [93], GWSTK [48], CerbNG [32]) are susceptible to races that are particularly easy to exploit on modern multicore systems. The main issue is that arguments that are passed to the kernel via pointers can be modified by a concurrent thread between the time of check and the time of use. Our approach to counter TOCTTOU attacks is to rely on information that is vulnerable to TOCTTOU attacks, such as the system call number and system call arguments (but no buffers in user-space). System call number and system call arguments are passed in registers into the kernel. Therefore, a concurrent thread has no access to the system call number and arguments. In this work we focus on the temporal order of system calls and some limited data-flow checking of the syscall arguments [14] and it could easily be extended to do more argument and data-flow checking.

Watson’s approach [130] is to set up arguments and spawn threads in such a way that page faults increase the chances that a race can be successfully exploited. Setting up these arguments and threads involves the execution of system calls. These system calls are not executed in the original program and, hence, most likely these system calls would be flagged as violating the normal sequence of system calls. Anomaly-based approaches use statistics on the sequencing of system calls to detect code injections. However, a smart attacker can execute additional system calls to make the sequence of executed system calls look normal [127]. In SwitchBlade, we do not use a statistical model but instead a model that describes for a given application the temporal order of system calls and the
data-flow between the system calls (see Sec. 2.3). Currently, we only check arguments passed in registers, i.e., we are not susceptible to TOCTTOU attacks.

Dynamic taint analysis [115, 28, 81, 55, 95] keeps track of untrusted input data, and detects attempts to misuse such tainted data, e.g., as a jump target. Several approaches to implement taint analysis exist based on emulation using emulators like QEMU [12], based on dynamic binary rewriting tools like Valgrind [78], or using hardware support. Using MMU support, [55] can dynamically switch between executing in QEMU while needing to track tainted data and executing natively when no tainted data is accessed. Recall that SwitchBlade only switches to taint mode after we detected a system call model violation. After a system call model became sufficiently complete, this should mainly happen because of the activation of an exploit. SwitchBlade provides in this case a way to reproduce the exploit and, hence, a way to help locating and fixing the vulnerability.

One possible application domain of SwitchBlade is to protect applications from worms. A reactive worm defense [23, 19, 92, 124, 18] consists of a set of monitoring sites that detect exploits and generated exploit-specific signatures. Signatures are used to filter out worm code from network traffic. These signatures might however not protect against polymorphic worms. One could address polymorphic worms by shipping vulnerability-specific execution filters like [80, 124]: the taint analysis is restricted to known exploits and in this way the overhead of the taint analysis can be reduced. However, all reactive worm defense systems have a vulnerability window that stretches from the time the worm starts spreading until the time a signature or execution filter is received and installed by a node. Sweeper [124] provides additionally a repair mechanism for detected infections by rolling back to an uninfected check-point.

SwitchBlade instead permits preventive worm defense. Worms need to communicate with the external world and, hence, are forced to perform system calls. This makes them detectable by system call interception and in this way they can be prevented from spreading.

2.2 SwitchBlade Architecture

The objective of SwitchBlade is to detect exploits with a very high likelihood while ensuring a low false alarm rate and a low performance overhead. SwitchBlade’s general architecture is depicted in Figure 2.1. SwitchBlade’s primary method for detecting exploits is a process-specific, fine-granular system call model (see Section 2.3). The system call model describes the set of permissible sequences of system calls, provides constraints on the data-flow of arguments of the system calls and the locations they can be called from. In our current implementation the system call model is speculatively enforced with the help of a Linux kernel module.

**Assumption 2.1.** An attacker has to execute system calls in the context of the attacked application achieve her goal.

As already outlined, our approach is based on the assumption that an attacker will make the application perform system calls, for instance, to start a shell or to open a network
2.2 SwitchBlade Architecture

Figure 2.1: SwitchBlade architecture: in normal mode, all system calls are checked against a system call model. Violations result in a switch to taint mode, in which the last requests are replayed using a fine-granular data flow and control flow analysis. To facilitate replays, all network traffic is routed through a proxy.

connection to another server. These system calls are very likely not part of our process-specific, fine-granular system call model. Hence, we need to detect when an application deviates from its system call model to detect an attack.

SwitchBlade requires that we can roll back an application to a previous state in case of a violation of the system call model. We then replay the outstanding requests in taint mode to decide if the model violation was caused by mis-speculation, i.e., an incomplete model or by an intrusion. In a stateless server, the outstanding requests are all requests that have not yet been replied to. In a stateful server these are all requests that need to be processed after the most recent check-point was taken.

Because of non-determinism in the execution, a model violation in normal mode might not reoccur during replay. This could result in a performance penalty because we might have to switch multiple times to taint mode and then switch back to normal before, eventually, we might succeed to extend the system call model. In our experience this is not an issue and models grow nicely with the number of model violations (see also Section 2.7).

Roll back of speculative processing (in enforcement mode) can in general be supported via check-pointing of application state and logging of input and output. For logging, we use the standard approach of sending all client interactions through a proxy. The proxy is application-specific and maintains the set of requests since the last application check-point. Checkpoints contain sufficient state to permit an application to roll back and then
replay/continue the execution. In SwitchBlade, we cannot use check-points that simply save the content of the entire address space. When restarting the application in taint mode, we would not know which parts of the address space are tainted and which are not. Tainting the entire address space would result in a high false alarm rate. Our approach is to use an application-level check-pointing mechanism. Most applications can save the state that is essential for them to recover from a crash and taint-based approaches can deal with such application-level check-points. For example, text editors (such as Vim) save the state of the currently edited buffer periodically to disk. For stateless servers we do not need an application-level check-pointing mechanism. For switching to taint mode the server is always restarted from its initial state. The proxy then resends all outstanding requests that have not been replied before the model violation.

A more detailed view of SwitchBlade is given in Figure 2.2. A server process initially starts in taint mode, forks a child process that switches on system call enforcement for itself before it goes to normal mode to process the first request. Starting a server in taint mode ensures confidentially in its initial state. Requests are processed until a violation is detected. The parent is notified via a signal of the violation and spawns a new child process that processes the outstanding requests in taint mode before switching back to normal mode. If during taint mode a security violation is detected (e.g., using a tainted address as a jump target), all outstanding requests are dropped and the child terminates. The parent can continue by spawning a new child that processes the next outstanding

Figure 2.2: Execution of an application under SwitchBlade. Violations of the model result in the re-execution of the affected request in taint mode. If the violation was caused by a false alarm, the model is updated appropriately.
request. Thus, the attack is detected and tolerated (by dropping the attacking request). If no vulnerability is found in taint mode, but a deviation from the current system call model is detected, the system call model is extended to cover the replayed execution. The proxy ensures that the replay is not visible to the client (except for some slight delay). The model is updated in the kernel and system call model enforcement is switched on before the process leaves taint mode.

### 2.3 System Call Model

To give an intuition for how a system call model looks like Figure 2.3 shows the system call model of the *Arithmetic Unixbench* benchmark. State $l_0$ is the backtrace of the first `getrusage` system call. The backtrace of a system call is the sequence of all return addresses on the stack at the time when the system call is issued. The next allowed system call is the second `getrusage` at node $l_1$ and so on. State $l_6$ is the end state and is not associated with a backtrace. When reaching the end state no system calls must be issued anymore. The data-flow between the system calls is constrained: the first argument of the `write` and the `close` call have to be the return value of the `open` call. The variable $v_1$ denotes the data-flow from `open`, via `write` to `close`. The digit 4 identifies the register that carries the first system call argument. Thus, $4 = v_1$ means that the first argument must have the same value that `open` returned at $l_4$. The suffix `:free` denotes that $v_1$ can be garbage collected at $l_3$.

**Definition 2.1.** A **stack backtrace** is the sequence of return addresses on the stack.

**Definition 2.2.** A **system call model** is a graph in which each node represents a unique stack backtrace and edges represent system calls with optional argument constraints.

One can compute the backtrace at runtime (with techniques also used by debuggers) for instance with the help of frame pointers. No source code is needed for computing the backtrace. An edge from a node $n_{src}$ to node $n_{dest}$ that is labeled $\text{sys}_N$ means that the system call with system call number $N$ is called with the backtrace that is represented by $n_s$. Because an edge defines the transition from one node to its successor, we use the terms edge and transition synonymously. For any system call number $N$ and any two nodes, there exists at most one edge that is labeled with $\text{sys}_N$. For better readability we often replace $\text{sys}_N$ by the name of the system call, e.g., `open` or `write`. Typically, all edges starting from a node refer to the same system call number. However, some library
functions do not properly update the frame pointer and, thus, some return addresses on
the stack are not part of the computed backtrace. This can result in (rare) cases in models
in which different system calls are issued with the same incomplete backtrace. This also
implies that there can be multiple edges between two nodes.

The outgoing edges of a node $n_{src}$ determine the set of system calls that are permitted
to be executed with the backtrace associated with $n_{src}$. If there are $k$ outgoing edges from
a node $n_{src}$, then a system call executed with the backtrace represented by $n_{src}$ has to
match at least one of the $k$ edges. Matching means that the system call number $N$ of the
edge and the number of the executed system call are identical and all argument constraints
are satisfied (see below). The edges that match determine the set of backtraces of the
next system call. As soon as the backtrace of the next system call is known, only one
edge is permitted to match. Say that the backtrace of the next system call is represented
by node $n_{dest}$, then there must be exactly one edge between $n_{src}$ and $n_{dest}$ that matches
the system call with backtrace $n_{src}$.

The system call model not only determines the permissible sequence of system calls,
but it also restricts the data-flow between system calls. Edges may therefore constrain
the arguments of the system calls. We achieve this by introducing variables and constants
in the system call model. Each model variable represents a set of values. A model can
contain argument constraints of the form $a = v$ on edges. This means that the value of
argument $a$ must be in set $v$. Most of these arguments contain handles like file descriptors
or sockets. Currently, we only support arguments that are passed in registers. Note, in
the automatically generated model graphs (e.g., see Figure 2.3) arguments are described
by their offset in the architectural depending CPU-state. For instance, 4 is the offset
of the register ebx on x86. Register ebx carries the first argument of a system call on
x86-Linux. Thus, $4 = v$ means that the value of the first system call argument has to be
in the set of variable $v$. 

Figure 2.4: Programs might not always close handles. The system call model can some-
times remove values from variable sets even if they are not freed by the pro-
gram. In this case, the file descriptor is removed from set $v_{20}$ on the last write
call.
Some arguments are always constant, e.g., one might pass always the same file name that is stored at a fix address and with a constant set of permission flags. We support constraints that say that an arguments is constant. These are of the form \( a = c \) where \( c \) is a constant value, e.g., \( a = 0x3 \). One could support a set of constants (as we support variables that contain a set of values; see below). However, for efficiency, our model and implementation is currently restricted to one value. Figure 2.4 depicts a model that contains several constant arguments. Note that, if a constant pointer points to a code page or a read-only page, an attacker will not be able to control this argument because this would require to make the page writeable. This would in turn require a system call and this system call would with a high likelihood not be covered by the system call model. We omitted the constant arguments in all following system call models for readability.

During runtime one can add and remove values from the model variables. The model permits to define one model variable per edge. The return value of the system call executed at this edge is stored in the set represented by this variable. In the graph we denote this by edge labels of the form \( v_i = \text{sys}\_N \). For example, an open call returns a file descriptor, which might be added to a model variable \( v \) and a later write call might contain an argument constraint that says the first argument has to be in \( v \). If a variable is never used as an input argument, we drop it from the model. This means that we do not need to maintain a model variable for each system call during runtime but only for those system calls whose return value may be used as an argument in some later system call.

We have to be able to remove values from the variable sets at runtime. To do so, we can add a free attribute to the edge at which a variable value is used the last time. For example, in Figure 2.3 we remove the value from variable \( v \) that is passed as argument to close. We represent the removal with the suffix :free. However, handles might not always be closed as depicted in Figure 2.4, where, no call to close is issued for \( v20 \). We can nevertheless remove the values from the variable set because from a model perspective we are only interested in keeping values that might still be used in an argument constraint. For example, see the edge from node \( l21 \) to node \( l23 \) in Figure 2.4. There we remove the value of the file descriptor argument passed to write from the variable \( v20 \).

Our system call model is similar to the one used in [104] to predict what system calls an application will perform. The main differences are with respect to the constraints of arguments in which we restrict ourselves to simple checks on system call arguments passed in registers. Moreover, our model explicitly states when certain model variables can be garbage collected and which arguments are constant. More importantly, we personalize and randomize the model since our use of the model is different from that of [104].

### 2.3.1 Personalization

Most injected code will have to perform system calls to achieve its purpose. For example, a worm will try to connect to other hosts, or an injected keyboard logger would need to write the keystrokes somewhere. To stay undetected by SwitchBlade, an exploit would need to:

1. perform system calls in an order that is consistent with the system call model,
2 Enforcing Dynamic Personalized System Call Models

2. for each system call, the backtrace needs to match the sequence in the model, and
3. the arguments of a system call need to satisfy all argument constraints active for the call.

To maximize the chances that an exploit violates the system call model – and unlike other system call interception approaches – we want to personalize the system call model. Our goal is to use a minimal system call model that describes the permissible sequences of system calls of an application within a given environment. We show in Section 2.4 that system call models are dependent on many factors like the clients of a protected server application and the error rate of the underlying system. Hence, by only permitting the system call sequences SwitchBlade observed in a given environment, we restrict the possibilities of an attacker.

An attacker cannot just study the source code or program traces of another installation to know for sure which sequences are permitted and which are not. For example, in a given environment certain program features might be disabled by a configuration file. Our system call model should therefore prohibit system call behaviors specific to the disabled features. Of course, an attacker could always try to restrict herself to a minimal system call model. We show in the next section how to minimize the chance of a successful attack by using randomization.

The potential disadvantages of personalizing the system call model are:

- one needs to learn the system call model in each environment, and
- the probability of false alarms will most likely increase because one cannot invest as much time in training in each environment as if all training would happen once centrally for all environments.

We address these two issues by automatic learning and updating the system call model at runtime (see Section 2.4). This facilitates the use of personalized system call models without increasing the risk of false alarms and with a low training overhead.

2.3.2 Randomization

Our goal is to detect code injections with a high likelihood even if these do not change the original system call behavior. Personalization makes it already more difficult to find permissible system call sequences. However, an attacker might still find system call sequences that are permitted in most environments. To minimize the chance that an exploit succeeds in imitating the system call behavior even further, we randomize the model in two ways. By default, Linux randomizes the location at which dynamic link libraries are loaded. This automatically randomizes the return addresses in backtraces. Our approach supports this randomization by updating the expected addresses in the system call model accordingly. An attacker would have to figure out the expected return addresses on the stack for each system call that she wants to perform (without performing any other system calls).
2.3 System Call Model

We do not solely rely on address space randomization. We also randomize the system call model by injecting random system calls. We implemented a wrapper generator that can wrap library functions used by an application and/or functions within an application. These function wrappers call system calls with a random and invalid system call number (32bit size) before and/or after calling the original function. Figure 2.5b shows a randomized version of the partial Apache model depicted in Figure 2.5a. The generated random wrapper inserts a new system call sys_2533835476 between open and socketcall. The execution of an invalid system call is particularly fast because the kernel performs a simple check if the system call number is valid. If it is not, the call just returns with an error code that is ignored by the wrapper.

Our wrapper generator uses randomization to generate new wrappers. Different installations of an application can have different random wrappers and, hence, have different system call models. Because of our dynamic model learner it is possible to have different wrappers for different executions of the same installation of an application. Each time the application is started a new random wrapper is generated. Because of the randomization the attacker cannot reason about a specific wrapper, even if she knows the wrapper generator.

An attacker would have to inject code that can guess or somehow figures out at runtime the permitted sequence of system calls together with the correct backtrace for each system call. One possible way to achieve this would be to emulate the original code using an emulation framework. However, the emulation framework would have to stay undetected. Therefore, it would have to work without library/application functions and system calls. The original code would be used as an oracle for the right calling sequence of system calls for the attacking code. To prevent this attack, we need to hide the wrapper code from

Figure 2.5: Extract from system call model for Apache worker processes.
2 Enforcing Dynamic Personalized System Call Models

the attacker. We want to set all pages containing wrapper code as execution only, i.e., we would prohibit any read accesses of the wrapper code. This would prevent the emulation of the wrappers. Changing the protection of wrapper pages would require a system call that would, by definition, violate the system call model. Supporting execute only pages on IA32 is possible but non-trivial and would require changes to the Linux kernel. So far, we have however not implemented these changes.

Preventing read access to the wrapper code, will prevent the emulation of code that might contain system calls. Since an attacker would not know which functions contain system calls and which not, injected code would have to call all functions in the original order to stay undetected. Moreover, changing the arguments to these functions might result in:

- a sequence of system calls that was not experienced during learning or
- a data-flow between system calls that might not be covered by the model.

In other words, by inserting more random invalid system calls in the program code, we can make it more and more difficult for an attacker to come up with a permissible sequence of system calls.

Our wrapper generator combines personalization with randomization, by generating one random wrapper per application installation. Furthermore, it is possible to start the same application with different random wrappers. We can increase the randomization even further by adding random data-flow and random but constant arguments.

2.4 Model Learner

SwitchBlade can learn and update the system call model at runtime. The need for dynamic model updates not only stems from our approach to use personalized and randomized system call models but is a general problem of system call interposition frameworks. To motivate this, we first show several measurements indicating the problem of coming up with a good system call model, i.e., one that has a low false alarm rate.

2.4.1 Problem: False Positives

As stated in Thesis [2] the system call model of an application depends on the application’s environment. This dependency can lead to an unacceptably high false alarm rate for model-based system call monitoring. The model is typically learned by the use of traces (e.g., see [104]). One can easily see that tracing can lead to incomplete models in which nodes (i.e., backtraces) and transitions (system calls with argument constraints) are missing. For example, it is unlikely that all error handling code is executed during tracing. Typically, error handlers will issue additional system calls, e.g., for logging error messages. Figure 2.6 shows the growth of the system call model for grep when injecting errors into malloc: in each run another malloc call returned NULL. During high system load one needs to expect a higher likelihood of executing error handling code because of resource depletion problems. An incomplete system call model, i.e., that does not cover
all error handling, might result in false alarms. In other words, without dynamic updates the model enforcer will abort the application because of model violations at times when the system is used (needed) most.

As already mentioned before, changes in a program’s configuration files might also result in a different system call behavior, e.g., by switching from a centralized logging demon to local logging.

Different client programs can also trigger different code sequences in a server program. Since the impact of different client applications might be less clear, we measured its impact for Apache. Figure 2.7 shows the size of models generated for Apache depending on the client program used. The unified model was generated using five different client programs: we used four different browser versions and wget to retrieve the same sequence of web pages from an Apache web server. Figure 2.7 already shows that it is not enough focus on one client in the learning phase. Especially, using the easily scriptable wget only leads to an incomplete system call model.

Figure 2.8 shows the overlap of the different Apache models. While the overlap in nodes (i.e., backtraces) is relatively high, the overlap in edges is much less pronounced. To give an impression of how the models differ from each other, we depict the three nodes and some of the edges that are missing from the wget-generated Apache model in comparison.
to the unified model (Figure 2.9). Three of the reasons for divergent models are:

- Slightly different requests from the clients, e.g., User-Agent field.
- Web-browsers in contrast to `wget` automatically try to fetch a file containing a web-site icon (`favicon.ico`).
- Connections can be used to transmit more than one request/reply pair. How many request/reply pairs are transferred per connection and who closes the connection depends on both the server and the client application.

A major problem of the enforcement of an incomplete system call model is that it might be used for denial of service attacks. For Apache it might be sufficient for an attacker to browse with an uncommon browser to cause the abort of an Apache worker process. While Apache will spawn a new worker process, this might nevertheless lead to a major reduction in throughput for all clients when a new browser version becomes available. For example, even minor changes from one browser version to another (like from Firefox 1.5 to Firefox 1.5.10; see Figure 2.7) can modify the system call model. This can lead to model violations which in turn can result in an increased service unavailability.

One might guess that the issue of incomplete models is mainly caused by the kind of model we are using. For example, some system calls associated with the missing transition of the `wget`-generated Apache model depicted in Figure 2.9 are actually already known from other transitions. More specifically, the open going to $l_{19}$ is the same as going to $l_{20}$. The same is true for the `sendfile64`, `socketcall` between $l_{28}$, $l_{30}$, and $l_{27}$ as well as, between $l_{28}$, $l_{29}$, and $l_{16}$, respectively. Hence, omitting the backtrace of a system call would merge $l_{19}$ and $l_{20}$ (and so on) and, thereby, minimize false alarms. But it might potentially increase false negatives which we want to avoid. To investigate
2.4 Model Learner

Figure 2.8: Overlap of the Apache system call models generated with 5 different client programs: left graph shows overlap in nodes and the right the overlap in edges.

Figure 2.9: Part of the system call model for Apache. Transitions and edges missing in the Apache model generated with `wget` (dotted lines) in comparison to the unified Apache model.
the issue of incomplete models, we also looked at alternative security mechanisms such as AppArmor. In our experiments we used an early (but at that time the only available) AppArmor profile for Firefox and displayed more than 10,000 pages. This resulted in a variety of policy violations. We adapted the profile to get Firefox 1.5 to run with the set of pages we experimented with. To do so, we needed to add 124 lines, remove 5 lines and modify 22 lines to the profile. Of course, we were not always sure if there was a policy violation or a real exploit of Firefox because some of the 10,000 pages contained exploits that worked for earlier versions of Firefox. Hence, using taint analysis during learning of the model is an important feature of SwitchBlade to make sure that we do not learn model transitions caused by malicious code.

2.4.2 Data-flow-Based Learner

Sekar et al. [104] describe a trace-based learner that tries to match values returned by some system call with arguments passed into subsequent system calls. For example, if there are two calls to open and the first returns value 1 and the second value 2 and a later call to write uses value 2 for the file descriptor, then the system learns that the 2nd open call and the write call are linked. The trace-based learner needs a specification of potential data-flow. The specification relates return values and arguments of different system calls (e.g., the return value of open can be used as argument for read, write and close). Without the specification a learner would interpret values that are accidentally equal as data-flow, e.g., open returns the same file descriptor value that is used as flag argument prot for a call to mmap. Learning in this way can result in wrong connections between system calls because values can be identical for other reasons. Also, some system calls $S_1, \ldots, S_k$ in a trace might return the same value $x$ and $x$ is later passed as an argument to another system call $S_s$ and it is not always clear which (if any) system call $S_p$ ($p \in \{1, \ldots, k\}$) did indeed produce value $x$. In other words, it is not always certain which system calls should be connected via argument constraints in the system call model. To distinguish between the different options, one would have to try to produce additional traces in which the system calls return different values that permit the learner to decide which system call produced a certain argument.

SwitchBlade tracks the data-flow of an application to learn the system call model of an application. In the above example, the data-flow analysis permits us to determine exactly which (if any) of the system calls produced the value $x$. Also, the data-flow analysis permits us to determine if some of the arguments might be constants.

To track the data-flow between system calls, SwitchBlade dynamically instruments the binary code using Valgrind [78], a binary instrumentation and analysis framework. The learning of the model is always performed in taint mode, i.e., in a mode in which we anyhow track the data-flow of an application. Hence, we use a combined Valgrind tool for taint analysis (see Section 2.5) and learning.

Our instrumentation assigns to each executed system call a unique ID (SCID) and records the backtrace of each call. Each return value of a system call is tagged with the SCID. Whenever this value is copied to another memory location or register, the new location/register is also tagged with the SCID. If a location is cleared or modified in some
other way, we remove the SCID from this location. We use the standard approach of keeping a shadow memory as described in [77] to track the SCID: we maintain 32-bit of shadow information for each register and potentially each word in memory. The shadow information does not only contain the SCID but also whether the content of the location contains a constant and whether it is tainted, i.e., untrusted data (see Section 2.5).

To track if a word contains a constant, we initially mark all code segments as containing constants. Whenever a word is read from a code page, the destination is also marked as constant. However, even if two constants are used as an input of an operation, we do not mark the result as a constant. Note that even if a memory location holding a program variable $v$ is marked as constant, this does not necessarily mean that $v$ indeed is a constant: the value of $v$ could of course depend on the control flow of the executed program. We use this marking to determine if an argument of a system call could be marked as constant in the system call model. This reduces the overhead of the learner because we only attempt to learn if an argument is a constant if it is marked as such by the data-flow analysis.

The learner records for each system call $S_i$ the system call number $SN_i$, the backtrace $b$, the SCIDs of the arguments (if any), and if arguments are marked as being constant. If the backtrace $b$ is not yet part of the current model, we add a new node $N_b$ that denotes the new backtrace. When executing the next system call $S_{i+1}$ with a backtrace $b'$, we check if there is already an edge between $N_b$ and $N_{b'}$ that is marked with $SN_i$. If this edge does not yet exist, we add it and mark it with $SN_i$ and the argument constraints. For each argument $a$ that contains a SCID, we add an argument constraint $a = v$, where, $v$ is the model variable associated with the SCID. If such a model variable does not yet exist, we add it to the model. For example, if the SCID refers to a system call $sys_n$ with a backtrace $x$, we add to all outgoing edges from node $N_x$ the prefix “$v_x = sys_n$”.

If an argument $a$ is marked as a constant, we add a constraint $a = c$, where, $c$ is the value that was passed as argument.

If the edge already exists, we check if it is consistent with our current observation. It is of course possible that the argument constraints of an edge might change between different calls. For example, the first time the edge is executed, argument $a$ may contain a value of some variable $v$ while in the second call $a$ may contain the value of a different model variable $v'$. To address this issue, we could permit disjunctions in constraints, i.e., $a = v \lor v'$. For now, we decided to keep the model simple and instead remove a constraint if we observe conflicting constraints.

Our learner does not only learn data flow that can be derived from the system call specification (e.g.: the file descriptor argument of $read$ was returned by some $open$). Other data flow like using the returned memory address from a $mmap$ call as buffer for a $recv$ is also learned.

Sekar’s et al. [104] trace based learner uses interface specifications for adding data-flow to its models. These interface specifications explicitly name constructors and destructors for every potential data-flow. For instance $open$ and $close$ are the constructor and destructor for the file descriptor interface, respectively. For our data-flow-based learner any system call is a potential constructor. Destructors are more difficult, but nevertheless, important to extract. At runtime, destructors garbage-collect no longer used model variables. Without the garbage-collection the set of variables might grow infinitely at
Enforcing Dynamic Personalized System Call Models

<table>
<thead>
<tr>
<th>Memory to check</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>tainted return address</td>
<td></td>
</tr>
<tr>
<td>tainted jump addresses calculated at runtime</td>
<td></td>
</tr>
<tr>
<td>tainted format strings</td>
<td></td>
</tr>
<tr>
<td>tainted code at jump targets</td>
<td>x</td>
</tr>
<tr>
<td>tainted heap meta data before free</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 2.1: Checks done by TaintCheck to detect and block exploits.

To detect and block exploits, our model learner marks any last usage of a variable with a :free. At runtime, the variable referred an argument marked with :free is garbage-collected. For example, in Figure 2.3 (page 17) the file descriptor value passed to close is removed from model variable v1. The last usages of a variable v are all edges e where v is a system call argument and where from e no other edge e’ is reachable where v is a system call argument.

2.5 Taint Analysis

Taint analysis is a well-known approach to detect and block code exploits. It can be implemented using a dynamic binary instrumentation framework like Valgrind [78]. One of the main advantages of this approach is that it works for arbitrary binaries without the need for recompilation. We have reimplemented the taint analysis tool TaintCheck [81] using our own data-flow engine. We added the support to generate traces to learn system call models (see Section 2.4). We also modified Valgrind to be able to switch to a native execution. This allows us to remove the overhead of Valgrind while running with system call enforcement switched on.

2.5.1 TaintCheck

Valgrind runs an application on a simulated CPU under the control of our TaintCheck tool. In this way, TaintCheck marks data from suspicious sources as tainted, traces tainted data flow, and blocks the usage of tainted data at vulnerable points. TaintCheck has a very low false alarm and false negative rate. One reason why tools like the original TaintCheck are not more widely used to protect vulnerable applications is Valgrind’s enormous slowdown (see Section 2.7).

TaintCheck maintains for each byte of memory and all CPU registers a shadow bit. This shadow bit is set to 1 if the corresponding byte of memory or register is tainted. Initially, all shadow bits are set to 0. In our current configuration all data that is read from the network is marked as tainted. Optionally, data from the file system can be marked as tainted too. Additionally, the software itself can mark data as tainted. Whenever memory words or registers are copied, their corresponding shadow bits are copied too. In that way we trace the propagation of tainted data throughout the address space of the application.
2.5 Taint Analysis

Table 2.1 lists all checks executed by our extended TaintCheck to detect and block exploits. All data is checked before it is used as a jump target, as format strings or to concatenate heap blocks after freeing them. Whenever a taint check evaluates to true, the corresponding operation is not executed. Instead, the current process is aborted with a detailed error message. The last two checks in Table 2.1 are not part of the original TaintCheck. We added them for completeness and better debugging. The first check is necessary if an attacker is able to overwrite code without changing the control flow. The second check is redundant because it should not detect anything that would not be detected by the other checks. However, it directly points to the vulnerable heap block and, thus, eases debugging.

2.5.2 Escaping Valgrind

Initially, we start the application under the control of TaintCheck. For stateless server applications, we take one process-level check-point (which is currently implemented by a simple fork) to be able to restart quickly in taint mode in case the model enforcer detects a model violation. After taking the check-point, we switch on the model enforcement and then switch from the simulated CPU under the control of Valgrind and TaintCheck to the real CPU. Therefore, we implemented an Escape feature for Valgrind. When an application wants to escape Valgrind, the state of the virtual CPU is copied to the real one and the application’s execution then proceeds on the real CPU. This is possible because Valgrind does not modify the memory layout of the application running on the simulated CPU.

TaintCheck stops after the escape. Thereafter, no data is marked tainted and no data flow is traced. While this might permit an attacker to change the control flow without being detected and to contaminate the application’s state, the model enforcer confines the effects of such an attack and will detect it as soon as the system call behavior deviates from the model. Our current implementation does not permit to switch back to Valgrind after an escape. This would be difficult to do correctly: If a vulnerability is detected by the model enforcer, the attacker might already have contaminated the application’s state and the control flow. Even if one could roll back to, say, some earlier process-level check-point, we would neither have taint information (unless we roll back to a point before we escaped from Valgrind) nor would we know if the state might be corrupted already. We address this issue by using application-level check-points to which we roll back for replay.

2.5.3 Replay of Requests

After the model enforcer detects an attack, it stops the application. To test if the detected attack is a false alarm because of an incomplete model, the outstanding requests are re-executed (see Figure 2.2 on page 16). Because the current state of the application might be contaminated by an attacker, we roll back to a known clean state that was still under the control of TaintCheck.

If the application is stateful, we load the most recent application-level check-point from disk. As this check-point could be already contaminated by an attacker the complete
check-point is marked as tainted, i.e., we do not trust the information that is in the check-point. From here on we replay all outstanding requests by asking the proxy to transmit them again.

During replay the execution does not escape from TaintCheck. Thus, TaintCheck verifies during the re-execution of the outstanding requests that no attack is taking place. The model learner checks if the model needs to be updated. In taint mode the enforcement of the system call model is switched off. If no exploit is detected during the re-execution, the learner will update the model.

The proxy ensures that a client does not see that its requests are replayed, i.e., duplicate replies are filtered by the proxy. SwitchBlade is transparent for client applications as long as they do not try to inject code into the server application.

For simple stateless server applications we created a generic wrapper library that provides transparent support for running stateless applications under SwitchBlade. In particular, we do not require any changes of the source code for stateless server applications that serve a single connection per execution (e.g., servers spawned via `inetd`). The generic wrapper library takes the initial process-level check-point (e.g., using `fork`) and switches from TaintCheck mode to model enforcing. We have used this approach for the micro benchmarks in Section 2.7.4. Servers serving multiple connections need some modifications of the application because some code for the initial check-pointing needs to be added to the application.

Although, it is in principle possible to substitute the proxy with system call recording and replay as used in [54, 14] and Chapter 3, it is technically difficult. The reason is that Valgrind changes the application’s behavior on system call level. For performance reasons, the enforcement mode runs without Valgrind. In the learning mode we need to enable Valgrind for the taint analysis and the data flow learner. Valgrind adds new system calls, e.g., to manage the memory for its own book-keeping. Some system calls (like `sigaction`) are completely implemented in Valgrind, they are not passed through to the kernel. Therefore, the sequence of system calls recorded in enforcement mode would not match the sequence of system calls issued in learner mode. Hence, it is not enough to record the system calls in enforcement mode and play them back in learner mode. With the proxy, we abstract from the system call level. The proxy replays network connections and indirectly the required system calls. But other system calls, for instance for memory management, are newly executed and not replayed.

## 2.6 Model Enforcement

The model enforcement ensures that an application escaped from the control of TaintCheck follows the application’s system call model. Therefore, model enforcement must intercept all system calls the application performs, compute the backtraces of all seen system call and compare these backtraces to the possible next states of the system call

---

2Valgrind registers its own signal handlers. When the protected application calls `sigaction` the pointer to the application’s signal handler is just stored in an internal variable. When signaled, Valgrind’s signal handler calls the application’s signal handler referenced by that variable.
2.6 Model Enforcement

We implemented our model enforcement tool as kernel module for Linux 2.6.20. Using the Linux Security Module (LSM), which is an interface to intercept security-related operations \[135\], would have been one option. Mandatory Access Control (MAC) systems like AppArmor and SELinux \[71\] are implemented on top of LSM. But our system call models are more fine-grained than the security policies of AppArmor and SELinux. We believe that SwitchBlade might therefore detect attacks earlier because an attacker might have to perform some system calls to setup the system call that is detected by the MAC system. We did not use LSM to intercept the systems calls because LSM does not intercept all system calls. Instead we use the utrace framework \[72\]. The utrace framework allows us to intercept and analyse all system calls an application does (similar to ptrace but in kernel space).

In this section we use the term enforced application and enforced process to denote an application and process running under the control of the model enforcer. The model enforcer checks that the application/process follows its given system call model.

2.6.1 Loading the System Call Model

When a process switches from TaintCheck to model enforcement, TaintCheck informs the model enforcer that the system call model for the current process needs to be enforced. Directly after this, the control is switched from the simulated CPU to the real CPU using TaintCheck’s escaping feature (see Section 2.5.2). This switch does not need a system call and, therefore, it does not appear in the system call model. The next system call performed by the application has to be from one of the starting states of the system call model.

When a process wants the system call model to be enforced, the model enforcer allocates space for the model and the model variables and sets the starting state of the enforced model. The model enforcer intercepts all system calls from all processes. When a system call is performed by an enforced process, the associated model is checked before performing the system call. For processes that do not require model enforcement (e.g., in learning model), the call is forwarded without further checking. The loading of the model and the request for enforcement are communicated via system calls. Any further attempts by an already enforced process to communicate with the model enforcer are rejected. Hence, there is no possibility for an enforced process to communicate to the model enforcer. This ensures that an enforced process cannot break out of the model enforcer’s control.

2.6.2 Checking System Calls

Before the current system call of an enforced process can be checked, the backtrace of the system call is extracted from the processes stack. This backtrace is compared with the list of possible next states. If no state matches the backtrace, the model enforcer stops the enforced process. If a state matches the system call and its arguments are checked according to the system call model. If this check succeeds, the system call is performed and the model state is updated. All possible next states are computed. If necessary, the
Enforcing Dynamic Personalized System Call Models

<table>
<thead>
<tr>
<th>Attacks</th>
<th>Without SwitchBlade</th>
<th>With SwitchBlade</th>
</tr>
</thead>
<tbody>
<tr>
<td>succeeded</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>failed</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>detected &amp; aborted</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>crashed</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2.2: Result of Wilander’s testbed [133] running natively and under SwitchBlade.

If the check does not succeed, i.e., if a process violates the current model, the system call is not performed. Instead, the process is stopped but not immediately killed. The parent process stays in taint mode and waits for the child process to terminate or being stopped. The parent process is informed when the child is stopped. Now it kills the child and forks a new child. But this child will stay in taint mode to replay the last requests for verifying that the detected model violation is no false alarm.

The model enforcer can deal with address space randomization. When a process wants to be enforced, the model enforcer extracts the mapping of the dynamic linked libraries to address ranges. This facilitates the comparison of the backtraces of the model (in which an address is represented as an offset and an unique ID of a dynamic linked library) and the backtraces extracted from the stack (in which addresses are converted to offsets and unique ids of dynamic linked libraries).

2.7 Evaluation

Our evaluation of SwitchBlade tests how far we have achieved our three goals: low false alarm rate, low false negative rate and, good performance. Therefore, we ran a series of micro benchmarks and applied our approach to the Apache web server. As expected, we experienced a number of false alarms during system model enforcement. However, we did not observe any false alarms during replay in taint mode in any of our runs.

All experiments were performed in a virtual machine with 512 MB memory running on an Intel Core 2 Duo with 2 GBytes RAM. We used Ubuntu 7.04 as operating system.

2.7.1 Synthetic Exploits

The Wilander Testbed [133] is a set of 18 vulnerabilities and exploits. In our environment and without SwitchBlade of the 18 exploits only six were successful, four resulted in crashes and 8 had no effect. SwitchBlade detects and confines all six of the otherwise successful exploits.

We used the Wilander's testbed with our generic check pointing wrapper (see Section 2.5.3). We generated an initial system call model for the run of the testbed by printing the help messages of the testbed – the only mode in which no attack is attempted. The resulting model consists of 27 states and 26 transitions. All 26 transitions
2.7 Evaluation

![Figure 2.10: Overhead of SwitchBlade's normal mode (i.e., system call enforcement) and executing the same requests in SwitchBlade's taint mode compared to a native execution without SwitchBlade. We measured the average connection time as seen by a client connecting to the proxy for different kinds of content.](image)

were write system calls which resulted from calling the LibC function `fprintf`. The 27th state is the end state. Each run that performed an attack violated the system call model and was replayed in taint mode. Table 2.2 shows the result of our experiments. It demonstrates that SwitchBlade was able to detect and abort all of the six previously successful attacks. Note that the behavior of all other executions was not affected by SwitchBlade.

### 2.7.2 Apache

We applied SwitchBlade to the Apache Webserver. First, we inserted the code for check-pointing and for replaying outstanding requests. Therefore, we added 181 lines to `server/mpm/prefork/prefork.c`. Originally, this file contained 1478 lines. The changes within the file were very localized. The effort of adding custom check-pointing and replay support to Apache therefore was very small. We expect a similar effort for other server applications (see Section 2.7.3 for GazTek HTTP Daemon). Apache supports loading plug-ins (called Apache modules) at runtime. SwitchBlade does not require any modification or recompilation of the Apache modules. Apache uses several worker processes to serve concurrent requests. SwitchBlade protects each worker separately.

Figure 2.10 shows the overhead of the average connection time for different kinds of content:

- **small static** A HTML file of 174 bytes (`index.html`).


2 Enforcing Dynamic Personalized System Call Models

![Apache Model Size](image)

Figure 2.11: Evolution of the Apache system call model for a sequence of 200 requests.

- **doc. root** The root document accessed implicitly via `/`. In contrast to small static Apache must additionally map the URI `/` to `index.html`.

- **error page** Request of an not existing file resulting in the 404 HTTP error page.

- **large static** A JPEG picture of 100,162 bytes.

- **dynamic** A hello world PHP script.

In all three configurations we used a proxy between the client and the Apache server. The overhead of SwitchBlade in normal mode (i.e., with system call model enforcement) varied between 3.8% for a static 400 byte file and 25.5% for retrieving a static 100 K byte file. If we keep the server process in taint mode instead (and without system call enforcement and without TaintCheck), the overhead varies between 50.0% and 288.5%. The overhead of taint mode is 2 times (100k static file) to 76 times (400 byte static file) larger than the overhead of SwitchBlade in its normal mode of system call enforcement.

We measured how the Apache system call model evolves with an increasing number of requests (see Figure [2.11]). For each requests that requires a new state or transition, the existing model needs to be updated. Initially, the model is updated frequently and growing fast but after about a hundred requests the model starts to stabilize. In other words, even without an explicit learning phase, one can expect that the number of model changes and, hence, switches to taint mode, will typically happen less and less often. However, as we pointed out several times, we cannot ever be sure that we reached the maximum model of a certain installation.
2.7 Evaluation

We have tested SwitchBlade against three real life exploits: two for the PHP module for Apache [114, 113], and one for the GazTek HTTP Daemon v1.4-3 (ghttpd) [4]. All three exploits were detected and aborted by SwitchBlade. Table 2.3 summarizes our experiments. We tested two different exploits for the same vulnerability in the PHP engine. One exploit injected code [114] without touching the control flow and the other one changed the control flow [113]. All three exploits violated the system call model and forced the server to replay the attacking request in taint mode. The taint analysis was able to detect and block all three exploits.

For ghttpd, we had to add some code for check-pointing and replay. We added 133 lines to main.c which originally contained 235 lines.

To illustrate how the exploit violates the system call model, we combined ghttpd’s system call model with the model of the exploit. Figure 2.12 shows the combined model.

<table>
<thead>
<tr>
<th>Application</th>
<th>Vulnerability</th>
<th>Exploit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache + PHP 4.4.4</td>
<td>double free in garbage collector</td>
<td>inject code</td>
</tr>
<tr>
<td>Apache + PHP 4.4.4</td>
<td>double free in garbage collector</td>
<td>control flow manipulation</td>
</tr>
<tr>
<td>GazTek HTTP v1.4-3</td>
<td>stack overflow in function Log</td>
<td>control flow manipulation</td>
</tr>
</tbody>
</table>

Table 2.3: Results of testing SwitchBlade against real world exploits.

Figure 2.12: System call model of ghttpd including model of the exploit (bold).
2 Enforcing Dynamic Personalized System Call Models

The model without exploit is show with the dotted edges. The bold edges show how the request with the exploit is processed by the application. First, the request follows the system call model from $l1$ to $l0$ (with a loop). But second, the request processing deviates from the system call model after $l0$ with a system call $execve$ executed from backtrace $X$. This deviation is a system call model violation. This violation is detected and $httpd$ aborted. Thus, $execve$ leads to the end node. After aborting our generic wrapper library forces the last request of $httpd$ to be replayed in taint mode. The exploit was also discovered in taint mode. Hence, the system call model was not updated. A detection of an exploit in taint mode results in the abort of the child process and the dropping of the corresponding connection via the proxy. The parent process will then spawn a new child that will execute in enforcement mode again.

2.7.4 Micro Benchmarks

We implemented several micro benchmarks to compare the performance overhead of the system call model enforcement and the taint analysis for common command line tools. We used our generic wrapper library (see Section 2.5.3). Hence, the command line tools did not need to be changed.

As expected, the system call models were different for different input files. We first used small input files (40 KBytes) to generate models for the five utilities and second, we used much larger input files (5.7 MBytes). We observed that the system call models of grep, diff and ruby (which runs the same script independent of the input file) differ.

Figure 2.13: Change of the size of the models when changing from a small input file to a larger input file.
2.7 Evaluation

Figure 2.14: Performance overhead of system call model enforcement compared to taint analysis. We expect that the applications run most of the time in model enforcement mode.

for the small and the large input files (see Figure 2.13). This is another indication that generating a general system call model for an application can be difficult.

Figure 2.14 compares the overhead of system call model enforcement to the overhead of taint analysis for four of the tools. We exclude the runtime measurements for grep because its execution time – even for large input files – was too brief to get a reproducible measurement error. All applications were initially started in taint mode and switched to model enforcement after taking the first check-point. For measuring the performance of the taint analysis we did not switch to model enforcement mode. The model enforcement overhead varied between 18% for gzip and 81% for diff. The overhead is larger than that reported for other system call interception tools (e.g., SysTrace [93], AppArmor and SELinux). This has two reasons:

- Our system call model is much more fine-grained. All system calls must be intercepted to minimize the possibilities for the attacker to make an attack conform to the model and to reduce the detection delay. Whereas other system call interception tools are only interested in the security related subset of the system calls.

  However, our goal is to detect an attack as soon as possible, even if the system call performed by the attack is not security relevant.

- For each system call, the backtrace must be computed.

In contrast to Apache (our primary focus) the enforcement overhead for the four measured utilities is more pronounced. Apache spends more time doing IO than the four command
2 Enforcing Dynamic Personalized System Call Models

line utilities. The enormous overhead of the taint analysis has two sources:

- We have not yet optimize our taint-analysis for performance as we use it only as fall-back.
- Additionally, taint analysis does also track the data flow of system call return values and constants.

However, we expect that the applications run most of the time under the model enforcement mode with much less overhead.

2.7.5 Model Size

The model generated in [104] used the first return address on the stack that points within the application to identify a system call. Our backtrace uses all return addresses on the stack to identify a node. In that way, our model becomes more fine grained. For example, the model contains parts associated to individual calls to dynamically linked libraries that issue system calls. For instance, our model of a call to the library function `gethostbyname` contains 26 distinct states with 28 transitions. Whereas, a model that only uses the return addresses within the application has only one state where all transitions are attached to. An attacker gaining access to the library implementing `gethostbyname` can issue all system calls of the 28 transitions in an arbitrary order without violating the model. This is not possible with our system call model because we use all return addresses. Therefore all 28 system calls get individual nodes and an attacked cannot freely call them any more.

To quantify the impact of the number of return addresses used to identify a node in the model, we generated different models for the Apache worker process by restricting
2.7 Evaluation

Figure 2.16: Growing size of the system call model of Vim. Each step contains an action like saving the file, calling external applications, and using the online help.

the backtrace length. Figure 2.15 shows the number of states and transitions and the maximum and average out-degree of the nodes of the generated models. The longer the backtrace, the larger the number of states and transitions. The out-degree of the states shrinks with a growing length of the backtrace. This indicates that models based on longer or unrestricted backtraces will be more restrictive. A more restrictive model will make it more difficult for an attacker to perform system calls without violating the system call model. In practice, our tool determines the length of the backtrace during initial tracing so that the number of states in the model is maximized.

2.7.6 Stateful Application

We have tested SwitchBlade with the text editor Vim. The state of Vim is the content of the currently edited file. Vim performs periodic application-level check-points. Using the initial system call model results in model violation for various actions such as saving, calling external applications, and using the online help. Our wrapper detects this and restarts Vim under TaintCheck. Vim restores its check-point, we can repeat the last action, the learner updates the model and Vim is restarted under model enforcement. The model was extended with each action that triggered a model violation according to TaintCheck’s trace. Figure 2.16 shows the growth of Vim’s system call model. In each step we one or more actions such as the ones given above. The growing of the average out-degree from 1.5 to 1.9 (right Y axis) shows that there were more new control flow operations than new system calls.
2.8 Conclusion

SwitchBlade automatically hardens an application by putting it into a sandbox. The sandbox enforces that the application follows a system call model. It trades security for availability: a detected code injection is converted into an abort. A user can use SwitchBlade without any changes to the application. However, with minor changes an application developer can support SwitchBlade with a check-pointing mechanism. Our changes to Apache allow us to tolerate a code injection by spawning a new worker process for the affected worker process. The request with the exploit is dropped. For other applications SwitchBlade can be combined with toleration approaches like Rx [94] or ASSURE [108]. These toleration approaches are evaluated with simple crash detection and in contrast to SwitchBlade, successful code injection attacks cannot be detected by these approaches.

The SwitchBlade system demonstrates that speculation can be used to combine system call interception (in normal mode) and dynamic taint analysis (when checking violation of the system call model). The result of this combination is a system that has the low performance overhead of system call interception but also the low false negative and false alarm rate of taint analysis. Hence, SwitchBlade demonstrates Thesis [1]: “Speculation can be used to reduce the perceived performance overhead of error detection mechanisms.”.

Another insight of SwitchBlade is that a system call model depends on its application’s environment (Thesis [2]). The implication of Thesis [2] are:

- System call models that are build dynamically are most likely incomplete. A dynamic update mechanism like SwitchBlade is needed to avoid false alarms.

- We also have evidence that much more coarse-grained system call models like for AppArmor suffer from the same incompleteness (see Section [2.4.1]) as our fine-grained system call models. These coarse-grained system call models are written by experts that need to know the target application. Not only these can experts make mistakes, but the system call model also depends on the system libraries (like the libc) and dynamically loaded extensions. System libraries and extensions depend on individual configurations, that might not be as well known to the experts as needed to create a complete and secure system call model. Hence, even enforcement tools of coarse-grained system call models need a dynamic update mechanism like SwitchBlade.

Our evaluation demonstrates Thesis [3] “System call model enforcement combined with taint analysis can detect control flow manipulation with in average low performance overhead, low false positive, and low false negative rate.”. Of course, there is a performance trade-off. As long as the system call model needs to be updated the performance overhead is dominated by the learner and the taint-analysis. Our measurements indicate that after an initial learning phase, on-demand learning is less often needed (see Figure [2.11] and Figure [2.16]). If learning is needed rarely, then the performance overhead is dominated by the system call model enforcement, which has a negligible performance overhead compared to the overhead of the learning mode.
2.8 Conclusion

SwitchBlade constraints itself to automatically hardening against security vulnerabilities. It uses speculation without parallelization. The ParExC approach (presented in the next chapter) is more general because it does not focus on security vulnerabilities only. The core idea of ParExC is to speculatively parallelize runtime checkers.
3 Speculation for Parallelizing Runtime Checks

Error toleration approaches implemented by frameworks like Rx [94] and ASSURE [108] rely on error detection mechanism that are implemented as sensors. While both approaches can work with any kind of sensor, they are only evaluated with very lightweight crash sensors in current research papers [94, 108]. A crash sensor is practically already provided by the OS and costs no additional runtime overhead. When a sensored application crashes, a signal is sent to recovery framework. This signal just needs to be handled to mask the error.

However, crash sensors cover only a small subset of possible errors. State corruption (errors) can lead to failures – such as corrupted output – that are not detected by a crash sensor. Unsafe languages like C/C++ are particularly prone to state corrupting bugs, for instance buffer overflows. Buffer overflows cannot only corrupt state but they additionally facilitate malicious attacks. State of the art precise out-of-bounds checkers can detect buffer overflows, but they introduce slowdown up to 12x [98]. A less expensive alternative is, for example, data-flow integrity checking [22] that does not prevent buffer overflows but instead detects invalid data-flows. Since buffer overflows generate invalid data flow, data flow integrity checking effectively prevents state corruption and malicious attacks due to buffer overflows. Nevertheless, data-flow integrity checking still introduces a slowdown of 2.5x or higher.

Given the potentially very large slowdowns introduced by runtime checks, our goal is to make runtime checking more palatable by reducing its impact on the runtime of applications. Given the prevalence of multi-core CPUs and the difficulty of many applications to harness the full power of multi-core CPUs, it makes sense to use additional cores to try to mask the overhead of runtime checks. This could be accomplished by:

(a) parallelizing the application and in this way, implicitly parallelizing the checks, or
(b) by parallelizing the checks only.

The problem with alternative (a) is that parallelizing existing applications and even writing new parallel applications is very difficult and introduces new sources of software bugs resulting in, e.g., data races and deadlocks. To simplify the parallelization of applications one could use software transactional memory (STM). STM alleviates several drawbacks of lock-based and also lock-free synchronization mechanisms. However, it still requires the developer to parallelize the application itself which is a non-trivial error-prone manual process.

Alternative (b), i.e., the parallelization of runtime checks has recently been investigated [138, 97, 128, 83, 59]. While approaches like Speck [83] scale very well with the
number of cores, the performance measurements of Speck show no performance gain when compared to sequential checking. The parallel taint analysis of Speck has a slowdown of 18.4x on an 8-core CPU [83]. This is in the same order of magnitude as checking without parallelization on one core [81]. Newer sequential taint checking approaches using static instrumentation perform even better: slowdowns are between 1.58x and 2.06x [136].

Given the current state of the art, it is not clear if one should choose alternative (a) or (b), or maybe even just use sequential checks using static instrumentation? The main question is on how much we can reduce the overheads of the competing alternatives. In our analysis of alternative (b) we have identified two sources for high overheads generated by existing checkers:

1. If parallelized runtime checks hold book-keeping state, state accesses are serialized. Parallel Dynamic Information Flow Tracking (DIFT) [97] uses a non-trivial hardware extension to stream meta data to another core. Speck [83] streams taint data to a single core thereby serializing data access. Both approaches limit scalability and performance.

2. In most previous work on parallelized runtime checking checks are added by dynamic binary instrumentation (DBI). DBI introduces a high overhead that must be compensated by additional cores. Our goal is to be faster than the sequential execution of statically instrumented code. Results of current DBI-based approaches indicate that DBI is not suitable therefore. Furthermore, some checkers cannot be implemented by DBI. For instance, bounds for objects on the stack are not available in binaries without debug information. Therefore, it is not possible to build a precise out-of-bounds checker for stack objects using DBI.

Beside the performance issues, we found two functional problems in the state of the art approaches:

1. All DBI based approaches can only add checks. For instance, with DBI it is not possible to parallelize checks that are already part of an application, such as assertions and sanity checks. The only approach that is (in principle\(^1\)) able to parallelize assertions and sanity checks is FastTrack [59]. However, FastTrack imposes some restrictions. Most importantly, FastTrack can only be applied to code blocks that do not contain system calls. It is not possible make the whole application one FastTrack block. Therefore the FastTrack approach limits scalability.

2. All parallelization approaches for runtime checks use speculation. Most approaches do not support speculation for system calls and therewith limit the possibilities for speculation and parallelization. Only Speck [83] supports speculation for system calls based on Speculator [82]. But Speculator does not isolate speculatively issued system calls. Their effects propagate through the whole system. If the speculation fails, not only the checked application needs to be rolled back, but also all applications influenced by this speculatively executed application. Futhermore, the missing

\(^1\)FastTrack was only evaluated with a checker that adds checks [59].
isolation makes it difficult to have more than one application issuing speculative system calls at the same time. The difficulties are, for instance, that speculative actions of two different applications can interweave or even conflict. Interweaving makes roll back difficult too.

Based on the above four observations, we developed a new framework, ParExC\(^2\) for writing parallelized runtime checkers. It parallelizes the runtime checks only, i.e., it does not require the application itself to be parallelized. To demonstrate the effectiveness and versatility of our approach, we implemented different checkers to detect state corruptions (see Section 3.6):

- A precise out-of-bounds checker to detect buffer overflows.
- A data-flow integrity checker to detect illegal data-flows.
- A checker relying on programmer given assertions.

Furthermore, we compare ParExC with an STM-implementation using two parallelized STM-benchmarks, i.e., benchmarks for which STM implementations have been optimized. We describe how to add checks to parallelized applications using software transactional memory in Section 3.6.4 and compare the performance of an STM-based checker and the same checker based on ParExC in Section 3.7.2.

A checker instruments applications with runtime checks. When a check fails at runtime, the application is aborted. The basic approach is very similar to Speck \[83\]. We execute the original application as predictor without any runtime checks on one core. The predictor’s execution is partitioned into epochs. Each epoch is replayed with runtime checks enabled by an executor. Because of the runtime checks, the executor is typically an order of magnitude slower than the predictor for the same epoch. We achieve a speedup by running the executors in parallel to each other and to the predictor. Our approach for parallelizing runtime checks is introduced in more detail in Section 3.1.

With ParExC we make the following novel contributions to the state of the art:

1. **Parallelize runtime checks that are already part of the application.** Unlike the DBI approaches, we transform the application statically. This enables us to also optimize the predictor. Unlike FastTrack (which also uses static instrumentation) we restrict our parallelization not only to some regions. We support application wide parallelization, by our novel StackLifter (see Section 3.4). The StackLifter facilitates switching from the code base without runtime checks (run by the predictor) to the code base with runtime checks (run by the executors).

   By using static instrumentation we also avoid the overheads introduced by DBI.

2. **Generic speculation support for checker state.** At runtime checks need to maintain book-keeping state. Because, the checks are performed by the executors checks can be executed in parallel to each other. Thus, checks need means to synchronize their state. Our approach is to use speculation to sync the book-keeping state. Therefore, we propose our novel speculative variables (Section 3.5).

\(^2\)This chapter is based partly on [121][122][120]. ParExC is called Prospect in [120].
3 Speculation for Parallelizing Runtime Checks

Figure 3.1: Our parallelization approach executes a fast variant on one core. The execution of the fast variant is partitioned into epochs. Each epoch is re-executed with a slow variant with more functionality. The re-execution happens in parallel to the fast variant on multiple cores.

3. **Isolated system call speculation.** The predictor’s execution is speculative until it is checked by the executors. Thus, if the predictor write to a file, the write itself is speculative until the computation is confirmed by the runtime checks in the executors. Our goal is that users see no speculative output. Usually applications perform output by system calls. Most approach either ignore or forbid system calls [138, 97, 128, 59]. Only Speck [83] supports speculative system calls. However, as outlined before Speck does not isolate the speculation from the rest of the system. Our system call speculation (see Section 3.3) isolates applications from each other. Thus, and in contrast to Speck, on miss-speculation only the affected application has to be rolled back. Furthermore, we support multiple parallelized applications running in parallel.

In our evaluation in Section 3.7 we show that parallelization of runtime checks can reduce the perceived performance overhead of expensive error detection (see Thesis 4). Additionally, we demonstrate that parallelizing runtime checks can be faster than parallelizing an application itself with the help of STM (see Thesis 5). Because the ParExC approach is heavily based on speculation, we also show Thesis 1 that speculation helps to improve the performance of runtime checking.

3.1 Approach

ParExC uses the *predictor/executor* approach of [138, 83, 59] to parallelize an application. Figure 3.1 (i) shows a *fast variant* and a *slow variant* derived from the same code base. For example, the fast variant can be the original application and the slow variant can be the original application including additional runtime checks. Our goal is to provide the functionality of the slow variant while not exceeding the fast variant’s runtime. To achieve this, the ParExC framework parallelizes the execution of the slow variant (see Figure 3.1 (ii)). At runtime, ParExC executes the fast variant in a *predictor* process that is used to compute future states of the slow variant. The execution of the predictor is
3.1 Approach

The state of the predictor at the start of an epoch is used to spawn an executor process. The executor re-executes its epoch using the slow variant and the predictor state from which it started. We can parallelize the checked application by running the individual executors and the predictor in parallel. The maximum possible speedup is the execution time of the slow variant divided by the execution time of the fast variant.

At each epoch boundary, ParExC takes a snapshot of the fast variant. The snapshot is similar to a UNIX fork. The fast variant continues its execution. The forked executor switches from the fast variant to the slow variant and starts executing the epoch in the slow variant. At the end of an epoch, the slow variant terminates, whereas, the fast variant forks the next epoch.

ParExC is not completely transparent for the application developer. The application developer has to ensure that the application calls parexc.chkpnt periodically and preferably, with a constant frequency. This function starts a new epoch and is provided by the ParExC runtime.

ParExC has two major components:

1. the ParExC compiler that generates the fast and/or slow variant of the given application, and

2. the ParExC runtime that provides deterministic replay for re-executing the slow variant in the executors and speculative execution for the fast variant in the predictor. The speculative variables used to sync the book-keeping state of the checker between the, in parallel running, executors is also part of the ParExC runtime.

3.1.1 Compiler Infrastructure

The slow and fast variant are generated by the ParExC compiler from the original application. We consider three cases of variant generation.

1. The original application is the fast variant. The slow variant is generated by adding additional code (like runtime security checks) to the original application [83].

2. The original application is the slow variant. The fast variant is generated by removing code from the original application. For instance, aggressive but potential unsafe optimizations can remove code [59].

3. The first and the second approach can by combined, i.e., both variants are generated from the original application.

Figure 3.2 shows the work-flow of the ParExC compiler. Our novel StackLifter (Section 3.4) takes the original application’s code and generates the initial versions of the fast and the slow variant. In particular, it prepares both variants in a way that it is possible to switch from the fast variant to the slow variant at epoch boundaries.

Both variants can then be instrumented, e.g., to remove existing functionality such as sanity checks or add error detecting functionality (see Section 3.6). We cannot allow
arbitrary instrumentation of the fast and the slow variant. The instrumentation must preserve the state equivalence property:

**Definition 3.1.** The application state of the fast variant at the end of epoch \( e \) must be equivalent to the application state of the slow variant at the end of epoch \( e \).

This property ensures that from an external point of view, the parallel execution of the slow variant is equivalent to the sequential execution of the fast variant. We currently do not enforce the state equivalence property. We show in Section 3.5 how to partly circumvent this restriction by using speculation.

The *fast variant generator* and *the slow variant generator* are provided by the checker developer. In our experience, the instrumentation process does not need to be aware of the parallelization. For instance, we can use the same instrumentation for checking for out-of-bounds errors for parallelizing with ParExC, and STM as well as without any parallelization at all. But, we need different implementations for managing the bookkeeping state of the checker, e.g., the bounds of the allocated buffers.

ParExC’s *framework generator* generates framework code that connects the code bases of both variants. The framework code primarily consists of a new `main` function. This
3.1 Approach

Figure 3.3: Overview of ParExC’s architecture at runtime. Deterministic replay and speculative system call execution is provided as part of the OS kernel. The checker runtime has access to speculative variables to manage the checker’s book-keeping state. Both, the fast variant and the slow variant (through the checker) have access to the epoch management.

main function sets up the ParExC runtime, any additional runtime that was added for the slow variant, and starts the first epoch in predictor and executor.

We have implemented ParExC using the LLVM compiler framework [65]. StackLifter and the variant generators are LLVM compiler passes. In general, the approach itself is not restricted to LLVM. ParExC can be ported to other compiler frameworks.

3.1.2 Runtime Support

Figure 3.3 gives an overview over ParExC’s architecture at runtime. The fast variant has access to ParExC’s epoch management via `parexc_chkpt`. The executor (and therewith the slow variant) additionally needs to approve its epoch after completely checking it.

ParExC performs external actions of the fast variant speculatively [83]. For instance, `write` system calls are held back until they are re-executed and approved by the slow variant. Hence, we ensure that external actions only become visible after the slow variant has verified them.

To support the state equivalence property we use deterministic replay [110] for re-executing the slow variant. ParExC records all non-deterministic external events that happen in the fast variant of epoch $e$. When the slow variant of epoch $e$ is re-executed, ParExC replays the recorded events. For example, the time value returned by the `gettimeofday` system call in the fast variant is also returned in the slow variant for the respective call to `gettimeofday`.

We have implemented the speculative execution and deterministic replay for Linux as a kernel module (see Section 3.3). In contrast to ParExC, Speck [83] provides system wide speculation, i.e., external actions of the fast variant are speculative propagated to other
processes. In case of an abort all processes containing speculative state are rolled back. In contrast to Speck, our kernel module isolates multiple applications running under ParExC at the same time from each other.

Our runtime support also contains speculative variables to circumvent the state equivalence property for checkers. Some checkers hold additional book-keeping state required for the checking. The state is managed by the checker-runtime given by the checker developer. This book-keeping state is only needed in the slow variant. But the state equivalence property requires that the book-keeping state has to be in the fast variant too. For instance, the checker runtime of an out-of-bounds checker needs to keep track of the size of the allocated buffers. The straightforward solution would be to track allocations in both variants (such as in FastTrack [59]). However, we want to avoid any additional overhead in the fast variant. Therefore, checker developers can use speculative variables to maintain book-keeping state in their checker runtime (see Section 3.5).

3.2 Related Work

The predictor/executor approach was first introduced by the hardware community [138]. A distilled program (i.e., a fast variant) is generated at compile time. The execution is similar to our approach. The main difference is on how the switch from the fast variant to the slow variant at epoch boundaries is performed. Zilles et al. [138] use a hash-map to translate program counters. States are communicated via a check-pointing unit located in the memory subsystem. However, it is not clear how this approach handles different stack layouts between fast and slow variant.

FastTrack [59] is the system most similar to ParExC. In contrast to ParExC, FastTrack is not designed to apply the parallelization to the whole application. It only supports fast-track regions that must not contain system calls. Fast-track regions raise composability questions. Is it possible to nest fast-track regions? On one hand, FastTrack does not need to switch between fast and slow variants code base at arbitrary locations nor does it need speculative system calls. But on the other hand, FastTrack’s approach of the fast-track regions makes it impossible to apply the parallelization to the whole application, which is our goal. To do so, it is necessary to have an approach, such as our StackLifter, to switch from the fast variant to the slow variant at arbitrary points. By applying the approach to the whole application we avoid the composability issues of FastTrack. Additionally, FastTrack makes no use of speculation for state added to the slow variant. Instead, FastTrack tracks all necessary checker state in the fast variant and therewith slows down the whole parallelization approach.

In the last few years the predictor/executor approach got some attention from the runtime checking community [128, 83, 97]. All these projects make use of dynamic binary instrumentation with the help of the Pin tool [67]. Whereas Pin allows adding code like runtime checks, it is not suitable for efficiently removing code like user-defined assertions. A StackLifter is not needed because the state of the runtime checks is completely separated from the application state. SuperPin [128] does not support speculation for system calls. Hence, speculative state might become visible as value failure, e.g., due to failing runtime
3.2 Related Work

checks or unsafe optimizations. Speck’s [83] support for system call speculation and deterministic replay is closest to ours. It is derived from Speculator [82]. Speculator is an operating system extension and it supports the speculative execution of one process. The speculation is propagated throughout the system, whereas, ParExC provides isolation. ParExC’s isolates the speculatively issued system calls of an application form the rest of the system until these system calls have been committed. Thus, it is possible to run several applications under ParExC at the same time. Speck’s and our replay support is very similar to Flashback [110]. DIFT [97] uses a non-trivial hardware extension to stream data from the core running the fast version to slave cores. Slave cores use this data to perform runtime checks.

The problems solved by the StackLifter are similar to the dynamic software update problem. The current version of an application shall be replaced by a new version without terminating or restarting the application [76, 68]. Previous work, for instance Ginseng [76], avoids stack rewriting by only upgrading a function that has currently no frames on the stack. Data is accessed solely indirectly to allow online updates. The recently published UpStare [68] uses an approach similar to StackLifter. However, UpStare works on C source code and seems to need manual intervention for mapping an old version’s stack frame to a new version’s stack frame, e.g., when pointers are involved. We avoid this issue by using the alloca stack (see Section 3.4.4) and with the help of the state equivalence property. The state equivalence property requires that there are no state changes that require manual mapping of stack frames.

ParExC uses speculation in two ways:

1. It speculates on the error free execution of the predictor, and

2. uses speculative variables to decouple executors.

Both ways of speculation have been used before in thread-level speculation [87] [111] [112], by transactional memory [21] [40] [54] and to implement speculation on synchronous IO [82]. Thread-level speculation tries to exploit parallelism in sequentially programmed applications. Therefore, applications are divided into parts similar to our epochs. This requires either code analysis or hints by the programmer [87]. The obtained epochs are characterized by no or minimal data dependencies and are executed in parallel [111] [112]. Our speculative variables are similar to value speculation used in thread-level speculation [111] [112] [91]. Transactional memory (TM) [21] [40] [54] provides atomicity for critical regions. Optimistic TM implementations speculate on low contention between concurrently executed critical regions. If a conflict between two concurrently performed critical regions is detected, at least one of the critical regions is rolled back and its changes are undone. ParExC speculates on a failure free execution. Thus, we abort the application as soon as we detect a failed speculation. Nightingale et al. [82] use speculation to turn synchronous file system write accesses into asynchronous write accesses. That basic idea is to speculate on a successful execution of the asynchronous write access. The application continues speculatively until the acknowledgment from the file server arrives.
3 Speculation for Parallelizing Runtime Checks

3.3 Deterministic Replay and Speculation

To prevent the fast variant from doing output for computation that was not yet checked by the slow variant, the fast variant performs every externally visible side-effects *speculatively*. The output done by the fast variant in epoch \(e_i\) is held back by ParExC until the checker (slow variant) explicitly approves \(e_i\). This is done without blocking the fast variant, i.e., the fast variant gets a speculative return value from ParExC’s runtime system. Because of the speculative side-effects an attacker cannot exploit a vulnerability and manipulate the fast variant output without being detected by the slow variant before the output becomes visible. Because the fast variant cannot be trusted, the ParExC speculation support is part of the OS.

Our parallelization approach is based on the assumption that the slow variant *deterministically* replays the same execution as the fast variant. Therefore, the same input values passed to the fast variant have to be passed to the slow variant too. Our solution is to record all input pass to the fast variant in a log. Whenever the slow variant reads input we replay the input from this log. ParExC’s deterministic replay support is also part of the OS.

For our deterministic replay and speculation support we make the same assumption as [110, 83]:

**Assumption 3.1.** All application IO is performed via system calls.

Therefore, we map all user-space non-determinism (such as `gettimeofday` on some systems) to system calls. For example, for `gettimeofday` we provide a wrapper that always uses a system call.

All system calls executed in the fast variant in epoch \(e_i\) are put into a log \(L_i\). A log entry comprises the system call ID and the system call’s arguments. System calls with external effects are also logged in \(L_i\) but not executed. Most other system calls are executed immediately. The only exception are system calls the depend on system calls with external effects. For instance a `read` system call is performed and the read data is logged for later replay in the slow variant. On the other hand, a `write` is only put into the log but not executed. A `close` following the `write` on the same file descriptor depends on the `write`. Therefore, the `close` is treat like the `write`. When the slow variant performs a `read` system call, ParExC replays the result from the log. When the slow variant executes a `write` system call \(s_w\), ParExC will check that \(s_w\) used the same argument values as the fast variant. External events of epoch \(e_i\) must be externalized after \(e_{i-1}\) committed. Thus, ParExC cannot yet commit the system call \(s_w\) if epoch \(e_{i-1}\) has not been committed until the time \(s_w\) was issued.

To avoid unnecessary delay the replay for the slow variant of \(e_i\) can immediately start after the fast variant started with \(e_i\). After epoch \(e_{i-1}\) has been committed by the slow variant, and epoch \(e_i\) was successfully checked all system calls with external side-effects in \(L_i\) are performed.

With the help of log \(L_i\) ParExC also checks that the slow variant deterministically replays the fast variant, i.e., the slow variant must perform all system calls in \(L_i\) with the logged arguments and in the logged order. In that way an attack that solely compromises
the fast variant is detected when it manipulates the fast variants output but not the slow variants output. An attack on the slow variant should by detected by the slow variants runtime checks.

The slow variant can perform additional system calls that bypass $L_i$. These system calls use special system call ids that ParExC excludes from checking. For example, these special system calls allow to implement logging. The set of system calls, allowed to bypass replay and checking, can be restricted for security. Without this restriction an attacker might be able to abuse these system calls for her goals, because, unlike replayed and checked system calls, the side-effects of theses system calls become visible before the slow variant approves the current epoch.

Due to executors ran in parallel to each other, system calls at the start of epoch $e_i$ in the slow variant might be executed before system calls at the end of $e_{i-1}$ in the slow variant. ParExC supports an out-of-order execution of system calls with an in-order retirement of system call side effects. We accomplish this by committing all side-effects of $L_i$ only after $e_{i-1}$, and therewith $L_{i-1}$, was committed.

The speculation support and the deterministic replay support are both implemented in a Linux kernel module and do not affect other parts of the kernel.

### 3.3.1 Interface

Our implementation exports the following interface for checkers and parallelized applications:

- **parexc_chkptn**: This function is only allowed in the fast variant. It marks the border of epoch $e_{i-1}$ to epoch $e_i$ in the fast variant. Additionally, it forks an executor for the slow variant of $e_i$. Like `fork` it returns twice. Once, in the fast variant and once in the newly created executor that will execute the slow variant of $e_i$.

When the fast variant calls `exit` or `exit_group` the current epoch $e_i$ is finished, but no new epoch is started. The fast variant blocks until the slow variant has committed $e_i$. Next the whole application terminates.

For each epoch $e_i$ a new log $L_i$ is created. Every system call in the fast variant until the next `parexc_chkptn` is recorded in $L_i$. As described above ParExC replays and checks all system calls in $e_i$’s slow variant with the help of $L_i$.

This is the only function needed by developers of applications to parallelize, i.e., the remaining interface is only used by the checkers.

- **parexc_replay_begin**: After forking the executor, the executor first switches with the code generated by the StackLifter to the code base of the slow variant (Section 3.4). The checker initializes the speculative variables for current epoch $e_i$. Then the slow variant calls `parexc_replay_begin` to start the replay of the system calls recorded in log $L_i$.

If a system call issued by the slow variant between `parexc_replay_begin` and `parexc_replay_end` does not match the next system call in $L_i$ the execution
3 Speculation for Parallelizing Runtime Checks

is aborted. The only exceptions are a restricted set of system calls that may be issued by the checker, and bypass replay. These system calls use previously unused system call ids. After executing a set of security checks (to prevent malicious abuse), these system calls are forwarded by changing their system call ID to the originally intended. For example, if the slow variant wants to allocate some private memory, it issues a system call with the ID sys_parexc_mmap. This system call is intercepted, checked and then forwarded to system call sys_mmap.

If the slow variant issues a system call that is not yet in log \( L_i \) the slow variant blocks. That an system call is not yet in \( L_i \) can happen because: (1) The slow variant overtook the fast variant or (2) the executions of the slow variant and the fast variant diverge. In case (1) the slow variant waits until the fast variant has caught up. In case (2) the execution must be aborted. Therefore, the slow variant blocks either until the fast variant issues the next system call or the fast variant ends the current epoch with a call to parexc_chkpnt or exit. In the former case, the execution continues as usual with comparing the system call ID and the arguments. In the latter case, the execution is aborted because the slow variant tried to perform a system call in \( e_i \) that was not performed by the fast variant in \( e_i \). This function is not allowed in the fast variant.

- **parexc_replay_end**: The current epoch lasts until the next call to parexc_chkptnt. Our compiler replaces each call to parexc_chkptnt in the slow variant’s code base with a call to parexc_replay_end. Hence, instead of calling parexc_chkptnt the slow variants ends the current epoch by calling parexc_replay_end. The function parexc_replay_end is also only allowed in the slow variant. It stops replay. If necessary it waits until the fast variant finishes the current epoch \( e_i \). Then it checks that the slow variant has replayed every system call in \( L_i \). If the check fails the application is aborted. Otherwise, the slow variant continues with checking its speculative variables (see Section 3.5).

The slow variant can execute arbitrary system calls after the call to parexc_replay_end. These system calls might be needed for logging or checking the speculative variables.

- **parexc_approve**: Only if the speculative variables contain no mis-speculation, the current epoch \( e_i \) can be committed. To commit \( e_i \) the slow variant calls parexc_approve. This function is not allowed in the fast variant.

The function parexc_approve first blocks until the preceding epoch \( e_{i-1} \) was approved. This ensures the in-order retirement of the out-of-order executed system calls. Then all speculatively performed system calls in \( L_i \) are performed in the order they appear in \( L_i \). Log \( L_i \) is then freed.

For deadlock avoidance (Section 3.5.2) parexc_approve can also be called before parexc_replay_end, i.e., parexc_approve can be called within the execution of the epoch in the slow variant. Then all outstanding external actions up to this
Deterministic Replay and Speculation

All further speculative system calls of this epoch are made visible as soon as they are replayed.

- **parexc_exit**: After any outstanding external action of epoch \( e \) has been committed by parexc_approve the slow variant terminates the executor by calling parexc_exit. The slow variant does never call parexc_chkpt.

Before terminating the current executor, parexc_exit checks that the slow variant called parexc_replay_begin, parexc_replay_end and, parexc_approve. If not, the application is aborted.

On abort all out-standing external actions are discarded. Hence, no unchecked output will become externally visible.

### 3.3.2 Implementation

Our replay and speculation support is implemented as a layer between the user-land and the system call implementations in kernel space. Our implementation uses the utrace [72] framework. The utrace interface is very similar to ptrace. In contrast to ptrace, utrace allows our layer to reside in kernel-space. Thus, we avoid additional context switches and the expensive copying of data between two address spaces as needed by ptrace [99].

We hook into syscall_enter and syscall_exit. Both hooks run in kernel space. They are called before and after a system call has been performed, respectively. Our implementation keeps also track of the current Linux task to distinguish the predictor (fast variant) and the executors (slow variant).

The system call recording can replay needs semantic information about each system call: What are the number of arguments to record? Does the system call need speculative execution? And, does the system call write to user space? We have implemented a secondary system call table to dispatch from the syscall_enter and syscall_exit hooks into system call specific implementations. The following description generalizes the specific hook implementations.

System calls are recorded in the fast variant in the syscall_enter hook. The syscall_enter hook allocates a log entry and stores the system call ID and the system call's arguments into the log entry. The syscall_exit hook stores the return value and any memory copied user space (for example for read) into the log entry. Next, syscall_exit appends the log entry to the current system call log.

The speculative execution of a system call (for instance write) in the fast variant skips the actual execution of the system call (a utrace functionality). But, first ParExC calculates a speculative return value from the arguments. For instance, for a write with a invalid file descriptor argument EBADF is computed as return value. If all arguments are valid the length of the data to write is computed as return value. The speculatively computed return value is stored in the log entry. Second, the system call is skipped and the pre-computed return value is returned.

---

3The primary system call table is maintained by the kernel to dispatch system calls.
3 Speculation for Parallelizing Runtime Checks

Figure 3.4: Translating the call stack at runtime.

For every system call in the slow variant between parexc_replay_begin and parexc_replay_end: in syscall_enter the next log entry is fetched and set as current log entry. If necessary syscall_enter blocks until the next entry is available. The system call ID and the arguments are checked against the current log entry. Depending on the system call, the actual system call is either skipped or performed. Most system calls are skipped. Only calls that read deterministic data or manage process local state (such as brk, and mmap) are actually performed. If skipped, the return value of the system call and memory copied to user space (for example for read) are replayed from the current log entry. If the system call is performed its return value is checked against the current log entry. In replay speculatively issued system calls (such as write) are skipped if the current epoch was not yet approved by parexc_approve. If the epoch is approved, speculatively issued system calls are immediately performed when replayed. If the epoch is not yet approved, the speculatively issued system calls will be performed in parexc_approve (see above).

The interface described in Section 3.3.1 is implemented utilizing unused system call ids. Because syscall_enter intercepts every system call, we reroute the interface’s system calls in the syscall_enter hook to our implementation without passing them further down into the kernel.

3.4 Switching Code Bases

One main goal of ParExC is to enable the variant generators to instrument slow and fast variants independently of each other. Therefore, we provide the variant generators with two separate code bases – one for the fast variant and one for the slow variant. The issue is that we need to switch from the fast variant to the slow variant at the start of an epoch. This is difficult because, for instance, a generator might remove some temporary variables from the fast variant or add new temporary variables to the slow variant. The StackLifter’s purpose is to instrument the two code bases to facilitate the switch from the fast to the slow variant.
3.4 Switching Code Bases

Variant generators may instrument a fast function \( F_{\text{fast}} \) and its slow equivalent \( F_{\text{slow}} \) in a way that \( F_{\text{fast}} \) and \( F_{\text{slow}} \) have different stack layouts. Figure 3.4 illustrates the different stack layouts for three functions: \emph{main}, \( F \), and \( G \). After the StackLifter run, a variant generator changed the order of variable definitions in \emph{main} and added a new variable \( A \) to \( G \).

The predictor executes the fast variant and the executors the slow variant. An executor receives its initial state from the predictor. Hence, at that point in time all stack frames on the stack belong to fast variant. ParExC needs to translate the stack frames of from fast functions to their \emph{slow} counterparts. The translation should be transparent to the application. Therefore, after the translation the call stack has to look like as if only slow functions had been executed. After translating the call stack, execution can continue normally. Global data and the heap do not need to be translated, because the state equivalence property requires the variant generator not to change the heap layout or the layout of global data.

Stack translation starts with setting the \emph{doUnwind} flag. Then the call stack is traversed up to the outermost function, i.e., usually to \emph{main}, saving all necessary information for each stack frame to allow reconstruction (Figure 3.4 step (1)). Once the top of the stack is reached (step (2)), we rebuild the call stack by calling the slow variant of each function. We use the saved information to rebuild the call stack (step (3)). When reaching the point where the stack lifting was triggered, execution resumes normally.

We use LLVM to perform the necessary modifications, i.e., to add code that saves and restores registers. All code modifications are performed statically. The input and output format is LLVM’s intermediate representation. Therefore, our StackLifter is independent of the underlying hardware platform.

3.4.1 Example

For an application programmer the main difference between FastTrack and ParExC is the programming interface. In FastTrack the programmer specifically starts and ends a fast-track region. This region is translated to an \texttt{if} branch. One branches is executed in the fast variant and the other in the slow variant.

In ParExC we support application wide instrumentation and do not want to restrict instrumentation to a set of regions. In our current implementation the programmer has to mark possible places where a new epoch could be spawned by calling \texttt{parexc_chkpt}. Listing 3.1 shows an example. Like \texttt{fork}, a call to \texttt{parexc_chkpt} issued in the fast variant returns twice: (1) in the fast variant, and (2) in the slow variant. When \texttt{parexc_chkpt} is called at runtime from the fast variant and returns into the slow variant, \texttt{parexc_chkpt} needs to switch from the fast variant’s code base to the slow variant’s code base.

Listing 3.1: API example and the problem of switching code bases.

```c
/* application as seen by developer */
int foo() { ... parexc_chkpnt(); ... }
void bar(char* b) {
```
3 Speculation for Parallelizing Runtime Checks

Listing 3.1 gives a small example of how the ParExC API should be used and the problems that arise. Lines 1 to 7 show an API example. Function foo calls parexc_chkpnt to start a new epoch. Function bar calls foo. In the fast variant’s code base (lines 9 to 14) the variant generator removed the assert. The slow variant’s code base (lines 16 to 22) is unchanged apart from function renaming.

For switching the code base from the fast variant to the slow variant it is not enough to jump from parexc_chkpnt_fast right behind the call to parexc_chkpnt_slow in foo_slow. In the code base of the slow variant, b might be loaded into a machine register in line 19. When foo_slow returns b is expected to be in this register in line 21. Whereas, in the fast variant the assert does not exist and when foo_fast returns b is still on the stack in line 13. That means after the switch from parexc_chkpnt_fast to parexc_chkpnt_slow, b has to appear in the right register when foo_slow returns.

The StackLifter adds the necessary code modifications that put b into the right registers. The general work-flow is that the StackLifter prepares both variants before the variant generators instrument the variant’s code bases. The next subsections give further details on each transformation step of the StackLifter.

### 3.4.2 Integration with parexc_chkpnt

The function parexc_chkpnt is the programming interface of ParExC. However, while a developer using ParExC sees one parexc_chkpnt only, we have three versions of parexc_chkpnt. The parexc_chkpnt inserted by the developer becomes replaced by parexc_chkpnt_fast in the fast variant and parexc_chkpnt_slow in the slow variant. The third parexc_chkpnt is the system call introduced in Section 3.3.
3.4 Switching Code Bases

system call parexc_chkpnt is only called by parexc_chkpnt_fast, but never directly by the application developer.

The functions parexc_chkpnt_fast and parexc_chkpnt_slow control the code base switching at runtime. These functions make use of the instrumentations inserted by the StackLifter.

• parexc_chkpnt_fast This function is called in the fast variant. It calls the parexc_chkpnt system call (Section 3.3). After forking the new epoch, the flag doUnwind is set in the executor and parexc_chkpnt_fast returns. Now the registers saving blocks of parexc_chkpnt_fast’s callers including main_fast are executed (see Figure 3.4). The register saving block of a function saves the current state of the function’s stack frame. Function main_fast returns to our newly generated main function (see Section 3.1.1). Our generated main now calls the slow variant’s main_slow and in this way, the states of all slow functions are restored until parexc_chkpnt_slow (see Figure 3.4). The Register restoring basic block restores the previously saved stack frame of a given function.

• parexc_chkpnt_slow In parexc_chkpnt_slow the flag doUnwind is cleared and execution continues normally in the slow variant. If the next parexc_chkpnt_slow function is reached in the slow variant replay is ended and the epoch approved. After approval the executor terminates with parexc_exit.

3.4.3 Code Transformations

Listing 3.2 shows an example for the input given to the StackLifter. Function main calls parexc_chkpnt. The StackLifter generates two outputs:

• The code base of the fast variant (see Listing 3.3).

• The code base of the slow variant (see Listing 3.5).

Listing 3.2: Input to the StackLifter.

```c
int main (...) {
    ...
    parexc_chkpnt ();
    ...
}
```

At runtime, the instrumentation of the fast variant needs to check after each function call if a code base switching is in progress, i.e., if the global flag doUnwind is set. To initiate code base switching parexc_chkpnt_fast sets the doUnwind flag. Function parexc_chkpnt_fast is part of the framework code in Listing 3.4 The framework code is generated by ParExC’s framework generator (see Section 3.1.1). If the code base needs to be switched, the fast variant saves all currently life registers. The registers are saved by pushing them to the code base switching stack. This stack is globally allocated. There
is exactly one code base switching stack for the whole application. After pushing the life registers an ID for the current location is pushed to the stack too. Next, the function returns to the caller which repeats the procedure for its stack frame.

Function main_fast returns to our main function (see Listing 3.4). Our main is part of the framework code. At runtime, if code switching is in progress main calls main_slow. Otherwise, main just returns with main_fast’s return code.

Listing 3.5 shows the slow variant’s code base for the input from Listing 3.2. In the slow code base each function needs to check at its entry if code switching is in progress. If not, the function continues at its original entry point. If code switching is in progress, the functions pops the location of the call instruction to switch to from the code switching stack. Then it restores the life registers from the stack too. The set of life registers depends on the popped call location within the function. This set is exactly the same as the one that was pushed in the fast variant for the given call location. After restoring the registers the slow variant jumps to the call instruction matching the popped location.
Next, the call is performed and the callee restores its stack frame. The process finishes with a call to parexc_chkptnt_slow which resets the doUnwind flag (see Listing 3.4).

Listing 3.5: StackLifter output: Code base of slow variant.

```c
int main_slow () {
    if (doUnwind) {
        switch (<pop location> ) {
            ...
            case call_01_location:
                <pop life registers>
                goto call_01;
                ...
            }
        }
        ...
    call_01:
        parexc_chkptnt_slow ();
        ...
}
```

We have one optimization. StackLifter applies the transformation only to functions and call instructions that are on a path to parexc_chkptnt in the original input. In the following we give more details about the instrumentation on the LLVM level.

**Function Naming**

The StackLifter gets the original application’s code base as input. Hence, the first step is to generate a copy of this code base. StackLifter clones all functions defined in a given module. Each function then comes in two flavors:

- a fast variant, and
- a slow variant.

The slow variant’s name will be the original name appended with an unique suffix, e.g., originalName_slow. Hence, all functions ending in _slow belong to the set of slow functions.

**Unwind**

StackLifter creates a new basic block for each function call. The purpose is to allow us to use the basic block as a branch label. We call such a newly created basic block *function call basic block*. This is necessary when reconstructing the call stack after an unwind. The new basic block only contains an LLVM call instruction and an unconditional branch instruction. The branch jumps to the next instruction after the call as defined by the original application. Basic blocks get a unique name to identify them as “function call”
3 Speculation for Parallelizing Runtime Checks

basic blocks, e.g., \texttt{call\_X} where \( X \) is a running number. The basic block reached via the unconditional branch instructions carries the same name as the function call basic block plus a suffix, e.g., \texttt{call\_Xsucc}.

The basic block of a function call in the fast variant carries two additional instructions. Upon return from each function call, the global \texttt{doUnwind} flag is checked. If the check fails, control flow continues as in the original program. If the check succeeds we do a stack translation from the fast to the slow variant. Thus, execution continues with a \textit{register saving} basic block (discussed below). Listing 3.6 shows LLVM code of a transformed function call of the fast variant of a function.

Listing 3.6: Function call transformation for fast variant.

\begin{verbatim}
call_1: ; name to ID block as a function call site
   call void @foo()
   ; check if doing an unwind
   %doUnwind = load i8* @doUnwind
   %doUnwindCmp = icmp eq i8 %doUnwind, 0
   ; branch to register saving code or continue
   ; execution normally
   br i1 %doUnwindCmp, label %call_1succ,

   label %save_regs_1
\end{verbatim}

Indirect Function Calls

The StackLifter ensures that all slow functions only call slow functions. Once switched to the slow variant, we do not want to leave the set of slow functions. Direct function calls are changed by altering the name of the called function. The target of an indirect function call, though, can only be changed at runtime. Hence, StackLifter inserts additional code before each indirect function call (see Listing 3.7):

Listing 3.7: Indirect call transformation for slow variant.

\begin{verbatim}
call_3:
  %castedPtrToOrig = bitcast void ()* %orig to i8*
  ; @fp2sp translates fast function pointer
  ; to slow variant’s function pointer
  %ptrToClone = call i8* @fp2sp(i8* %castedPtrToOrig)
  %castedPtrToClone = bitcast i8* %ptrToClone to void ()*
  call void %castedPtrToClone() nounwind
\end{verbatim}

We call our runtime function \texttt{fp2sp} which, given a fast function’s address, returns the address of the corresponding slow function. Function \texttt{fp2sp} will use the input function pointer as an index into a map. This map is constructed before executing the \texttt{main} function by code generated by the StackLifter.
Saving Registers

During a stack translation each stack frame of a fast function must be replaced by an equivalent stack frame of its slow variant. Hence, StackLifter inserts register saving code into the fast functions. During a stack translation – when unwinding the call stack of the fast functions – for each function \( F \), the state of \( F_{\text{fast}} \) is stored onto the code base switching stack. This stack permits the reconstruction of the state in \( F \)'s slow counterpart \( F_{\text{slow}} \). The storing of the state is done in the register saving basic block.

For each function call within a fast function, there is a separate register saving basic block. In LLVM, the state of function \( F \) at instruction \( I \) is represented by the values of all live registers in \( F \) at \( I \). After each call instruction different registers might be live. Hence, each function call has its own register saving basic block. We perform liveness analysis for every basic block containing at least one function call. All registers marked live on entering the basic block need to be preserved.

An example of a register saving basic block is shown in Listing 3.8. External helper functions (\@pushI64 and \@pushFloat) are called to store all live registers (\%reg1 and \%reg2) on the code base switching stack. In the slow function all live registers are restored from the code base switching stack in reverse order.

Additionally, the label that uniquely identifies the location of function call basic block is pushed too (\@pushLabel). This label is of importance when the stack frame is reconstructed in the slow variant. The end of each register saving basic block is marked by a simple return instruction. Stack unwinding then continues in the caller.

Listing 3.8: Save register block.

```assembly
1  save_regs_1:
2     call void @pushI64(i64 %reg1)
3     call void @pushFloat(float %reg2)
4     call void @pushLabel(i32 1)
5     ret i32 undef
```

Restoring Registers

While a fast function needs to save the live registers, its slow function needs to restore the live registers with the help of a register restoring basic block.

As with register saving basic blocks, there is a restore basic block for each function call in a slow function. When entering a slow function, we need to check if this is a call because of a stack reconstruction. This is done by calling the external helper function \@popLabel. It returns zero, if no reconstruction is going on. Otherwise, the return value is a label identifying a specific function call basic block in the current slow function. A switch statement will divert control to the correct restore basic block depending on the return value of \@popLabel. Listing 3.9 shows an example of a slow function's entry basic block.

Listing 3.9: New entry block.
3 Speculation for Parallelizing Runtime Checks

```plaintext
new_entry:
  %next_label = call i32 @popLabel()
  switch i32 %next_label, label %old_entry [ 
    i32 1, label %restore_regs_1
    i32 2, label %restore_regs_2
  ]
```

If no reconstruction goes on (%next_label is zero) execution continues at original entry basic block %old_entry. Using zero as default if no code base switching is in progress eliminates the additional if (doUnwind) statement from Listing 3.5.

Listing 3.10 shows an example of a register restoring basic block.

Listing 3.10: Restore register block.

```plaintext
restore_regs_1:
  %reg2 = call i64 @popFloat()
  %reg1 = call i64 @popI64()
  br label %call_1
```

The restore basic block calls external helper functions (popI64 and popFloat) to retrieve the values of the live registers (compare with Listing 3.8). After all live registers have been restored, execution continues by branching to the function call basic block identified by the label popped at function entry. Restoring then continues with the function called in the function call basic block.

Restoring Static Single Assignment (SSA) Form

LLVM uses static single assignment (SSA) in its intermediate representation: “Each virtual register is written in exactly one instruction, and each use of a register is dominated by its definition.” All modifications need to preserve SSA. In the slow variant StackLifter introduces additional definitions for all registers within the register restoring blocks.

Listing 3.11: SSA-Example: before StackLifter.

```plaintext
%a1 = load i32 * %p1
call void @foo(i32 %a1)
```

Listing 3.11 shows a function call in the original code base before any instrumentations by the StackLifter. The argument value %a1 of the function call is defined by a load instruction. This code is in SSA, because the only virtual register %a1 is written by exactly one instruction (the load) and its use (the call instruction) is dominated by its definition (again the load).

Listing 3.12: SSA-Example: after StackLifter – in the slow variant.

```plaintext
new_entry:
  %next_label = call i32 @popLabel()
  switch i32 %next_label, label %old_entry [ 
    i32 1, label %restore_regs_1
  ]
```
3.4 Switching Code Bases

Listing 3.12 shows the code from Listing 3.11 after applying StackLifter’s transformations for the slow variant. This code violates the SSA constraint because virtual register %a1 is defined twice:

1. The original definition is in line 12.
2. The register restoring basic block for the call to foo contains the second definition in line 9.

Thus, the StackLifter’s transformations temporarily violate the SSA constraint. This sub-section describes how we restore the SSA form.

We use a modified version of the algorithm described in [31]. The basic idea is to insert special Φ nodes, that may break SSA in a defined way. In LLVM a Φ node is represented by a phi instruction. LLVM’s phi instructions do not need to be dominated by the definition of their used registers. However, virtual registers are still allowed to be written only by one instruction and the domination rule applies to all other uses of virtual registers.

Listing 3.13: Example of a phi node in LLVM.

Listing 3.13 shows an example of the phi instruction in LLVM. phi instructions are only allowed at the begin of a basic block (in this case Y). The arguments of a phi instruction is a non-empty set of pairs. The first element of each pair is a so-called incoming value and the second element is the incoming basic block. At runtime, if control branches from the incoming basic block X to basic block Y, then register %v_2 get assigned the incoming value %v_1. If control branches from Z to Y, then %v_2 get assigned 0. There has to be exactly one pair of incoming value and incoming basic block for each direct predecessor basic block in the control flow of the current function. More details about the phi instruction can be found in [65].

One assumption made in the algorithm for placing Φ nodes is that all variables are defined and initialized in the function’s entry basic block [31 pg. 25]. This, however, does not hold true for programs in LLVM IR. As far as variable-to-register mappings can
be reconstructed from LLVM IR code at all, variables can be initialized anywhere in the function.

Using the unmodified algorithm proposed in [31, pg. 25] can lead to wrongly placed $\Phi$ nodes. See Figure 3.5 for an example. Registers $v_1$ and $v_2$ refer to the same variable $v$. Variable $v$ is defined and initialized in basic block $X$, and only used in basic block $W$. Following the algorithm in [31], a $\Phi$-node for $v$ should be inserted in $Y$. First, $v$’s scope is limited to basic blocks $X$ and $W$. Hence, the $\Phi$-node in $Y$ has no uses. Second, because variable $v$ was defined and initialized in $X$ there is no incoming value for this $\Phi$-node for incoming basic block $Z$. If $v$ would have been defined and initialized in basic block entry $Y$, as assumed by [31], there would be an incoming value. However, as $v$’s scope is limited to basic blocks $X$ and $W$, the $\Phi$-node in $Y$ is neither possible nor needed.

Our solution is:

1. Apply the algorithm from [31].
2. Remove any illegal $\Phi$-nodes. We use a fix point algorithm and iteratively remove $\Phi$-nodes. If, during one iteration, we find no $\Phi$-nodes to remove, the algorithm terminates. We use three rules to detect an illegal $\Phi$-node. A $\Phi$-node is deleted if:
   a) It is never used.
   b) Its uses are exclusively incoming values for itself.
   c) The definition of at-least one of its incoming values does not dominate the basic block for which the incoming value is specified.

Rule a) is self-explanatory.

Rule b) is a special case of Rule a). A $\Phi$-node according to Rule b) has uses (but only itself). Therefore, it cannot be removed according to Rule a). But, as the $\Phi$-node is not used by any other instruction it is nevertheless not needed.

Consider the control flow graph in Figure 3.5 as an example for Rule c). As a default incoming value for each direct predecessor of $Y$, the original definition of $v_1$ is inserted, i.e.,

$$\%v_2 = \text{phi i32} [\%v_1, \text{label } \%W], [\%v_1, \text{label } \%Z])$$

When updating incoming values later on, the pair $[\%v_1, \text{label } \%Z]$ would remain unchanged. As explained above, this $\Phi$-node is illegal. Because of Rule c) we remove this $\Phi$-node. The reason is that the basic block $X$ (where the incoming value $\%v_1$ is defined), does not dominate basic block $Z$ (the basic block for which the incoming value is defined).
3.5 Speculative Variables

In ParExC runtime checks need to update and read checker specific book-keeping state. For instance, the out-of-bounds checker (OOB) adds for each executed `malloc` the address and size of the newly allocated buffer to its book-keeping state (Section 3.6.1). OOB verifies that each memory access reads from or writes to a currently allocated memory area.

Listing 3.14: Source code for Figure 3.6

```c
1 char* buf = malloc(10); // A
2 ...
3 buf[5] = ...; // B
```

Figure 3.6 shows an example for Listing 3.14. In Listing 3.14 and in the predictor (fast variant) the `malloc` (A) and the memory write (B) are in a causal order. However, the `malloc` and the `write` may happen in a different order in the executors (slow variant) as can be seen in Figure 3.6: the `write` access in the second executor happens before the `malloc` in the first executor. Therefore, the OOB checker cannot immediately verify that the `write` accesses an allocated region in the memory. Note that the predictor

3.4.4 Stack-local Variables

Addresses of stack-local variables can change during stack translation because additional local variables might be present in the slow function variant. Therefore, we put all addressable variables on a separate `alloca stack`. This stack is not changed by stack translation. Hence, the address of any variable on the `alloca stack` is the same for the fast and for the slow variant.

Technically, we replace all LLVM `alloca` instructions with our own implementation that allocates the variables on the `alloca stack`. On function entry we store the current frame address of the `alloca stack`. And on each function exit we restore the previously stored frame address.

Figure 3.6: The temporal order of A (malloc) and B (the write access) are different between predictor and executors. Checks use speculative variables to defer the check of the write.
also calls malloc to allocate the memory, but the predictor does not update any bookkeeping information about buffer bounds. Thus, the second executor can safely access the memory, but it does not know its bounds. Previous work [83, 97] streams the meta-data updates and lookups to a separate core, where all accesses are serialized. This serialization limits scalability and performance.

We propose to use speculation to decouple the accesses to shared state. Book-keeping state is stored in speculative variables. At runtime, each executor has a private view of the speculative variables. After finishing the checking of an epoch, an executor merges its private view into a shared view. Merges happen exclusively and in the order of the epochs, e.g., executor $E_i$ has to wait for executor $E_{i-1}$ to finish its merge before $E_i$ can merge its private view into the shared view.

### 3.5.1 Interface

Speculative variables have a generic interface to allow different checkers to use the same speculative variable implementation. The semantics of the values stored in speculative variables depend on the checker. For example, the OOB checker maintains for each buffer a speculative variable storing the bounds of the buffer. Speculative variables are addressed by ids. The OOB checker uses the memory address of the buffer as ID.

A speculative variable provides two operations:

- The read $(id, ob)$ operation returns the value of the speculative variable addressed by $id$. The read operation only operates on the private view. If no speculative variable for $id$ exists yet, the value $ob$ is returned. Additionally, a new speculative variable is created in the private view and its current value is set to $ob$. We call $ob$ an obligation. A checker must always provide an obligation to a read operation. The obligation is calculated by assuming the current check succeeds. Hence, the obligation is a speculative value. For instance, the OOB checker speculates that the checked memory access is valid. Therefore, the obligation is the memory area size at least required by the checked memory access. We explain in Section 3.6 how to calculate the obligations for our other checkers. The obligation is also stored within the speculative variable for later validation.

- The write operation $\text{write}(id, val)$ stores $val$ in the speculative variable addressed by $id$. Writing does neither create an obligation nor does it change or delete an existing obligation for $id$. The latter property is important: Consider that the memory access $B$ in Figure 3.6 would be followed by a realloc in the same epoch. At the realloc a write is used to update the size of the buffer. Although, we know the size of the buffer after the realloc, we still do not know the buffer’s size at the time of the memory access (before the realloc). But we need to check the validity the memory access. Thus, the obligation has to remain unchanged until it is verified using the shared view.

After the epoch $e_i$ is completely replayed, all its obligations need to be validated. Therefore, executor $E_i$ that checks epoch $e_i$ waits for $E_{i-1}$ to finish checking $e_{i-1}$. After $E_{i-1}$
finished its checking, it updates the shared view. Then $E_i$ validates all obligations of its private view against the view shared by all executors. The shared view does neither contain speculative values nor obligations. The exact semantics of the validation is provided by the checker via a callback function. The OOB checker validates that bounds stored in the obligation of a speculative variable are within the bounds stored in the shared view. A failed validation is treated as a failed check, i.e., the application is aborted. After the validation, $E_i$ updates the shared view with its current values of all speculative variables.

### 3.5.2 Deadlock Avoidance

Because of the use of speculative variables, we need to postpone the commit of external actions until all obligations are validated. If a checked application in the same epoch sends out a request to a server and then waits for a reply, it could block forever. The reason is that the request is held back until all obligations are validated at the end of the epoch. Hence, it is never made visible to the server and the reply will never be sent.

To overcome this problem, we implemented a second validation scheme. After executor $E_{i-1}$ successfully finished checking epoch $e_{i-1}$, $E_{i-1}$ validates the obligations of the private view of $E_i$. As soon as all obligations are validated and no new obligations were created during the validation, $E_i$ commits all outstanding external actions and also all new external actions immediately. Executor $E_i$ continues now unspeculatively. Our OS level speculation supports the unspeculative mode by allowing `parexc批准` to be called before replay has finished (Section [3.3.1]).

### 3.5.3 Storage Back-ends

To implement speculative variables, we have to map an ID to a speculative variable instance. We have two back-ends that implement this mapping:

- **Hash-Map** The variables are stored in a hash-map. This approach is optimal if there are not many speculative variables or if the ID space is sparsely populated. The out-of-bounds checker has one speculative variable per allocated buffer. As the start address of the buffer is used as ID, the hash-map back-end is best for our out-of-bounds checker.

- **Page Table** Speculative variables are grouped by by pages. We use a page table to map a speculative variable to a page. This approach is useful if a checker maintains many speculative variables that are near to each other in the ID space. For example, the Data-Flow Integrity Checker (DFI) has one speculative variable per accessed memory word, which is optimal for this back-end (see Section [3.6.2]).

The two back-ends give the checker developer the possibility to choose the optimal implementation for her checker.

### 3.6 Parallelized Checkers

To evaluate the ParExC framework, we implemented three checker plug-ins:
3 Speculation for Parallelizing Runtime Checks

- The Out-of-Bounds (OOB) checker instruments the slow variant with additional out-of-bounds checks for each memory access,
- The Data-flow Integrity (DFI) checker instruments the slow variant with additional data-flow integrity checks for each memory access [22], and
- The FastAssert checker removes all asserts from the fast variant.

We use the three plug-ins to evaluate the performance of our framework. We did not try to push the state-of-the-art of out-of-bounds or data-flow integrity checkers. Because the plug-ins are applied after the StackLifter (see Figure 3.2), the StackLifter’s instrumentation is visible to the plug-ins. In general, we found that these instrumentations are transparent to the plug-ins as the instrumented code is valid LLVM. It is even possible to change the restored state. In order to do so, a plug-in has to wrap the restoring helper functions, e.g., popI64 and popFloat.

3.6.1 Out-of-Bounds Checks

In our experience ParExC permits runtime checks to be added by a plug-in in almost the same way as one would add them to a sequential program. To justify this claim, we first show briefly how our simple OOB checker is implemented without ParExC and then how we adapted the OOB checker for ParExC to parallelize its runtime checks. Our goal is to detect out-of-bounds accesses to buffers allocated on the heap.

**OOB without ParExC**

The OOB plug-in is implemented as an LLVM pass. At compile time the OOB plug-in adds a runtime check for every memory access. To keep track of all currently allocated buffers at runtime, our instrumentation wraps all malloc and free calls of an application.

The OOB checks fail if and only if, the checked memory access goes to the heap but not into a currently allocated buffer. In LLVM, for most memory accesses the base pointer holding the base addresses (start of an allocated buffer) can be identified at compile time. In LLVM nearly all pointer arithmetic uses the getelementptr instruction that expects the base address as first operand. Our current prototype only instruments such memory accesses with checks. To facilitate the checks at runtime, we keep the size and base-addresses of allocated buffers in a hash-map at runtime. When using OOB with ParExC this hash-map is substituted by speculative variables. For every malloc call, a new entry is added to the map. Consequently, for every free call, the corresponding entry is removed from the map.

Memory access to the stack are not checked by our OOB checker. Therefore, at runtime all memory accesses to the stack are filtered out and ignored. When encountering a heap-access the corresponding buffer bounds are looked up and our checker tests if the memory access is within the bounds of the allocated buffer. If the lookup or the verification fails, the whole hash-map is searched for a buffer with matching bounds. For instance, the
lookup can fail, if the base pointer determined at compile time already points into the buffer. In this case the access might be valid, but base address in the base pointer is not the base address of the buffer. Therefore, the first check fails and the OOB checker has to search the whole hash-map. However, our experience suggests, that this happens rarely in practice. If no matching buffer is found, the application is aborted.

**OOB with ParExC**

The instrumentation for OOB with ParExC is the same as the instrumentation without ParExC except that only slow functions are instrumented. The fast variant does neither update the map of buffer sizes nor does it perform any checks. The major problem is that the slow variant at the start of epoch $e_{i+1}$ is a fork of the fast variant. Therefore, the slow variant does not know the sizes of all buffers allocated before the start of $e_{i+1}$.

Listing 3.15: Allocation and memory access in different epochs.

```c
char * buf = malloc(20);
prospect_chkpt();
buf[0] = 'h';
```

For example, in Listing 3.15 the allocation and the memory access happen in different epochs $e_i$ and $e_{i+1}$. Because the slow variants of $e_i$ and $e_{i+1}$ are executed in parallel to each other, the slow variant of $e_{i+1}$ might access variable `buf` and check its size, before $e_i$ allocates `buf` and stores the size of `buf` in the hash-map (see also Figure 3.6). We solve this problem by using speculative variables as introduced in Section 3.5.

### 3.6.2 Data Flow Integrity Checks

The motivation of the *data-flow integrity checker* (DFI) [22] is to protect against memory access errors. DFI checks that a read value was written from an allowed store instruction, i.e., variables are only allowed to be written at certain positions in an application. Our DFI checker conservatively extracts for each memory location $m$ a set of stores that are permitted to write to $m$. All loads and stores are instrumented. At runtime, each used memory location has a speculative variable that contains a unique writer ID. The writer ID identifies the last store instruction that has accessed this memory location. Hence, each executed store updates the corresponding writer ID. For each load we check that the writer ID of the last writer of the read location is within the set of allowed stores for this load. If we do not find a writer ID for the loaded memory location, the store was executed in a previous epoch. In that case, our obligation contains the set of allowed stores for this location. On obligation checking, we verify that the writer ID in the shared view is indeed an element of the set of allowed stores.

### 3.6.3 FastAssert Checker

Software developers are encouraged to add runtime assertions to their source code [73]. One of the trade-offs of runtime assertions is their runtime overhead. FastAssert (partly)
mitigates the negative effects of assertions on the application runtime. The plug-in removes any assertions and functions that neither change the internal nor the external application state from the fast variant. The slow variant still contains the assertions. Hence, assertions are still checked at runtime, but their computational overhead is parallelized.

For each function $f$, FastAssert computes if $f$ might change the internal or external state of the application or if $f$ transitively calls a function that might change the internal or external state of the application. Internal state changes are identified by store instructions. External state changes are identified by calls to external functions. If a function $f$ does neither, calls to $f$ are removed from the fast variant. The external function assert fails is a special case. It is used to implement the assert macro on our platform. Therefore, assert fails is considered as not changing any state.

By removing not only assertions but also side-effect free computations, we also remove user defined sanity checking code. We expect that FastAssert not only mitigates the perceived performance overhead of existing assertions, but also motivates to the inclusion of more (computationally expensive) assertions.

FastAssert has no additional book-keeping state. Thus, it does not use speculative variables.

### 3.6.4 Runtime Checking in STM-Based Applications

In our evaluation in Section 3.7.2 we compare ParExC with checking an application parallelized with the help of Software Transactional Memory (STM). For the comparison we used the OOB checker from Section 3.6.1. This subsection provides details on how we implemented the OOB checker using an STM. We chose to run the experiments on STAMP [21] a benchmark from the STM community. This selection restricts the checkers we can apply. The STAMP benchmarks contain not enough assertions for the FastAssert checker. The DFI checker cannot be trivially parallelized with STM because of the non-deterministic reader/writer interleavings. The STM’s inherent non-determinism makes it difficult to learn all possible reader/writer interleavings for DFI. Hence, we applied only our OOB checker to the STM parallelized applications.

STM provides transactions at the programming language level. A transaction is a set of statements that are executed atomically. An STM detects read/write and write/write conflicts of transactions and aborts or delays conflicting transactions. On abort all state changes are rolled back and the transaction is then retried. STM requires all shared data accesses within a transaction to be instrumented. This instrumentation incurs synchronization and book-keeping overhead. However, this book-keeping overhead can often be amortized by achieving better scalability through speculation.

In our experiments we used a C++ version of TinySTM [40]. Instead of instrumenting the code by hand, we used Tanger [39] which is an extension for LLVM [65] that automatically transactifies applications. The programmer only has to mark the start and the end of transactions. The instrumentation redirects all shared data accesses within transactions to an STM. Although an experienced programmer might exploit optimizations by manually transactifying the application, we do expect that many applications will not be manually transactified.
3.7 Evaluation

To parallelize an application with OOB checks, we perform two steps:

- First, we apply the same compiler transformations as the ParExC OOB plug-ins.
- Second, we apply the Tanger transformations on the code resulting from the first step.

Due to this ordering Tanger automatically puts the book-keeping state of the runtime checks under the control of TinySTM. Therefore, we do not need speculative variables in this approach.

To parallelize an application together with runtime checks an application developer has to parallelize the application itself. Parallelizing an application with STM (and with or without runtime checks) involves two major tasks:

- Insert code to spawn threads.
- Add critical sections and encapsulate these in transactions.

Often an application rewrite is necessary to optimally exploit parallelism. Thus, we believe that an STM is more difficult to use for the application developer who wants to distribute the load of runtime checks over many cores than the ParExC approach. Furthermore, while an application itself must be parallelizable to parallelize its runtime-checks using STM, this restriction is not the case for ParExC. ParExC can be applied to applications that are difficult to parallelize or even not parallelizable at all. However, because ParExC’s scalability is limited by the overhead of the runtime checks an application parallelized with STM may run faster if it scales very well and runs on sufficiently many cores.

3.7 Evaluation

Our evaluation focuses on the scalability of our checkers, the comparison between Tanger/-TinySTM and ParExC, as well as ParExC’s overhead. We performed all measurements using Fedora 10 on a 2xIntel Xeon E5430 with 2.66 GHz (8 cores) and 16 GB RAM. Like [83], we used the real time reported by the time command to measure the runtime.

3.7.1 Performance

We evaluated ParExC’s performance with the three checkers introduced in Section 3.6:

- OOB: The out-of-bounds checker.
- DFI: The data-flow integrity checker.
- FastAssert

Since we used existing benchmarks, we had to adapt these benchmarks manually. However, we only needed to add calls to parexc_chkpt. For none of the benchmarks we needed to add more than 5 lines of code. Every presented value is the average of 3 runs.
3 Speculation for Parallelizing Runtime Checks

Out-of-Bounds Checker (OOB)

To measure the speedup of parallelizing the bounds checks with ParExC, we used five different benchmarks from different application domains. The first set of benchmarks are Genome and Vacation that are part of the STAMP [21] benchmark suite for transactional memory. The performance of both benchmarks is CPU and memory-bound. All STAMP benchmarks can be executed in parallel using multiple threads (see Section 3.7.2). However, we run them single-threaded since we want to show the parallelization with ParExC. The second set of benchmarks are Whetstone and LinPack. Both come from the high performance community and are, at least in our measurements, CPU-bound only. The last benchmark is bzip2, a real application and not an artificial benchmark.

Figure 3.7 shows the runtime (in s) of the five benchmarks. We run all benchmarks in four configurations:

1. **without OOB**: without ParExC and without out-of-bounds checks (OOB) to show the lower bound for the runtime,

2. **ParExC without OOB**: without OOB but with ParExC to show the framework’s overhead,

3. **OOB with ParExC**: with OOB and with ParExC to show the runtime reduction by ParExC’s parallelization, and

4. **OOB without ParExC**: with OOB but without ParExC to show the slowdown of the OOB without parallelization.

All runs without ParExC are single-threaded, whereas with ParExC we make full use of all 8 cores. The ParExC overhead (configuration ParExC without OOB relative to without OOB) is most visible for Whetstone (3.1x) and LinPack (3.0x).
In Figure 3.8, we plotted the speedup of OOB with ParExC relative to OOB without ParExC. To estimate the maximal possible speedup on our 8 core machine, we use the following upper bound:

\[
\text{slowdown} = \frac{\text{runtime OOB without ParExC}}{\text{runtime without OOB}}
\]

\[
\text{upper bound} = \frac{\text{number of cores}}{1 + \frac{1}{\text{slowdown}}}
\]

The upper bound for the speedup assumes that we can distribute load of the executors perfectly over all available cores. Additionally, the upper bound takes into account that the fast variant needs also a share of about \(\frac{1}{\text{slowdown}}\) of the CPU.

Figure 3.8 shows that we do not reach our estimated upper bound. Hence, there is room for improvements. However, Figure 3.8 also shows that whenever the speedup drops it corresponds to a drop in the upper bound too. That indicates that the ParExC framework itself might introduce some constant overhead. In Section 3.7.3, we analyse ParExC’s framework overhead in more detail to explain the gap between ParExC’s speedup and the upper bound.

**Data-flow Integrity Checker**

To evaluate the scalability of our DFI checker we limit the number of parallel executors to simulate CPUs with fewer cores. If the maximum number of parallel executors is reached, the predictor blocks before spawning a new executor until at least one of the currently checking executors finishes. In our experiments we measured either throughput or runtime.

Figure 3.9 shows the runtime for parallel and sequential DFI checking of the Vacation and Labyrinth STAMP benchmarks. For Vacation we plot transactions per second and for Labyrinth the total runtime. Vacation performs 363,000 transactions per second.
without runtime checks. Thus, according to the measurements in Figure 3.9, DFI has a slowdown of 7.26x in the sequential execution, i.e., without ParExC, and 1.91x with ParExC. Labyrinth runs for 56.21s without runtime checks on a 512x512 maze. The DFI slowdown without ParExC is 7.14x and 1.37x with ParExC.

**FastAssert Checker**

We tested FastAssert with real world code. We choose the words unit-test of BOOST’s multi-map implementation [74]. This unit-test contains a set of very expensive (in terms of runtime) sanity checks. Figure 3.10 (left) shows the runtime of the test in three configurations:

1. **without assertions**: without assertions and without ParExC,
2. **assertions without ParExC**: with assertions but without ParExC, and
3. **FastAssert**: with assertions and with ParExC.
3.7 Evaluation

Figure 3.11: Throughput/runtime of the OOB checker with ParExC and Tanger/TinySTM. The error bars show the minimum and the maximum of our measurements.

Configuration without assertions is 458x faster than configuration assertions without ParExC. This is an unusual runtime overhead for sanity checks and assertions. FastAssert is 7.2x faster than configuration assertions without ParExC (Figure 3.10 right hand side). But the runtime overhead compared to configuration without assertions is still impractical 64x. Nevertheless, we believe that given more cores FastAssert would reduce the runtime of configuration FastAssert even more. We believe this example shows that FastAssert enables the inclusion of heavy-weight user defined sanity checks into production code.

3.7.2 Checking Already Parallelized Applications

The STAMP benchmarks are explicitly written for benchmarking STM libraries. Therefore, all shared memory accesses have been marked explicitly by experts. However, as we focus on as few manual changes as possible, Tanger ignores these markings (except the transaction boundaries) and puts all potentially shared memory accesses under the control of TinySTM. We expect different results if TinySTM would be directly called by the developer, i.e., without Tanger’s automatic transactification. But this would additionally involve the manual transactification of the OOB checker. We want to avoid any additionally manual and (possibly) error-prone overhead. Every measured value represents the average of 5 runs.

We measured the percentage of application runtime that is parallelized. In particular, the startup and the cleanup phase of both benchmarks are not parallelized. In Vacation, 99.84% of the single threaded runtime is parallelized. In contrast, Genome’s startup phase takes very long, i.e., only 85.7% of Genome’s single threaded runtime is parallelized. While ParExC can parallelize the checks even if the code is not parallelized, Tanger/TinySTM can only parallelize the checks in parallelized code.

Figure 3.11 compares the OOB with ParExC and OOB with Tanger/TinySTM. In general, OOB with ParExC has better throughput (for Vacation) and shorter runtime (for
3 Speculation for Parallelizing Runtime Checks

Figure 3.12: Overhead of the system call speculation in the fast variant and the deterministic replay in the slow variant for Vacation.

Genome). This indicates that the instrumentation of OOB with ParExC is more lightweight than the instrumentation of OOB combined with STM (Tanger/TinySTM). In contrast to applications instrumented with ParExC, those instrumented with Tanger/TinySTM need to acquire and check locks for every memory access within original transactions of the application and in the OOB runtime library.

Furthermore, we can see in Fig. 3.11 that both benchmarks scale better when instrumented with ParExC than if instrumented with Tanger. Vacation instrumented by Tanger actually does not scale at all if more threads are used than cores are available. One reason is increased contention, which leads to higher abort rates. Another reason is the lack of transaction scheduling by TinySTM. Both issues do not arise if the ParExC approach is used. First, there is no contention between executors. Second, executors are scheduled by the OS transparently. However, more measurements are necessary to clarify this issue.

Since only the checks are parallelized, the application parallelized using the ParExC framework cannot be faster than the original application. Therefore, if the workload is easily parallelizable, and enough cores are available, the Tanger approach will eventually result in better scaling applications. On the other hand, the ParExC approach also works with applications or parts of applications that are difficult to parallelize, as long as heavy runtime checks have to be applied.

3.7.3 ParExC Overhead

To analyze the runtime overhead introduced by ParExC, we measured the overhead of system call speculation, deterministic replay, and the StackLifter individually. Figure 3.12 shows the overhead of system call speculation and deterministic replay for the Vacation benchmark with four different workloads identified by the number of performed transactions. System call speculation and deterministic replay are implemented by our Linux kernel module. We run Vacation only with the kernel module. The slow variant was forked right before the execution of the main function. Slow and fast variant share the same code. To avoid measuring overhead of the StackLifter, the whole execution took place within one epoch. No further instrumentation (especially no StackLifter) was applied. The overhead of the smaller workloads is dominated by the start-up time of our
3.7 Evaluation

Figure 3.13: Overhead of the StackLifter instrumentation with three optimizations for Vacation.

Figure 3.13 shows overheads introduced by the StackLifter. Again, we executed Vacation with the four different workloads. The StackLifter adds instrumentation to both, the fast variant and the slow variant. In this experiment we measured the overheads introduced by this instrumentation but not the stack lifting process itself. The main sources of overhead are the check of the doUnwind flag in the fast variant. The slow variant has two sources of overhead:

- The new entry block in every instrumented function.
- The look-up of function pointers at indirect function calls.

We ran both variants separately without system call speculation and deterministic replay. Applying the StackLifter to all code of the fast variant increases the runtime up to 1.8x compared to vacation without ParExC. If the StackLifter is restricted to functions on the path to parexc_chkpnt, the overhead is below 3%. The instrumentation of the slow variant introduces a higher overhead. Instrumenting all functions, the overhead is between 2.4x and 4.6x. If only functions on the path to parexc_chkpnt are instrumented the overhead introduced by the StackLifter is between 1.8x and 3.0x.

The overhead of the slow variant can be further reduced by optimizing indirect function calls. Note that all function pointers (also in the slow variant) point to functions of the fast variant. Hence, before an indirect call we need to look up the function pointer of the slow variant. Therefore, we use a map. This map is indexed by a function pointer to a fast variant function (see Listing 3.7). The look-up can be optimized by caching the result of the last looked-up element in the look-up function fp2cp. This optimization is not used in the fast variant. Therefore, it does not influence the overhead of the fast variant. The slow variant’s overhead is reduced to 1.16x for the largest tested workload.
3 Speculation for Parallelizing Runtime Checks

![Figure 3.14: Time to perform a stack lifting for different stack depths.](image)

In our experience the time needed to switch from the fast variant to the slow variant is very small. Hence, it is difficult to get reliable measurements from our benchmarks. Therefore, we built a micro-benchmark to measure the time to switch code bases. The micro-benchmarks consists of a recursive function that calls `parexc_chkpt` exactly once at the deepest nesting level. Apart from StackLifter we did not apply any other instrumentation. Figure 3.14 shows the time needed to switch from the fast variant to the slow variant for growing stack depths. Each stack frame contains one label and three live integer registers. Unsurprisingly, it takes longer to un- and rewind from greater stack depths. A linear relation exists. This indicates predictable behavior.

The runtime overhead in the slow variant is noticeable for large stack depths. However, this runtime overhead is already parallelized by ParExC. Ideally, this runtime overhead could be made negligible by adding sufficiently number of cores.

3.8 Conclusion

ParExC makes heavy use of speculation. Both, the fast variant, and the slow variant are executed speculatively. While this is easy to see for the fast variant, it is not so obvious for the slow variant. The slow variant’s execution is split into epochs. At each start of an epoch the slow variant takes over the speculative state of the fast variant. Only at the end of the epoch the speculative starting state is compared to the end state of the previous epoch. Therefore, the slow variant is executed speculatively, too.

Our evaluation in Section 3.7.1 demonstrates that ParExC’s speculation can reduce the perceived performance overhead of checking (Thesis 1). Our performance evaluation is also an argument for Thesis 4 which says that parallelization of runtime checks can speedup checking. The comparison of ParExC with checking STM-based applications in Section 3.7.2 shows that ParExC’s approach can be faster than parallelizing a checked application with STM (Thesis 5).

ParExC’s parallelization of heavyweight state corruption sensors (checkers) is clearly a benefit for approaches like Rx [94] and ASSURE [108]. It allows these approaches to utilize more heavyweight sensors with low additional performance costs. Furthermore, ParExC can provide more support to these approaches. Both approaches also make use of check-points and replay techniques, similar to the techniques used in ParExC. Therefore,
an appropriately adapted ParExC that exports internal interface for check-pointing and replay can be used as infrastructure to build system such as Rx and ASSURE on top of it.

To use ParExC, application developers have to compile their applications with LLVM and the ParExC compiler and they have to mark potential epoch boundaries in their source code (using calls to `parexc_chkpt`). The only difference for the users is, that they have to start the applications under ParExC’s runtime environment. However, this startup setup can be automated to hide ParExC from the user. Thus, ParExC can be completely transparent to the user apart from the performance benefit.
4 Automatically Finding and Patching Bad Error Handling

The previous two chapters discussed approaches needed for error tolerance. Error tolerance approaches are the first sub-class of automatic hardening approaches (see Figure 1.1). The second (and last) sub-class of automatic hardening approaches is bug removal. Bug removal covers approaches that deal with errors in already deployed software. While error tolerance tries to detect and mask errors at runtime, bug removal removes bugs after deployment but before using the software in a production environment. By removing bugs the likelihood of errors at runtime is reduced. If is, of course, possible to apply bug removal before integrating a software into a production environment and later at runtime error tolerance.

Individual bug removal approaches focus on certain bug patterns. This chapter focuses on bugs in error handling code. The next chapter discusses the removal of bugs from 3rd-party components.

Many service outages are caused by buggy error handling code [29]: the error handling is often the least tested, least documented, least executed and least understood part of a software component. Under high load more resources (e.g., more memory and file descriptors) are needed and errors will occur and will need to be handled correctly when resources become depleted. Bad handling of resource depletion errors are therefore more likely to become visible at times when the system is needed the most, i.e., when the load is high.

Because of economical reasons, most dependable systems cannot be built from scratch. Instead one has to reuse existing software components. Some of these components might, however, not been built to the standards required for dependable systems. Developers can use different static and dynamic analysis techniques that aid them evaluating and improving the dependability of software components.

Error handling code can be a large part of a code base ([29] reports up to two thirds) and it can be the cause of a large percentage of outages ([29] also reports up to two thirds even though error handling code is rarely executed). Therefore, we focus in this chapter on the problem of how to evaluate and improve error handling code. This chapter describes the AutoPatch project[^1] that aims to increase the dependability of software-based systems by:

- using static and dynamic program analysis to evaluate source and binary code, and
- generating hardening patches to remove certain robustness issues found in the analysis phase.

[^1]: This chapter is based on [117] [118] [116].
4 Automatically Finding and Patching Bad Error Handling

The generated patches can be used in three different ways:

- User can relay on generated patches as temporary solution until the software vendor makes patches available. This is, for example, important for vulnerability bugs that can be used to gain malicious control over a system. Generated patches are immediately available and can protect the system as soon as they are generated. Nevertheless, patches released by the software vendor usually undergo more testing. Hence, in the long term users are more likely to apply patches released by the vendor.

- Generated patches can be used to improve the efficiency of application developers by helping them to correct their code with less effort [131]. Application developers can uses a generated patch as a template for their bug fix.

- For less critical components generated patches might be used instead of patches supplied by a vendor. This is especially useful if a vendor is not willing or able to provide patches.

The approach of this work is to use error injection techniques to discover bugs in the error handling of programs (Thesis 7: “Fault injection finds failures even in mature software.”). We present some of the bugs we found using our error injection in Section 4.7.2. Our patch generation is based on the following observation: Even though error handling is the buggiest part of the code, nevertheless most programs handle most of the errors correctly. We use this observation by trying to map errors that a program cannot handle to errors that the program can handle (Thesis 8: “Automatic error mapping can mask failures.”). To do so, we define patch patterns that can be applied in well specified situations. A patch pattern is code template that can be instanciated to remove certain bugs. Whether a pattern can be applied in a given situation is verified by using static analysis of the binary code.

Our approach can be applied at the end-user site without any support from the application developer. Because our static analysis works on binary code, we do not require source code of the applications to harden. Our evaluation in Section 4.7 shows that our automatically generated hardening patches remove 84% of previously found crash failures (Thesis 6: “Automatic bug removal based on patch patterns can decrease the number of failures.”).

4.1 Related Work

The work in this chapter is related to other research in the area of robustness analysis and patch generation. The Ballista project [62, 63] provides a toolkit to automatically determine the robustness of POSIX functions using error injection. The functions are called with extreme values as arguments to determine their robustness. Ballista differs in two major aspects from our work:

- The error injection is done into POSIX functions using extreme values while we inject errors into applications.
• Ballista’s results are used for measurements only while we use them for patch generation.

The HEALERS project \cite{45,44,119} uses a Ballista style error injection approach to determine safe and unsafe data types for function arguments. Values which belong to unsafe data types lead to non-robust behavior (e.g., crashes) or insecure behavior (e.g., buffer overflows). A data type can be a value range, e.g., positive integers, or a certain value, e.g., a NULL pointer. More abstract data types, for example “buffer that is allocated on the heap”, are also possible. The knowledge about the safety of function arguments is used to generate various types of wrappers for shared libraries. For example, a robustness wrapper prevents a library function from being invoked with unsafe argument values. Similar to the wrappers generated by HEALERS, our patches intercept function calls. However, in this work we protect the application and not the library as it was done in HEALERS. Both approaches are complementary: HEALERS makes sure that libraries return errors instead of exhibiting non-robust or insecure behavior and our approach – AutoPatch – makes sure that programs can cope with errors returned by libraries.

We reuse some of the techniques that HEALERS uses to determine the type of shared library functions: We parse the Unix \texttt{man} pages and C header files of the given library to extract function signatures. HEALERS uses these signatures for selecting the extreme values to test a function. We, instead, use the signatures for static analysis and for patch generation.

The FIG project \cite{17} aims at determining the robustness of applications in the presence of errors returned by system functions. Therefore it uses \textit{error injection} from the interface of the Standard C library into the application. FIG generates wrappers to intercept calls to functions of the Standard C library and to inject errors into the callers. Depending on the given configuration, either an error value is returned or the wrapped function is executed. The authors of \cite{17} have found several robustness issues in applications. Like Ballista, the error injection is done to evaluate the robustness only. We use the approach of FIG to inject errors. With the error injection we identify calls to library functions for which an application does not perform proper error handling. In contrast to FIG, we use the error injection results not only for evaluation purposes but also to determine which function calls need to be patched.

LFI \cite{70} improves over FIG in several ways. While FIG can only inject faults for calls to functions of the libc, LFI is a general fault-injector for arbitrary libraries. Therefore, LFI uses static analysis of the libraries, an application is linked to, to determine fault-injection profiles. For each exported function of a library LFI extracts constant return values and other error information transported via side-effects from the library’s binary code. These constant return values and error information form the fault profile of a library. The fault profile is used by LFI to control the fault injection. In contrast to AutoPatch that gathers the fault profile from the documentation of a library, LFI does not rely on documentation like \texttt{man} pages and header files to build the fault profile. That makes LFI usable even with library that have not any documentation. While documentation might be incorrect, LFI’s static analysis is also sometimes incomplete. For instance, LFI cannot handle the rare cases when a function contains indirect jumps. Which approach is preferable depends on the requirements. However, LFI, like FIG, is a fault injection tool only. Whereas,
AutoPatch’s patch generator needs anyhow expert knowledge (such as documentation) to decide if a certain patch pattern can be used for a given function. For instance, error handling bugs for calls to `write` cannot be patched by with the preallocation pattern (see Section 4.6.2). However, AutoPatch’s fault injector can be substituted by LFI.

The authors of [107] automatically generate security patches for applications. Their approach tries to patch vulnerabilities abused by exploits. The approach needs access to the application’s source code. The application’s source code is compiled with a buffer overflow detector. The buffer overflow detector is similar to our out-of-bounds checker in Chapter 3.6.1. The application is executed and the exploit is performed on it. The buffer overflow detector records any buffer overflows. Based on the recorded data the applications source code is patched to prevent all detected buffer overflows. The authors introduce a common patch pattern to alter the application’s source code. Although the patches are also automatically generated our work differs in its focus and requirements. We focus on robustness issues. As we use error injection, we do not need any knowledge about robustness flaws of the inspected application. Our requirements are much lower. We only require the application’s binary, but not the application’s source code. Our patches are wrappers located between an application and the libraries the application uses. Using wrappers we do not need to alter the application’s binary or source code.

ASSURE [108] uses error mapping similar to ours. If an error is detected ASSURE rolls back to a previously determined rescue point. Rescue points are points in the application execution known to have safe error handling. After the rollback ASSURE triggers this safe error handling by injecting an appropriate error. In contrast to ASSURE our approach does not rely on rollback. Rollback is difficult to implement if external actions are involved, e.g., the deletion of a file or message send via network.

Other approaches like Exterminator [84] and First-Aid [50] also deal with memory errors. Exterminator uses a randomized heap to probabilistically detect memory errors. First-Aid relies address space randomization and on the crash detection of the OS to detect memory errors. The assumption of First-Aid is that thanks to address space randomization any memory error leads to a crash. Both approaches patch memory errors by over allocating. Over allocation means that more memory than needed is allocated, but enough so that the exploit cannot overflow a buffer any more. However, these approaches do not deal with error handling bugs of code calling the memory API (for instance calls to `malloc`).

### 4.2 Overview

Systems-oriented libraries like the standard C library return error codes when a failure occurs during the execution of a function. For example, if all file descriptors are currently in use, an error code is returned by functions that open a file. Dependable applications need to handle such error codes appropriately. Error handling code is, however, often buggy. Such bad error handling can be a major cause for service unavailability.

In this work we focus on bad handling of errors returned by systems-oriented libraries. Figure 4.1 shows the caller/callee relationship. The caller calls the callee (step 1). The
4.2 Overview

Figure 4.1: Caller/Callee relationship together with our patch.

callee processes the caller’s request (step 2). Last, the callee returns to the caller. Now, the caller might need to handle errors of the callee (step 3). Even though, we most often call the caller application and the callee library, our approach works for any kind of caller and callee. However, the implementation may be different for other kinds of caller/callee relationship.

We use systematic error injection to identify function calls that do not handle errors properly. We understand that under proper handling of errors an application does not crash if we inject an error, i.e., return an error code instead of calling the original callee. In this work we do not address other possible consequences of bad error handling like an incorrect output or application hangs.

To perform error injection into applications on the library level, we intercept calls to library functions and return in some cases an error code instead of executing the function that was called. We do this in a systematic way to identify unsafe function calls. Function calls are unsafe if the application crashes when an error value is returned. These are the function calls that must be patched. We restrict our analysis to calls to library functions, which we want to patch. We use the term function of interest to refer to such library functions.

Patching of the application is based on patch patterns. In this chapter we present two patch patterns:

Preallocation The basic idea of the preallocation pattern is that one maintains some reserved resources for calls that allocate resources and for which the application cannot cope with errors.

Whenever a resource allocation fails and the application cannot cope with an error for this allocation some of the reserved resources are returned. To counter depletion of reserved resources they are refreshed in a fault tolerant way.

Error Value Mapping The error value mapping pattern transforms an error of a function call that the application cannot deal with, into an error that is returned at some other function call for which the application deals with errors correctly.

Both patch patterns cannot guarantee 100% robustness for the patched bugs. For instance, if the reserved resources are depleted and refresh is not able to allocate new reserved resources, then preallocation cannot help anymore. However, our evaluation shows that the patches reduce the number of crashes significantly (see Section 4.7.1).
4 Automatically Finding and Patching Bad Error Handling

Figure 4.2: The data flow of analyzing an application for patch generation.

Figure 4.1 shows the position of the patch. The patch intercepts the call of the application (caller) to the library (callee). The patch removes error handling bugs (in step 3) by either masking an error (with the help of preallocation) or by triggering the error handling of known safe function calls (error value mapping).

Additional information about calls to library functions is needed in order to apply our patch patterns. The preallocation pattern needs to know the argument values of function calls. The values are used as parameters for preallocation, which must be done before the unsafe function call. We use a dynamic and a static approach to obtain these argument values:

- We record dynamically all arguments for function calls of one or more runs of the application.
- We do a static analysis of the application’s binary to extract further information that helps us to generate the patches.

Figure 4.2 shows the data flow of our analysis. The argument recorder and the error injector are dynamic analysis techniques – they execute the application and observe it. The user must provide a run configuration (work directory, command line arguments, etc.) for repeatable execution. Only bugs found by the error injection can be patched. Thus – like for any dynamic analysis approach – the run configurations should provide a good coverage of the application’s execution. The static analysis tool works on the application’s binary. Hence, our approach does not depend on the application’s source code. If the binary contains debugging information, our system can also extract various information that will aid a developer in fixing the bugs the system finds.

All data gathered by our analysis tools is stored within a database. There are two reasons for using a database:

- Some of the data gathered is expensive to acquire and one wants to keep it persistent.
4.3 Learning Library-Level Error Return Values from System Call Error Injection

For instance, the error injection has a quadratic overhead. The error injection tool has to perform one separate application run for each function call the application performs. Hence, if \( N \) is the number of function calls in the application, then \( N^2 \) functions calls are performed by the application under the control of the error injection tool.

- The error injector and our static analysis need some of the data gathered during argument recording. Thus, it is very convenient to have the data accessible via the database.

Please note that our approach does not rely on already found bugs. The only input our approach needs are the application’s binary and one or more run configurations of the application. For example, existing unit tests of the application can be used as run configurations. The error injector needs the names of called functions and the return values that signal an error for those functions. We call these values *error values*. Error values can be obtained in three ways:

- An expert defines them.

- Automatic tools can extract them statically from documentation, specification, source or binary code.

- They can be automatically learned by injecting errors into system calls (Section 4.3).

The signatures of functions used within the various generated wrappers are derived with the approach of [15]: we parse header files and man pages to obtain those signatures.

In the following we describe how error values can be extracted using dynamic analysis (Section 4.3). Then we describe our analysis approaches (Section 4.4). In Section 4.5 we propose a novel approach for systematic error injection (such as we use in Section 4.4.2). Section 4.6 describes our patch generation based on the results of the analysis described in Section 4.4.

4.3 Learning Library-Level Error Return Values from System Call Error Injection

For our approach we need the *error values* of the functions of interest. We propose a novel technique to learn error values of library functions by injecting errors at the *system call* level. Therefore, we intercept system calls issued by libraries and return an error instead of executing the system call. When the library passes the control back to the application, we record the error return value. This approach has several advantages:

- The system call API is small compared to the number of functions implemented in all libraries installed on a common system.
4 Automatically Finding and Patching Bad Error Handling

- The system call API is well defined and sufficiently stable. Hence, a small hand-written error specification of the system call API is sufficient to derive a sound error specification for library functions that use system calls directly or indirectly. At the end of this section we show how to even avoid the hand-written error specification by mining the error specification of the Linux system call API automatically.

- The learned error specification of a library can be reused for injecting errors in other applications that use this library. Hence, the learning done with one application benefits the error injection for many applications.

To derive an error specification of a library, we execute applications that use this library under the control of our system call error injector. All errors returned by the library are recorded. We record the function’s return value and value of the \texttt{errno} variable. The recorded error values form a sound but possibly incomplete error specification. We use this error specification later on to inject errors into an application when performing library calls. This is depicted in Figure 4.3: to get the error specification needed for directly injecting errors from a library into an application (possibility (1)), we inject errors into the library at the system call level (possibility (2)).

Please note that always injecting errors at system call level only has two disadvantages compared to our approach:

- It would be more difficult to infer where the actual bug is located. Injecting errors at the system call level will expose software bugs at the library and application level while our approach will in general only trigger application bugs.

- The error specification of a library can depend on the library’s version and on the hardware platform the library runs on. Hence, if we would inject errors only on system call level the application is only tested against the error specification of the library currently in use.
4.3 Learning Library-Level Error Return Values from System Call Error Injection

Then again, with our approach we can learn different error specifications for different library versions and hardware platforms. These error specifications can be merged to a more useful error specification. For example, with this merged error specification application bugs can be found that are only triggered on seldom used hardware platforms.

4.3.1 Components

Our implementation consists of two components:

**Recorder Wrapper**  The recorder wrapper intercepts function calls from the application into the library. It also records the return values and the `errno` values returned by library functions.

**Error Injector**  The error injector intercepts all system calls. Only if the system call was issued by the library, an error is injected. To know if or if not to inject an error the error injector relies on the recorder wrapper, because the recorder wrapper keeps track of the invoked library functions.

The recorder wrapper and the error injector can be implemented as one monolithic wrapper.

Our approach can be extended to other programming languages and error handling paradigms. For example, for Java the recorder wrapper would intercept method invocations to the methods of certain (library) classes. The recorder wrapper would inform the system call error injector that a library function is going to be executed and then invokes the wrapped method. The error injector would inject errors into system calls, as long as the execution is within the wrapped method (signaled by the recorder wrapper). All exceptions thrown by the wrapped method are caught and recorded by the recorder wrapper. These exceptions form the error specification of the invoked method.

Note the learned error specification differs from the exception specification that is part of the method’s signature in Java. First, the exception specification does not contain runtime exceptions (such as `OutOfMemoryError`). But the error specification also contains all recorded runtime exception. Second, some exceptions from the specification might not be used in practice and, thus, will not be part of the learned error specification.

Two issues are left to be discussed. First, efficiency of the error injection and second, we show how the error specification of an operating system API can be obtained.

4.3.2 Efficient Error Injection

Just before injecting an error, we take a check-point of the application’s and the library’s state by forking a new child process. The error injection is done in the child process. The child process is only executed until the error signals of the library function have been recorded by the wrapper. The parent process waits until the child process has finished and continues with normal execution until the next system call is encountered. This approach is similar to that of [31].
Without using fork the application would have to be executed once for each error that we want to inject. If \( N \) is the number of system calls performed by library functions during a certain run of an application, then our analysis would need to execute \( O(N^2) \) system calls. With the fork approach the number of system calls reduces to \( O(N) \).

However, the forking of a child process introduces a new issue: The child process might corrupt the external state (like files) so that the parent process cannot continue. We discuss this problem and a possible solution to it in Section 4.5 in more detail.

### 4.3.3 Obtain OS Error Specification

For the error injection at system call level we need an error specification of the system call API. While this API is significantly smaller than the number of library functions, one might still want to create this error specification automatically. Therefore, we present an approach that infers this error specification for Linux using the Kernel’s header files and the system call documentation.

1. We parse the header file of the Linux kernel that defines all system call numbers (unistd.h). This results in a mapping from numerical system call IDs to textual system call IDs.

2. We parse the header file errno.h where all error codes of all system calls are defined. Again, the result is a mapping from numerical error codes to textual error names.

3. We parse the man pages for each textual system call ID for all textual error names from the errno-mapping. We assume that all error names mentioned in the documentation of a system call can be returned by it.

Thus, we automatically obtain at least a subset of all errors that a system call can return. The result of the last step forms the OS error specification.

### 4.4 Finding Bad Error Handling

This section introduces the three components of the failure analysis shown in Figure 4.2:

- Argument recording to collect argument values later used to generate patches that, for example, do preallocation.

- Systematic error injection from the library into the application for finding bad error handling.

- Static analysis of the binary code to extract constant arguments and to look for patterns used to apply error mapping.

The first two approaches are dynamic analysis techniques, i.e., they need a run configuration, for instance, unit tests or a testing environment. The third approach only uses the binary code of the application to patch without executing it.
4.4 Finding Bad Error Handling

4.4.1 Argument Recording

The objective of the argument recorder is to learn common arguments for specific invocations of functions of interest. These common argument values are, for example, useful for the preallocation patch pattern. The preallocation pattern keeps some preallocated resources as backup. The typical resource demands to reserve as backup can be derived from the recorded argument values.

The argument recorder also learns the return addresses of calls to functions. We use the return address within the application’s binary as an unique identification of a function call. The advantages of using the return address as identification of function calls are as follows:

- The return address is independent from a certain execution and the scheduling behavior of multi-threaded applications. Thus, in multiple runs the same function calls have the same return addresses. Hence, we can easily combine the results for several dynamic analysis runs with different run configurations.

- The return address is also used to combine the dynamic analysis results with the static analysis results, since the return address of a function call can also be extracted from the disassembled code (see Section 4.4.3).

In summary, our argument recorder works as follows:

1. It generates a wrapper that intercepts all invocations of (library) functions of interest to record their argument values,
2. the application is executed with this wrapper preloaded, and
3. the recorded argument values are written to our database for the next analysis steps and patch generation.

Our argument recorder is a dynamic analysis tool and, hence, the recorded arguments are associated with specific runs of the application. There are two coverage issues with this approach. First, a call to a function of interest might not be executed in the run we used for recording the arguments. Second, the argument values of a function call might be specific to a run and not a good estimate of a typical argument value. Our approach is to run the application multiple times in different configurations within the argument recorder. The results of all runs of an application are included in all further analysis and patch generation.

In the following we describe our argument recorder implementation in more detail. Before running the application, an argument recording wrapper is generated. The purpose of this wrapper is to intercept all calls to functions of interest and to write the argument values and return address of all intercepted calls to our database. For each function of interest one wrapper function is included in the generated wrapper. The return address of the current function can be found on the stack. A wrapper function writes the current argument values and the current return address to the database and then it calls the function it wraps. Listing 4.1 shows pseudo code to illustrate the operation of a function wrapper.
Listing 4.1: The structure of a function wrapper part of the argument recording wrapper.

```c
return_type function_name(type1 arg1, type2 arg2, ...)
{
    write_to_db(return_address, arg1, arg2, ...);
    return original_function(arg1, arg2, ...);
}
```

The argument recorder gets a run configuration from the user. This configuration is saved to the database to facilitate re-execute of the run within the error injector (see Section 4.4.2). The run configuration consists of:

- the applications path,
- command line arguments,
- the working directory of this run, and
- optionally a setup script.

The setup script prepares the environment (external state) of the application. For example, the setup script can be used to remove files left over from a previous run. Therefore, the argument recorder runs the setup script and changes the working directory. Then, the application is run with the configured command line arguments. The argument recorder inserts the wrapper between the application and the library using the LD_PRELOAD approach of the Linux linker [33] (as shown in Figure 4.1 for the patch).

### 4.4.2 Systematic Error Injection

Our implementation uses systematic error injection to classify function calls as safe or as unsafe.

**Definition 4.1.** A function call is safe, if the application does not crash when the function called returns an error value. All function calls for which the application crashes at-least once if they return an error value are unsafe.

Even if a crash was only observed for one invocation in one run and all other observed invocations of this function call do not crash we treat the function call as unsafe. Unsafe function calls have to be patched. The results of the systematic error injection runs are written to the database after the classification.

The analysis works as follows: for each call to a function of interest grouped by return addresses an error injection wrapper is generated. The wrapper intercepts all calls to the function of interest. It filters the calls for a given return address. Calls that do not match the filter are pass through to the original function. For matching calls, however, a given error value is returned. Listing 4.2 illustrates with pseudo code the operation of an error injection wrapper function.
4.4 Finding Bad Error Handling

Listing 4.2: Structure of an error injection wrapper function.

```c
return_type function_name(type1 arg1, type2 arg2, ...)
{
    if (return_address == error_injection_return_address)
        return error_value;
    return original_function(arg1, arg2, ...);
}
```

The `error_injection_return_address` is constant for a given run of the error injector to filter the calls by return addresses. We repeat the process for all return addresses seen by the argument recorder.

For each function call the application is run with the corresponding wrapper preloaded. The same run configuration as for argument recording is used by the error injector. The error injector uses the same preloading approach as the argument recorder. After running the application, the error injector waits for the application to terminate. The error injector records, if the application crashes or exits and then starts the application with the next wrapper for the next return address.

Our systematic error injection makes four assumptions:

**Assumption 4.1.** The application behaves good *without* error injection, i.e., does not signal errors: our wrappers are the only parts of the system that return error values and the application will not crash because of other errors.

Note that this assumption can be checked by the wrapper during runtime. It is violated if the application crashes without error injection.

**Assumption 4.2.** The system has to know the error return value of functions of interest *a priori*.

Error values can be given by experts or automatically extracted using the approach presented in Section 4.3.

**Assumption 4.3.** The only bad behavior is a crash of the application.

If this assumption is sufficient, depends on the application. For instance, it might be useful to check the application’s output for consistency and correctness. Both properties depend on the application. But it would be unproblematic to aid the classification by a correctness checking script provided by an expert knowing the application. We omitted this step to make our approach more generally applicable.

**Assumption 4.4.** All error handling bugs are triggered by exactly one function call signaling an error.

Our approach misses bugs that are only triggered if more than one function call signals errors. The systematic error injector can be extended to search for such bugs by testing all combination of function calls. While this will be simple to implement, the number of wrappers to generate and the runs to execute will increase *exponentially* with the number
of calls to functions of interest. That is why we decided to assume that all bugs are triggered by exactly one unsafe function call.

As the error injection is a dynamic analysis technique it suffers from the same code covering drawbacks discussed already in Section 4.4.1 about argument recording.

To aid the developer in locating and fixing the found bugs, our error injector wrappers catch all signals sent to the application. The signal contains, among others, the address of the actual crash. Thus, we can provide the developer with the address where the error is injected and the address of the actual crash. If the applications binary contains debugging information, these addresses are translated into a source code file name and a line number. We have used this information to find the bugs presented in Section 4.7.2. Note that the requirement for debugging information is optional and is not needed for generating the patches.

Doing systematic error injection has a runtime complexity problem. We give more details about the complexity of the error injection in Section 4.5. Furthermore, we also propose a solution for this problem in this section. However, our current implementation matches the description of this section and is not based on the architecture proposed in Section 4.5.

4.4.3 Static Analysis

We do static analysis to aid the patch generation. The patch patterns, we are presenting in Section 4.6, need additional information that is not provided by our dynamic analysis techniques.

- The error value mapping pattern needs to know, if an unsafe function call is in a so called call group together with a safe function call.

**Definition 4.2.** A call group is a sequence of function calls that is executed completely or not at all.

This implies that there exists no jump statement and no jump target between the corresponding call statements in the application’s binary code. This pattern is used to map a failed call from an unsafe to a safe function call, i.e., in case of a failure at the unsafe call, the safe call will also return an error code.

- For preallocating resources the patch generator needs to know the argument values of the function call before the call is executed by the application. One way to get the argument values is to look into the binary code if the argument values for a specific function call are hard-coded into it.

**Definition 4.3.** A function call that has hard-coded argument values has static arguments. Otherwise the function call has dynamic arguments.

Preallocation is only possible if all argument values are known a priori. Thus, we require that all argument values have to be hard-coded into the binary for a function call with static arguments.
Our implementation includes a static analyzer to obtain call groups and static arguments from the application’s binary code. To be independent from the access to the source code of the application, the static analyzer works on the disassembled binary code. Because disassembling a binary is in general a surprisingly difficult problem, we cannot always guarantee that we can perform a static analysis on the complete code. We also check the results of the static analysis by comparing them with the data obtained by the argument recorder. The static analysis only considers function calls of which function names and return addresses (the address of the next statement after the call statement) have been found by the argument recorder. Nevertheless, there is a possibility for both false positives (false call groups or function calls with false static arguments) and false negatives (where we miss a call group or a function call with static arguments). We discuss these issues below.

We use the disassembly output of the GNU tool `objdump`\footnote{All assembler code in this chapter is in AT&T syntax.}. Looking for function calls is simple pattern matching: Whenever a call operation is found, its operand is inspected. If the operand points to a function of interest, the call itself and its return address (the address of its successor statement in the disassembly output) is saved into our database. Since functions of interest are always external functions (functions implemented in a 3rd-party library), they can be easily recognized because their names must be part of the binary. The tool `objdump` includes the function names for calls to external functions into the disassembled operands of call statements.

### Static Arguments

To identify calls with static arguments, we iterate over the predecessor statements of a function call. Because the Linux IA32 ABI\footnote{All assembler code in this chapter is in AT&T syntax.} defines that argument values to a function call have to be passed over the stack, we are looking for stack operations. We look for two possible stack operations:

1. push \(<\text{const}>\) It pushes a constant onto the stack.

2. mov \(<\text{const}>, <\text{offset}>(<\text{stack ptr}>)\) It moves a constant onto the stack.

   The stack pointer is always %esp and the offset is an integer constant (or zero if omitted).

The number and types of the arguments of a function are known a priori from its signature. Thus, the static analyzer knows the number of predecessor statements to inspect (plus the set of possible offset values if the mov operation is used). Only, if all arguments of a certain function call are hard-coded into the binary with push or mov operations, we classify the arguments of this call as static (see Definition 4.3). The arguments are extracted and saved into the database together with the name of the called function and the return address of the call.
4 Automatically Finding and Patching Bad Error Handling

Call Groups

We show a real source code example of a call group in Section 4.7.2. To identify call groups, we treat all function calls within the disassembly as one single call group. We then split this call group at the start of functions, so that all function calls within one function are part of the same call group. Next we identify all jumps and jump targets within a function and split the call groups further so that they do not contain a jump or a jump target. We only store call groups with calls to functions of interest into the database.

The start of a function is detected by the statement sequence shown in Listing 4.3.

Listing 4.3: Statements at the start of a function.

```
push %ebp
mov %esp, %ebp
```

This sequence is generated by common compilers. It first stores the frame pointer of the caller on the stack and then loads the current frame pointer. For all our analyzed programs we successfully identified the starts of functions using this approach.

Jumps are identified as operations that start with a `j` [57]. If the operand of a jump is a constant, we record it as jump target. Our implementation treats the `ret` operation also as a jump but without a jump target.

False Positives

One prominent source of false positives (misinterpreted call groups and function calls with static arguments) are indirect jumps. The target of an indirect jump is calculated during runtime. In general, it is impossible to derive the exact set of possible targets for an indirect jump from the disassembly. Our current implementation ignores indirect calls. But it would be possible to add heuristics to determine some indirect jump targets, for instance for `switch`-statements. The missed jump targets might lead to too large call groups, when such an indirect jump goes into a call group. It is also possible that such a jump target is between a function call and the statements that move its operands onto the stack. In this case our implementation mis-classifies a function call to have static arguments.

We did a survey with a set of sample C programs that contained common syntax constructs. They were compiled with different optimizations options. We inspected the resulting assembly output of the programs. We found no indirect jump as described above. Of course, this is only an indication that for the used programs and compiler indirect jumps are not an issue.

False negatives are also possible. One cause for false negatives that we observed in our survey is tail recursion. The GNU compiler `gcc` sometimes generates for function calls at the end of a function the statements shown in Listing 4.4.

```
3
The GNU compiler `gcc` inserts sometimes a `mov <constant>, %edx` statement between the two statements. Our implementation skips this additional statement.

98
Instead of calling the function and returning to the current function’s callee, the stack is restored and a `jmp` to the next function is executed. The return address of the current function is now the return address of the function `jumped to`. Because we cannot predict the function’s return address, we skip such tail calls.

If the static analysis completely misinterprets the disassembly (for instance, if the binary was altered to prevent re-engineering), there is only a little chance that the results of the static analysis match the argument recording results (remember, we compare function names and return addresses of function calls). In this case it is theoretically possible that a generated patch contains code that might be useless or, even worse, could result in some inconsistencies (for instance a static argument is assumed where there is none). But our patch generator ensures that a patch stays within the specification of the function that is called. Hence, our patch might not be able to mask certain existing errors, but the patch does not introduce additional errors because of a wrong static analysis. Here are two examples how the generated patch mitigates problems from the static analysis:

- For instance, if preallocation assumes that a function has static arguments, then it calls this function with the known static arguments to reserve backup resources. But, the generated patch always checks if the actual arguments match the static arguments and only in this case uses the reserved backup.

- If call group is analysed to be larger than in reality a error might be mapped from an unsafe function to a safe function that are not in the same call group. Because the safe function is allowed to return an error it stays in its specification. However, if because of the erroneously extracted call group the unsafe function call might be too far in the past to be handled by the safe function, then the original bug is triggered. Our patch is ineffective but not worse.

It is also possible that the unsafe function call will never be performed. Then, the safe function call returns an error where there would be none visible. However, as this only happens when the patch has a failed preallocation (see Section 4.6.1) the patch just makes an error visible that would be hidden otherwise.

### 4.5 Fast Error Injection using Virtual Machines

When doing systematic error injection, one has to do one error injection experiment per API usage of the application to test. Hence, one needs to run the application once per error injection. Thus, the runtime of all experiments together grows with $O(n^2)$ where $n$ counts API usages. One can reduce runtime complexity to $O(n)$ by utilizing snapshots and rollbacks (see Section 4.3.2 and [11]). The idea is to make a snapshot of the application directly before an error is injected. After the error injection experiment is finished the
application’s state is rolled back to the last snapshot. This approach reuses the progress an application has already achieved until the error was injected.

Figure 4.4 illustrates an error injection run with three API usages in total. The original error-free execution of the application is depicted with a bold line. Each API usage is an error injection point. Because an API usage, e.g., a function call, can fail for multiple reasons, usually more than one error needs to be injected per error injection point to do an exhaustive analysis. In the example, at the first and the third error injection point two errors need to be injected and at the second one three errors. The different errors are derived from the specified error values of the called API functions. In our experiments we are not interested in a complete run after the error has been injected. Therefore, it is sufficient to execute a limited number of instructions after injecting the error, for instance, until the currently executed function returns to its caller. Thus, the runtime complexity is $O(n)$ instead of $O(n^2)$ when doing each experiment in a separate run. If every error injection run would be executed completely until the end, the runtime complexity of the error injection would be $O(n^2)$.

To underpin the complexity issues, we have parsed the documentation of the Linux 2.6.15 system call API. We have found in average 3.3 possible error return codes (errorss) per system call. The maximum is execve with 22 unique error return codes. GCC performs more than 1,800 system calls when compiling and linking a Hello World program. Whereas compiling and linking the GNU Math Library issues more than 2.5 million system calls. If one wants to do errors injection for the latter compilation run, one has to run more than 8 million error injection experiments (3.3 errors $\times$ 2.5 million system calls). If the runtime complexity would grow quadratic, then system calls in the order of 64 trillion would have to be executed.

### 4.5.1 The fork Approach

Currently, we do the snapshot by spawning a new child process via fork (see Section 4.3.2). The error-free parent process waits until the error injection experiment in
the child process has finished. If another error has to be injected at the current error
injection point, a new child is spawned. Otherwise, the parent continues the execution to
the next error injection point. In our recent research we found that the snapshots done
with `fork` are incomplete for various reasons:

**Multi-Threading** Applications that use multiple processes to implement multi-threading are not supported by the `fork` approach.

**File System** The snapshots do not contain the state of the file system. A error injection execution might alter files in a way that influences the original execution.

**Shared Memory** If the snapshot is done via `fork`, the original execution might be influenced by changes to shared memory. For example, if an application accidentally maps private data as shared, then it will see all changes to this data done by the error injection experiments.

**Operation System Resources** The error injection execution might bind certain resources – even after its termination – that can therefore not be acquired by the original execution. One example for such resources are the System-V IPC Semaphores. A Semaphore hold by a process is not automatically freed upon its termination.

**Indirect Influence** Furthermore, a error injection execution might influence another application which itself influences the original execution again. This can be very difficult to detect. For instance, because of the injected error a signal could be send to a third application, which in turn forwards this signal to the original application. The original application would not see this signal without error injection.

**Distributed Applications** The `fork` snapshot captures only one application. Applications distributed over more than one process or more than one host cannot be taken into one consistent snapshot using `fork`.

### 4.5.2 Virtual Machines for Fault Injection

Because of the described insufficiencies, we propose to include the complete state of one computer into a snapshot. In the case of distributed applications, the states of all participating computers have to be included. In this way, we avoid interference between error injection execution and original execution. Virtualization tools provide snapshots of virtual machines that include the states of the CPU, the virtual devices, the volatile and the stable storage. But there are some problems with current VM toolkits:

- None of the VM toolkits are optimized for fast snapshots and rollbacks, and
- We need to store the analyses of error injection executions outside of the virtual machine. Otherwise, it will be discarded when rolling back to the last snapshot.

We have implemented fast snapshots and rollbacks for XEN [10]. Our plan is to avoid snapshots of the file system by splitting it into an immutable part mounted as read only
4 Automatically Finding and Patching Bad Error Handling

and a mutable part within a ram disk. Both parts are joined together by UnionFS [134]. The mutable part of the file system is stored in the virtual machine’s RAM. The RAM itself is saved within a snapshot via copy-on-write. Thus, only modified data is part of the snapshot.

The pseudo code in Listing 4.5 shows how the error injection – running within a virtual machine – cooperates with our snapshot/rollback approach.

Listing 4.5: Integrating Error Injection with a VM.

```
1 set_state (INJECT)  # writes to the external state
2 snapshot ()
3 if get_state () == INJECT:  # get_state reads external state:
4     # true after the snapshot,
5     # but false after rollback
6     do_error_injection ()
7     log_analysis_results ()
8     set_state (NEXT_ERROR)  # writes to the external state
9     rollback ()  # roll back to line 2
10 end
```

Because the rollback discards all changes done since the last snapshot, we have to store the analysis results outside of the virtual machine (log_analysis_results). Also, the information about the completion of the current error injection is lost. Therefore, we need an external state that is not influenced by the rollback. The functions get_state and set_state are used to access the external state. Currently we only need one bit.

The snapshot and rollback tool run on the VM’s host (or the privileged domain in XEN’s case) as shown in Figure 4.5. The host stores also the external state and the error injection results. The experiments are controlled from within the virtual machine VM1 by the error injection tool. If an experiment is distributed over more than one computer, all involved computers run as virtual machines on the VM’s host. In this way, a snapshot of all of them can be created.

4.6 Patching Bad Error Handling

The patches we generate are wrappers. Such wrappers are generated for each application with unsafe function calls. To apply the patch the generated wrapper has to be preloaded,
4.6 Patching Bad Error Handling

just like the fault injection wrapper, by utilizing the pre-loading feature of the dynamic linker [33]. The wrappers intercept unsafe and safe functions calls. Each wrapper is a sequence of instantiations of patch patterns. We have implemented a patch generator for each patch pattern. A micro-generator architecture [44] integrates the code produced by these generators into one single patch wrapper per application.

We use two patch patterns to fix unsafe function calls:

- Preallocation tries to ensure that the resources requested by an unsafe function call are available when the call is executed.

- Error value mapping maps the error value from unsafe function calls to safe function calls. The assumption is that the unsafe function call’s error can be handled indirectly by the safe call.

We have observed that both patch patterns work, but they cannot, of course, mask every failure occurring at an unsafe function call.

To test these patterns, we implemented them for calls to `malloc`, `calloc`, and `realloc`. The patterns are not restricted to these functions. Hence, we describe them in a more general way.

4.6.1 Error Value Mapping

We apply the error value mapping patch pattern to call groups with at least one unsafe function call and one safe function call. In short, whenever one of the unsafe function calls within a call group returns an incorrectly handled error value, our patch wrapper ensures that the safe function call also returns an error value.

**Assumption 4.5.** If a unsafe function call $U$ and a safe function call $S$ are both part of the same call group, we assume that the error handling of $S$ also handles all errors of $U$ properly.

The basic assumption of error mapping is that instead of crashing, the application handles the error of the unsafe function call together with the safe function call. In Section 4.7.2 we present a bug found in `grep` where this pattern is applicable.

Figure 4.6 shows how errors of unsafe function calls are mapped to a safe call. Function calls $f1$ to $f6$ belong to the same call group. If a call group contains more than one safe function call, we choose the last safe function call as target for error mapping. We call this safe function call **error target**. Failures of unsafe function calls with dynamic arguments executed **after** the error target cannot be mapped back to the error target, because the argument values of such calls are not known when executing the error target.

The patch works as follows: All return values of unsafe function calls before the error target are stored within the patch wrapper. When the error target call is executed the stored return values of all unsafe function calls executed **before** the error target are checked. If one of them contains an error value, the error target call returns also an error value. We call this forward mapping because errors are mapped forward to the error target.
Figure 4.6: Shows the error mapping patch pattern on a call group: errors of unsafe function calls $f_1$, $f_3$, and $f_6$ are mapped to safe call $f_4$. Unsafe function call $f_5$ can not be patched by this pattern.

Errors of unsafe function calls with static arguments executed after the error target are mapped with backward mapping: Within the intercepted error target call all unsafe function calls with static arguments after the error target are executed early. The patch wrapper stores the results of this early execution. If one of the results contains an error value, the error target call returns an error value, too. When the backward mapped function calls are performed by the application, the function calls are intercepted and the stored return values are returned. Unsafe function calls with dynamic arguments (like $f_5$ in Figure 4.6) cannot be patched because their argument values are unknown at time the error target is executed.

The pattern is applicable if the order in which the function calls are issued does not matter, i.e., there are no side-effects between these function calls. In the call group in Figure 4.6 the function call $f_6$ is performed together with $f_4$ and before $f_5$. If the order of the function calls must be preserved, no backward mapping is possible. Please note that all calls in a call group can call different functions with different error codes. Because the failed unsafe function might differ from the error target, the application’s user might see a wrong error message. To test the influence of the call order one can generate a special wrapper that shuffles the order of calls in a given function group. If the application’s results of executions with this wrapper does not differ from executions without this wrapper, it indicates that the call order does not matter.

Error Mapping without Call Groups

We have also implemented a weaker form of error value mapping: The patch wrapper can optionally return error values for all safe function calls as soon as one unsafe function call fails. This variant of error mapping does not rely on call groups. Thus, it is more generally applicable. In the terms of the previous sub-section: This variant resembles forward mapping. It maps any error of an unsafe function call to all following safe function calls.

Our implementation uses an error flag. The error flag is initially not set. When the patch detects an error return value for an unsafe function call, it handles it and sets the error flag. All safe function calls are intercepted and the error flag is checked. If it is set, an error value is returned. The assumption is – like with the previous variant of
error mapping – that the application handles the error of the safe function call together
with the unsafe one before it crashes. Usually the application will exit returning an error
message to the user. That is why we call this variant *early-exit*.

4.6.2 Preallocation

The assumption of the preallocation pattern is that one should try to reserve all resources
as soon as possible. This approach can be combined with early-exit to try to gracefully
signal a preallocation error to the application through a safe function call. Preallocation
for unsafe function calls with static arguments can be done without any additional action:
At the start of the patch wrapper it preallocates the resources for all unsafe function calls
with static arguments. Whenever an unsafe call fails, the preallocated resources are used.
Whenever a safe function call succeeds, all preallocated resources are checked if they need
to be renewed. We only renew preallocated resources in safe function calls, because in
safe function calls we can safely signal preallocation errors as errors of the safe function call
itself (see below). There are two options if the renewing fails:

- Use early-exit and signal an error value at the safe function call that does the
  renewing.

- Ignore the failure. Ignoring assumes that the renewing might succeed at a safe call
  performed later on but *before* the unsafe call belonging to the resource.

For the second option it would be possible to renew also at unsafe function calls, because
we ignore any preallocation failure.

The argument values of unsafe function calls with dynamic arguments are not known a
priori. So the exact parameters for preallocating the resources are unknown until the call
is performed. Hence, we use the argument values from the argument recorder. Therefore,
we take all argument values recorded for a specific function call and apply a heuristic to
get a set of parameters for preallocation. For example, we take the maximum argument
value seen as size parameter for malloc. Other options are to take the average or the
average plus the deviation. We did not see any different results for these options in our
experiments (see Section 4.7). When an unsafe function call with dynamic arguments
is performed and fails, our patch wrapper intercepts the call. It compares the current
argument values with the parameters used for preallocation. If they do not match, some
corrective actions can be done (e.g., trying to resize a preallocated block).

The preallocation pattern can only be used for unsafe calls to functions which reserve
some resources. So it is much more limited in its use than the error mapping pattern.
Preallocation does not preserve the order of the function calls, because for the renewing of
a resource the corresponding unsafe function call is executed out of order. Thus, prealloca-
tion is only possible if the order of execution of the wrapped functions does not matter,
for instance, if the wrapped functions have no side-effects. For dynamic arguments pre-
allocation uses arguments derived from the argument recorder. Hence, corrective actions
must be possible to adapt preallocated resources if the derived arguments do not match
the actual arguments of the unsafe function call. All these assumptions hold for the C API
for memory allocation: `malloc`, `calloc`, `realloc`, and `free`. Whenever the size of a preallocated memory chunk does not fit the current argument values, `realloc` can be used to correct the size of the allocated memory. Other functions to which preallocation can be applied are, for instance, opening files (e.g. `fopen`) and sockets (e.g. `socket`, `bind`, `connect`).

**4.6.3 Patch Generation**

The patch generator classifies all unsafe function calls learned from error injection before generating the patches.

- All unsafe function calls that can be patched by error mapping using call groups are assigned to the call group patch generator.
- All unsafe calls to memory allocating functions with static arguments that are left are assigned to the static preallocation generator.
- All unsafe function calls with dynamic arguments to memory allocation functions that are left are assigned to the dynamic preallocation wrapper.

We have implemented a micro-generator per patch pattern. Our architecture integrates all patches generated by the micro-generators into a joint patch wrapper. If there are more than one function to wrap – like in our case: `malloc`, `calloc`, and `realloc` – the process is repeated separately for each function. To simplify the presentation we focus in the following on a single function to wrap.

Listing 4.6: Pseudo code of a wrapper function of a generated patch.

```plaintext
return_type function_name(type1 arg1, type2 arg2, ...) {
  if (early_exit && is_safe(return_address)) return error_value;
  switch(return_address) {
    case backward_mapping_group1_call1:
      return preallocated_value(backward_mapping_group1_call1);
      //..
    case error_target_group1:
      if (!group_failed(group1)) // executed backward mapped unsafe calls with static arguments
        if (group_failed(group1) || has_error( backward_mapping_group1_call1) ...) {
          set early_exit;
          unset group_failed(group1);
          return error_value;
        }
  }
}
```
4.6 Patching Bad Error Handling

```c
unset group_failed(group1);
break;
}
return_value = call_to_wrapped_function(...);
if (return_value != error_value && need_preallocate) {
    // read need_preallocate and
    // preallocate static and dynamic buffers
    // by calling the original function (e.g., malloc)
    // with the arguments stored in need_preallocate
}
switch(return_address) {
    case forward_mapping_group1_call1:
        if (return_value == error_value) {
            set early_exit;
            set group_failed(group1);
        }
        break;
    //..
    case unsafe_call_static_args1:
        if (return_value == error_value) {
            set early_exit;
            return_value = preallocated_value(
                           unsafe_call_static_args1);
            set need_preallocate(unsafe_call_static_args1);
        }
        break;
    //..
    case unsafe_call_dynamic_args1:
        if (return_value == error_value) {
            set early_exit;
            if (preallocation_params(unsafe_call_dynamic_args1)
                != current_args) {
                // try corrective actions
            }
            if (preallocation_params(unsafe_call_dynamic_args1)
                == current_args || correction_successful) {
                return_value = preallocated_value(
                               unsafe_call_dynamic_args1);
                set need_preallocate(unsafe_call_dynamic_args1);
            }
        }
        break;
    }
```
Listing 4.6 shows the pseudo code of a wrapper function of a generated patch. The only difference between patches with and without early-exit are the lines with references to early_exit. These lines are only part of early-exit patches. The generated function intercepts all calls to a given function. It identifies certain calls by their return address. The code generated for a certain call depends on the patch pattern used for that call.

After intercepting a function call an early-exit patch checks if the last unsafe call failed and the current call is a safe one: if true, the wrapper returns an error value (line 2). A non-early-exit patch skips this step. The error mapping patch pattern for call groups inserts some code before executing the wrapped function. The patch returns the return values for unsafe calls with static arguments executed after the error target (line 5). At the error target itself, those backward mapped unsafe calls are pre-executed (line 10). Because the unsafe calls of a call group might invoke functions different from the error target, library functions that are different from the intercepted one might be invoked for pre-execution. For instance, the error target can be a calloc whereas the backward mapped unsafe function call goes to a malloc. If one of the pre-executions fails or one of the forward mapped calls has marked the group as failed, an error value is returned (line 14). We assume that the returned error will be handled correctly (Assumption 4.5). In both cases the flag that marks the group as failed is reset for the next execution (line 13).

After executing the original function call (line 19) without a failure, preallocation is tried for following all unsafe calls, that are patched by the preallocation pattern (line 22). If the original function call returned an error value and the current call is patched by forward mapping the corresponding call group is marked as failed for this execution (line 28). For unsafe calls with static arguments the preallocated resource is returned (line 35) and it is marked for a new preallocation. The next safe function call will try to do the renew the preallocated resource in line 22. The action for calls with dynamic arguments is similar. Additionally, the current argument values are checked in line 46 if they match the predicted arguments used for preallocation in line 22. Corrective actions are taken, if the arguments do not match. After the optional corrective actions the arguments are checked again, because the corrective actions might have failed. If the check is true, then the preallocated resource is used and marked for renewal at the next safe function call. If the checks fails we are not able to mask this bug, because we must return the error value for this unsafe function call. In the (common) case that none of the executed upper code lines have already exited from the wrapper, the wrapper passes the control back to the caller by returning the return value (line 56).

4.7 Evaluation

We have implemented our approach on top of Ubuntu Linux 5.10. Our four components (argument recorder, error injector, static analysis, and patch generator) are implemented in Ruby 1.8 and we use Postgresql 8.0 as database back end. All generated wrapper code
4.7 Evaluation

is in C. Our implementation is currently restricted to IA32. This is mostly because of
the static analysis component. At least the regular expressions used by this component
would have to be reimplemented for other platforms.

We have applied our approach to ten command line applications that are part of the
standard installation of Ubuntu Linux. Each of the ten applications crashed at least
once during error injection. Our generated patches are able to cope with up to 84% of the
unsafe function calls we have found. The worst runtime overhead of a patched application
was 9.14%.

4.7.1 Measurements

First, we present the number of unsafe function calls we have found for each of the ten
applications. After that we discuss the effectiveness of the generated patches that we
examined with the help of various error injectors. Furthermore, we present measurements
regarding the overhead of the patches. Finally, in Section 4.7.2 we will describe some
concrete bugs that our tools have found.

Unsafe Function Calls

Figure 4.7 shows the number of unsafe calls per application and how many of that calls
have static arguments. Since our patches are currently limited to the function of the C
memory managing API (malloc, calloc, and realloc), we have only simulated errors
of these functions. All applications suffer from at least one unsafe function call. Except
for grep and du, the unsafe function calls are performed by the Standard C Library.
The crashes are also within the library. For sum, uname, wc, df, md5sum, sort, and
touch the crash happens while executing the Standard C Library function setlocale.
Application unzip crashed while executing the Standard C Library function tzset.
Since the unsafe function calls are part of the Standard C Library, they are not taken
into account within the static analysis. Therefore, our tools assume that these calls have
dynamic arguments. In grep 4 of the 6 unsafe function calls have static arguments and
in du 2 of 4 unsafe function calls have static arguments.
Robustness Evaluation of Patches

Each patched application was first tested without error injection. All runs had the expected output. So, none of the generated patches did any harm in the absence of failures in our test setup. To stress the patches, we run various error injection experiments with the un-patched and patched applications. The results for patches without early-exit are shown in Figure 4.8. We used four different error injectors to stress the generated patches:

- Figure 4.8 (a) shows the results obtained with the systematic error injector used to find the bad error handling in the first place (Section 4.4.2). We made one run for each function call \( f \) and injected only errors that could be returned by \( f \).

- The knock-over error injector simulates complete resource depletion (Figure 4.8 (b)). It works like the systematic error injector (one run per function call), but after it returned an error value for the first time, it returns an error value for all following calls (independent of the return address, i.e., call site) of the current run. This stresses the error handling code, for example retrying an allocation does not work with this error injection scheme. Except for \texttt{grep} and \texttt{unzip}, we have found more crashes with the knock-over error injector than for the systematic error injector.

- In Figure 4.8 (c) we simulated a maximum amount of memory. The amount of currently available memory is limited by the error injector. Whenever an application tries to exceed this amount, an error value is returned. We experimented with
amounts of 100, 1,000, 10,000, 100,000, 1,000,000 and 10,000,000 bytes. The most crashes were observed for the limit of 1,000 bytes. All of these crashes happened within the C-library at the same return address before the actual application code got executed. We did not examine this bug any further.

- Figure 4.8 (d) depicts a probabilistic error injection. Error values are returned with a given probability per function call. We run 12 experiments with a fixed seed for reproducibility. One bug in the un-patched du results in a crash while our patch prevents this crash. Within wc none of the 4 found crashes are prevented by the patch. And even worse for grep our patch introduces an additional crash. We assume that the additional crash is a result of the preallocation. Our patch alters the ordering and the amount of memory reserved by the application. That is why a run with the patch behaves differently from a run without the patch.

We only counted crashes, but not graceful degradation like termination with an error-code (Assumption 4.3). Overall we found 79 crashes. Our generated patches prevent 64 of them – that are 81% of all crashes. This rate includes the additional crash of grep with the patch. We ran the same experiments with the early-exit patch. The numbers are the same except two additionally prevented crashes for du (with systematic and knockover error injection). So this patch prevents 66 of 79 crashes (84%).

**Performance Overhead**

Figure 4.9 shows the run times of the applications with normal and early-exit patches relative to the run times without patches. All runs were done without error injection. Each measurement was repeated 25 times and the results were averaged. The largest runtime overhead is 9.14% (for uname without early-exit). Surprisingly, the smallest is −4.88% for touch with early-exit, i.e., the generated patch accelerates the application. Experiments indicate that the wrapper preloading is responsible for the speedup. Running touch with a wrapper that intercepts malloc without any additional actions takes in average 16 ms versus 21 ms for running touch without a wrapper. We see no evidence that the early-exit patch outperforms the normal patch or vice versa. We conclude that our patches add little to no runtime overhead.

**4.7.2 Bugs Found**

In this section we present a few of the error handling bugs we have found with our tool. We explain the bugs and how the generated patch masks them.

**Missing Error Handling**

We found a common bug pattern in grep:

Listing 4.7: grep-5.2.1 src/search.c:152

```c
152 char *mb_properties = malloc(size);
153 mbstate_t cur_state;
```
4 Automatically Finding and Patching Bad Error Handling

![Figure 4.9: Runtimes relative to the unpatched application in %.](image)

Memory is reserved in line 152 and it is accessed without any error handling in line 157. Error mapping will not be successful here as there is no safe function call between the unsafe call and the memory access to map to. The only thing the generated patch can do is preallocation. It tries to ensure that the unsafe call to `malloc` in line 152 always returns a pointer to a preallocated buffer. Of course, that might not be possible under all circumstances. It gets even more difficult because the argument of the unsafe call is dynamic. Our patch can prevent some crashes but we cannot guarantee that we can prevent all. The general solution is of course that the developer adds proper error handling code.

Error handling can be done in many ways. Two common strategies are:

**Test Early** This strategy should be sufficient for the bug described above.

**Test before every use** This could be the strategy the developers had intended for the next bug. But at least before one usage, the error handling is missing.

The next bug is in the hash table implementation used by `du`:

```c
154 wchar_t wc;
155 int i;
156 memset(&cur_state, 0, sizeof(mbstate_t));
157 memset(mb_properties, 0, sizeof(char)*size);
```

<table>
<thead>
<tr>
<th>Listing 4.8: coreutils-2.5.1 lib/hash.c:537</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash_table * hash_initialize (size_t candidate, const Hash_tuning *tuning,</td>
</tr>
<tr>
<td>Hash_hasher hasher, Hash_comparator comparator,</td>
</tr>
<tr>
<td>Hash_data_freer data_freer)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>Hash_table *table;</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Listing 4.9: coreutils-2.5.1 lib/hash.c:578</th>
</tr>
</thead>
<tbody>
<tr>
<td>537 Hash_table *</td>
</tr>
<tr>
<td>538 hash_initialize (size_t candidate, const Hash_tuning *tuning,</td>
</tr>
<tr>
<td>539  Hash_hasher hasher, Hash_comparator comparator,</td>
</tr>
<tr>
<td>540  Hash_data_freer data_freer)</td>
</tr>
<tr>
<td>541 {</td>
</tr>
<tr>
<td>542  Hash_table *table;</td>
</tr>
</tbody>
</table>
Table -> bucket = calloc (table -> n_buckets, sizeof *table -> bucket);

Listing 4.10: coreutils-2.5.1 lib/hash.c:591

591 return table;

In function hash_initialize the memory for a hash table is reserved and initialized. While for the table itself a proper error handling is performed, there is none for the call to calloc in line 578. We found no code that accesses table -> bucket directly outside of the hash implementation. So it might work if before each use of table -> bucket the hash implementation would check for an error value. But the next function in the source code file accesses the bucket without any check:

Listing 4.11: coreutils-2.5.1 lib/hash.c:602

void hash_clear (Hash_table *table)
{
    struct hash_entry *bucket;
    for (bucket = table -> bucket; bucket < table -> bucket_limit; bucket++)
    {
        if (bucket -> data)
        {
            Error value mapping with early-exit can prevent a crash. This call has also dynamic arguments. Thus, preallocation might not work, if the predicted arguments are wrong and corrective actions fail.

Bad Error Handling

The last bug we present tries to handle all possibly failing function calls:

Listing 4.12: grep-5.2.1 src/dfa.c:3423

3423 mp[i].in = (char **) malloc(sizeof *mp[i].in);
3424 mp[i].left = malloc(2);
3425 mp[i].right = malloc(2);
3426 mp[i].is = malloc(2);
3427 if (mp[i].in == NULL || mp[i].left == NULL ||
3428    mp[i].right == NULL || mp[i].is == NULL)
3429    goto done;

Listing 4.13: grep-5.2.1 src/dfa.c:3622

3622 done:
4 Automatically Finding and Patching Bad Error Handling

Listing 4.14: grep-5.2.1 src/dfa.c:3633

for (i = 0; i <= dfa->tindex; ++i)
{
    freelist(mp[i].in);
    ifree((char *) mp[i].in);
    ifree(mp[i].left);
    ifree(mp[i].right);
    ifree(mp[i].is);
}

But the error handling itself contains a bug:

Listing 4.15: grep-5.2.1 src/dfa.c:3240

static void freelist (char **cpp)
{
    int i;

    if (cpp == NULL) return;
    for (i = 0; cpp[i] != NULL; ++i) {
        free(cpp[i]);
        cpp[i] = NULL;
    }
}

The error handling starts in line 3427 directly after reserving some memory. If one of the reservations fail already reserved resources are freed and the current function returns. But what if only one of the three calls from line 3424 to line 3426 fails? The error handling code is executed and freelist tries to free the uninitialized list mp[i].in. If mp[i].in itself is NULL, i.e., malloc in line 3423 failed, freelist returns. But if mp[i].in is allocated it is not initialized in this context. Hence freelist walks outside of mp[i].in and the application crashes (see for loop in freelist from line 3245 to line 3247).

A developer would possibly fix this bug by changing the call in line 3423 from malloc to calloc. Function calloc initializes the allocated memory with zeros. Hence, the list will be initialized and the for loop (line 3245) will not leave the list. But our static analysis finds a call group with one safe call (line 3423) and three unsafe ones (lines 3424–3426). The unsafe calls have static arguments. Thus, the error value mapping within a call group can be applied. When the call in line 3423 is issued, the memory allocations of the next three calls are pre-executed. When one of this four calls fail the patch returns an error value. This error value will be handled properly. For the other three calls it returns the memory allocated via pre-execution. In this way the generated patch prevents the crash by mapping the error values from three unsafe calls to a safe one.

114
4.8 Conclusion

We have introduced a novel approach to detect and patch bad error handling. The basic idea is to use error injection to locate calls to library functions that do not perform proper error handling. We then use static analysis to determine if and what type of patch can be used to correct such a call. We show the effectiveness of our approach regarding several open source programs: we can reduce the number of potential crash failures without introducing any unacceptable performance penalties.

Our evaluation shows theses 6, 7, and 8. We found bugs in real world applications with error injection (Thesis 7: “Fault injection finds failures even in mature software.”). Our patches can mask up to 84% of the failures introduced by these bugs (Thesis 6 “Automatic bug removal based on patch patterns can decrease the number of failures.”). The examples in Section 4.7.2 and our evaluation of our patches in Section 4.7.1 show the usefulness of error mapping (Thesis 8 “Automatic error mapping can mask failures.”).

Patching of bad error handling is orthogonal to the problem solved in the next chapter: patching of (library) functions called with unsafe arguments.
5 Robustness and Security Hardening of COTS Software Libraries

When building dependable systems, one can rarely afford to build everything from scratch. This means that one needs to build systems using software components implemented by third parties. These software components might have been designed and implemented for less critical application domains. The use of such components without further hardening is therefore not recommended for dependable systems.

Third party software components are often provided in form of libraries. In this chapter we focus on crashes within a 3rd-party library when given certain input arguments. For instance, some implementations of the C function `free` crash when the input to `free` is a `NULL` pointer. In the previous chapter our goal was to harden applications by working around buggy error handling in the application. In this chapter we want to harden 3rd-party libraries against crashes because of invalid input.

Experience with an existing library hardening tool HEALERS [45] has shown that such tools can help users and developers to harden libraries. The idea of HEALERS is to use automated fault injection experiments to determine automatically the robust argument types of functions. Any argument that does not belong to the given robust argument type will result in a crash of the function. The robust argument types of a function are derived from fixed hierarchy of predefined robust argument types. The predefined robust argument type hierarchy itself is part of the derivation algorithm. For example, HEALERS is able to automatically determine that the C function `strcpy(d,s)` requires `s` to be a string and `d` has to be a writable buffer with a length of at least `strlen(s)+1` bytes.

Based on the extracted robust argument type a hardening wrapper is generated. This wrapper filters the input values for any call to a hardened function. If not all argument values of the called function have robust argument types the original function is not called. Instead an error code is returned. Thus, this hardening wrapper is a patch that removes bugs from the library (Thesis 6: “Automatic bug removal based on patch patterns can decrease the number of failures.”).

Our experience with HEALERS shows however deficiencies regarding

1. the extensibility, and
2. the performance of the tool.

These deficiencies need to be addressed to make such hardening tools more widely applicable. Our new tool Autocannon addresses these issues by facilitating

\footnote{This chapter is based on [119].}
Robustness and Security Hardening of COTS Software Libraries

**Extensible fault injections** New test case generators can be added, e.g., to test new handle types more carefully (Section 5.3).

**Extensible runtime checking** New checks for arguments can easily be added (Section 5.4).

**Flexible computation of robust argument types** These are computed without a given type hierarchy and can be automatically extended with new runtime checks (Section 5.5).

**Performance** The use of static analysis can reduce the number of fault injections dramatically (Section 5.6).

We applied our approach to the Apache Portable Runtime (APR) [2]. Our fault injection experiments found bugs in over 80% of the 148 analyzed functions (Thesis 7: “Fault injection finds failures even in mature software.”). However, our hardening patch is able to prevent of 56% of all crashes seen without the wrapper (Thesis 9: “Automatic filtering can mask a high percentage of failures injected by bit-flip faults.”).

5.1 Related Work

In this chapter, we combine dependability benchmarking [62, 63, 17, 70] and automatic patch generation [45, 132, 117, 107]. HEALERS [45] already presents a general approach to harden COTS libraries. But HEALERS contains an inflexible type system that couples test types and checks: that makes it very difficult to extend. The mapping from argument types to test types is done via a predefined map. Thus, HEALERS is not able to test functions that have unsupported argument types. For example, HEALERS is only able to test four functions of the Apache Portable Runtime. All other functions contain unknown argument types. Furthermore, HEALERS cannot tolerate contradictions. But all of those four testable functions produced contradictions. Hence, HEALERS is unable to generate any protection hypothesis for the APR.

AutoPatch (Chapter 4) presents an approach to patch bad error handling. Autocannon might introduce unexpected error values when a wrapper filters a function call and returns an error value. The application might not be able to deal with this unexpected errors and might itself behave unrobust or insecure. One can apply the bad error handling patching to counter this problem. Thus, Autocannon and AutoPatch are orthogonal to each other.

Stelios et al. [107] introduced an approach to automatically patch buffer overrun bugs in applications. They also evaluated their approach with Apache. But they fix bugs within the application and not at the interface to dynamic libraries. Hence, their approach is not comparable to ours. Furthermore, the approach needs some code that exploits the bug to patch, for instance a zero-day exploit. We are, in contrast, able to detect bugs and patch them without any exploiting code as input for our approach.

Our current work is based partly on Ballista [62, 63], a dependability benchmark for POSIX implementations. We present in Section 5.3 in detail how we extended Ballista to our test system that is part of Autocannon. The biggest difference is, that Ballista is bound to a specific API while Autocannon is more general and can test arbitrary APIs.
5.2 Approach

Our goal is to automatically increase the robustness and security of third-party libraries. To be applicable for users without expert knowledge, we want to minimize the required user input. However, most developers will want to have control over the generation of protection wrappers. Hence, we permit developers to verify and modify the robust argument types that our tool system derives. The tool also provides developers with evidence (in form of a truth table) about why it derived certain robust argument types and that these are reasonable for the wrapped application.

To improve the applicability of our approach, we do not require access to the source code of the applications and libraries that need to be hardened. Source code might not always be available. And dealing with source code that was written for different compilers and even for different programming languages is very difficult and time consuming to get right (e.g., see [35]). However, we demonstrate in Section 5.6 that static analysis helps to reduce the complexity problem of fault injections. For doing the static analysis we assume that we have access to the intermediate bytecode of libraries and applications. Since intermediate languages like MSIL [16] and LLVM [65] become more widespread, we believe to have access to the bytecode is a reasonable assumption.
In this chapter we focus on the interface between an application and its dynamic linked libraries. Our approach protects the libraries used by an application from unrobust and insecure input from the application. In contrast to Chapter ?? we want to protect the library from the application’s input and not the application from the library’s output. Figure 5.1 illustrates the work-flow of our approach. In the analysis stage the libraries used by an application are first statically analyzed to reduce the number of test inputs used during the dynamic analysis phase. In the following dynamic analysis phase the library’s functions are exercised by fault injection. Based on the observed behavior of the libraries protection hypotheses are generated. A hypothesis is a boolean expression over predicates of the arguments to a function. We refer to these predicates as checks. In the second stage a protection wrapper is generated. The wrapper protects the library functions at runtime from being called with unrobust or insecure argument values. Therefore, the wrapper intercepts all function calls from the application into a library. The protection wrapper only forwards the current argument values to the wrapped library function, if they satisfy the protection hypothesis. If the argument values do not satisfy the protection hypothesis an error code is returned instead of executing the wrapped function.

The two main contributions of this chapter are a new flexible fault injection tool for libraries (compared to HEALERS \cite{healers} and Ballista \cite{ballista1, ballista2}) and a table-based approach to generate the protection hypotheses for the protection wrapper.

We evaluate the robustness of a library with fault injection. Figure 5.2 shows that Autocannon’s fault injector calls the library’s functions with different inputs. For each executed call the fault injector measures the robustness. Autocannon makes use of static analysis to determine the set of input values for fault injection on arbitrary functions.

For each library function a table like the one in Table 5.1 is built in the analysis stage. One row represents one fault injection experiment with the function. The input vectors (italic) are not part of the truth table. They only denote where the rows come from. All test values used in the dynamic analysis stage are classified by boolean checks. A 0 means the check evaluates to false, 1 to true. A check is a predicate over one or more argument values from the test case. For instance, string?(a) evaluates to true if argument a is a pointer to a 0-terminated character buffer (a string). The right most column contains the boolean result of the call: either robust or unrobust (1 or 0, respectively). A function’s execution is robust, if the function does not crash (e.g., by a segmentation fault). We show in Section 5.3.3 how to create test values in a way that security violations (like buffer
overflows) are converted into robustness issues, i.e., a crash. Our approach to generate a protection hypothesis is to minimize the truth table to a boolean expression. We view this truth table as a boolean function \( f(\text{check}_1, \ldots, \text{check}_n) \). A truth table minimizer computes a boolean expression of \( f \). This boolean expression is the protection hypothesis. It takes the role of HEALERS’ robust argument types but it is much more flexible because it is an arbitrary boolean expression over a set of given checks.

Table 5.2 sketches the truth table for the Standard C function `strcpy`. The truth table is first preprocessed before minimizing it. In the preprocessed step among others removes the redundant row 3 from Table 5.2. Check \( \text{check}_1 \) is true if argument `src` points to a string, \( \text{check}_2 \) is true, if the first `strlen(src) + 1` bytes of the buffer pointed to by `dest` are writeable. The resulting protection hypothesis is: `string?(src) AND buf_write?(dest, strlen(src) + 1)`. The protection wrapper will reject all inputs for which the protection hypothesis does not evaluate to true.

The protection wrapper prevents calling a function \( f \) with unsafe arguments. Therefore, it evaluates the protection hypothesis of \( f \) on the current argument values given by the caller of \( f \). Only if the evaluation yields true, the control is passed to \( f \). Otherwise the wrapper returns with an error code without executing \( f \).

In order to use our approach, some knowledge about the library functions used by an application is needed: we need at least to know the return type and the types of the arguments of a function to perform the dynamic analysis. Depending on the target platform, this information might already be included within the library (e.g., a library given in LLVM bytecode). Otherwise, some other form of specification (such as C header files) is needed. In the following we will refer with public source to files that contain this information. Note that even if this is part of the source code of the library, it must be also available for developers using the library. Whereas private source is the source code of the library implementation, which is typically not available to all users and third-party developers.

Even though our approach is independent of the programming language and platform of a library, we will focus in the following on libraries implemented in the programming language C.

<table>
<thead>
<tr>
<th>input vector</th>
<th>check(_1)</th>
<th>check(_2)</th>
<th>\ldots</th>
<th>check(_n)</th>
<th>robust?</th>
</tr>
</thead>
<tbody>
<tr>
<td>test case(_1)</td>
<td>0</td>
<td>1</td>
<td>\ldots</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>test case(_2)</td>
<td>0</td>
<td>0</td>
<td>\ldots</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>test case(_3)</td>
<td>0</td>
<td>1</td>
<td>\ldots</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>test case(_m)</td>
<td>1</td>
<td>0</td>
<td>\ldots</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: Table based approach to generate robustness and security checks. For a check the 0 means that a given check evaluates to false. A 1 means true.
5 Robustness and Security Hardening of COTS Software Libraries

\[
\begin{array}{ccc}
\text{check}_1 & \text{check}_2 & \text{robust} \\
0 & 0 & 0 \\
1 & 0 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

\(\text{check}_1 = \text{string}?(\text{src})\)
\(\text{check}_2 = \text{buf}\_\text{write}?(\text{dest}, \text{strlen}(	ext{src}) + 1)\)

Table 5.2: Part of truth table for the function \text{strcpy}(\text{char* dest, const char* src}).

5.3 Test Values

Test values are used in the analysis phase as input values for performing fault injection experiments with the libraries functions. For each of a function’s arguments a set of test values is generated. The set of input vectors is the cross product of all test value sets. For example, for function \text{void* calloc(size_t nmemb, size_t size)} Autocannon computes first the test values for argument \text{nmemb} based on its type \text{size_t} and then for argument \text{size}. In this case the test value sets of \text{nmemb} and \text{size} are the same, because both arguments have the same type \text{size_t}. The cross-product of the two test value sets is the set of input vectors for doing fault injection into \text{calloc}. If the test value set of type \text{size_t} is \{0, 10\}, then Autocannon calls \text{calloc} with the input vectors \{(0, 0), (0, 10), (10, 0), (10, 10)\}. In reality the test value set for type \text{size_t} is much larger.

The test value sets must be large enough to exercise the function under analysis in a way that the resulting truth table is sufficiently complete. Of course, executing the function on all possible input values is in general infeasible, e.g., a function with two 32-bit integer arguments has more than \(1.8 \cdot 10^{19}\) possible input vectors.

In comparison to HEALERS and Ballista, we want Autocannon to handle arbitrary functions. Therefore, Autocannon has an extensible test type system. New test types can be added without any further change to the generation of the protection hypothesis. Each argument type of a function is mapped to a test type (see Section 5.3.5). A test type has one or more representative test values. The test type system is richer than the argument type system of C in the sense that the test type system has more semantics. For instance, the C type \text{char*} can be a pointer, a string, a file name or a format string depending on its usage. Generic pointers, strings, file names and format strings are test types. Some test types contain other test types (e.g., file names and format string are also plain strings, which itself are pointers). We call file names and format strings special test types. In general, special test types have very clear semantics. They are often “sub-types” of more generic types with more vague semantics.

Our test type system is based on Ballista’s test type system [62, 63]. Ballista is a dependability benchmark for POSIX implementations. Ballista’s test type system can handle a predefined fixed set of argument types and functions. In order to test arbitrary
libraries, we extended it to handle arbitrary argument types. We use argument type characteristics to map an argument type to a test type (e.g., its size, if an argument type is a pointer, or if it can be casted to an integer). The argument type characteristics are extracted using static analysis on the public sources (e.g., C header files or bytecode).

We introduce meta types, type templates, and feedback with Autocannon. A meta type combines a set of Ballista’s specialized types into one general type. A specialized test type implements applications semantics. For instance, a test type that represents pointers to a time data structure `time_t*` is a specialized test type. However, Ballista has some specialized test types like `file name` that cannot directly be used without further semantic knowledge of an argument type. For instance, when Autocannon sees the argument type `char*` it needs to test this argument as a generic string. Since Autocannon does not know any further semantics, the argument type may be also more specialized than a generic string. We want Autocannon to test this argument also as file name and format string. We show in Section 5.3.2 how a meta types combines several specialized test types. For `char*` Autocannon uses the meta test type `meta_string` that includes generic strings, file names and format strings.

In Section 5.3.3 we discuss how Autocannon uses feedback regarding test types to further refine the test types’ test values. We use this feedback mechanism to detect conditions for buffer overruns.

We do not want to rely in Ballista’s specialized types only. That’s why we introduce type templates to generate specialized test types from type characteristics (see Section 5.3.4). We use these templates to generate test types for handles of abstract data types and for data structures.

The meta types have only predefined test values. Meta types come from two sources. Some are predefined meta types and are shipped with Autocannon. The others are dynamically generated after the static analysis. Feedback types adapt dynamically while performing fault injection. Template types are dynamically generated from the results of the static analysis phase.

In the following part of this section we first briefly introduce Ballista’s test type system before we present Autocannon’s improvements to Ballista: meta types (Section 5.3.2), feedback (Section 5.3.3), and type templates (Section 5.3.4). The second part explains the mapping from argument types to test types (Section 5.3.5), and how we deal with a large set of input vectors (Section 5.3.6).

### 5.3.1 Ballista Type System

Ballista is a test system for measuring the dependability of POSIX implementations. It contains a flexible test type system that we have extended to meet our requirements. Ballista’s test type system is extensible: one can easily add new test types to it. All test types are arranged in a single type tree. Part of this tree is shown in Figure 5.3. The root is the most general type, the leaves are the most special ones. A child type can also a parent type. For instance, on one hand, the string type is a child of the more general pointer type. On the other hand, the string is also a parent of the more special file name type. Each type has a set of test values that are representatives of this test type. The
root’s set of test values is empty.

A child type inherits all test values of its parent. For instance, a function expecting a file name is also tested with plain string values and general pointer values, but not with format strings. But a function whose argument is mapped to the plain string type is not tested with file names and format strings. Ballista contains already fixed mappings for a given set of functions. This mapping was generated manually with regard of the function’s semantics. For instance, for function open the first argument must be a file name. Whereas, it does not give any additional value to test function strdup with something more special than a generic string.

To test a function with usual and exceptional values a Ballista type has more than one test value. For example, the Ballista type for file names has 432 test values. These 432 test values are the result of all possible combinations for the content (empty, non-empty), the file permissions (readable, writeable, etc.), the file state (existing, non-existing, directory, etc.), and the file name (local, temporary, with spaces, etc.).

### 5.3.2 Meta Types

The mapping between test types and argument types is predefined within Ballista. It is done by an expert knowing the function’s specification. In order to test arbitrary libraries, our system works without such a predefined mapping. In Autocannon, we introduced meta types to combine specialized Ballista types into more general types. A meta type has more than one parent. So our type system is a directed acyclic graph instead of a tree. For simplicity meta test types do not contribute test values. They join the test values of their parents.

For example, the test values of the meta type meta_string in Figure 5.3 are the union of the test values of Ballista’s test types file name, format string, string, and pointer. Because we do not know the specification of a function foo(char*), we map the argument type char* to meta_string. Hence, foo will be tested with file names, format strings, and their parents.

We have defined 4 generic meta types:
5.3 Test Values

Meta string combines all special string types.

Meta pointer combines all specialized pointer types including meta string.

Meta integer combines all integer types with width 32 bit.

Meta short combines all integer types with width 16 bits.

More meta types are generated dynamically when needed to combine generated types (such as instantiated type templates from Section 5.3.4) with Ballista types or predefined meta types.

5.3.3 Feedback

Autocannon uses feedback to determine the minimal size for buffers arguments. Since our goal is to protect arbitrary functions Autocannon must be able to derive the size of a buffer argument. Consider function int sum5(int buf[]) that returns the sum of the first five integer values of buffer buf. Autocannon does not know the semantics of sum5. However, Autocannon needs to know the minimal size of buffer buf (in this case five integers) in order to protect sum5 from inputs with too small buffers. The key idea is to use the feedback loop from Figure 5.4. This feedback loop is originally a part of HEALERS [44].

Feedback is used to convert buffer overruns into robustness issues. Test types that represent buffers, for instance pointers, have additional dynamic behavior to adapt their test values while doing fault injection. We guard any buffer used as test value with unaccessible guard memory pages. Hence, a buffer overflow results in a segmentation fault. Autocannon catches the segmentation fault and the address of the erroneous memory access. If the bad memory access occurred within the enclosing memory pages of the current test value, then Autocannon gives feedback to the test type. The test type enlarges the test value (the buffer) and the test is repeated. If the memory access is outside of the guard pages the current test case is treat as unrobust. The feedback cycle goes on until either the function executes without a segmentation fault or the test type is unable to allocate more memory.

The test system itself does not detect the conditions for buffer overruns. This is done by the checks. Let us illustrate this using function strcpy(char* dest, const
The test system tests `strcpy` with a string (`src`) provided by Ballista’s string test type and a read/write buffer of size 1 byte (`dest`) provided by Autocannon’s feedback test type. Function `strcpy` will overrun `dest` and cause a segmentation fault if the length of `src`’s string is larger than 1 byte. Successively, the feedback type for argument `dest` will enlarge `dest`’s buffer until its size equals `src`’s string length + 1. The check correlate the buffer’s size and the string’s length so that the hypothesis `sizeof(dest) > strlen(src)` will be finally extracted.

We have added a test type to Autocannon that (1) accepts and responds to feedback and (2) generates different representatives: buffers of different size (1 byte, 4 kbytes, 64 kbytes), content and protection (read only, read/write). Beside this one feedback test type, templates are used to generate special feedback test types for data structures.

### 5.3.4 Type Templates

We use type templates to generate specialized test types for arbitrary argument types. Type templates are parameterized test types. Currently, we have two kinds of type templates: one for Abstract Data Types and one for data structures. The data structure type gives feedback back to the testing environment (see Section 5.3.3).

#### Abstract Data Types

Abstract Data Types (ADT) are often implemented by a set of functions operating on some hidden state. This hidden state, the instance of the ADT, is referenced by a handle. Examples for such handles are file descriptor handles, socket handles, or a handle to a random number generator.

The Ballista type system already includes a set of test types generating handle values. But this handles refer all to ADTs implemented by the POSIX API. To test libraries implementing arbitrary APIs, we generate a test type for each kind of handle we identified (see Section 5.3.5). In this way, functions expecting handle values are not only tested with extreme values, but also with input values that might appear in a non-faulty execution.

For generating a test type of an ADT, we need all constructor functions that create handles to this ADT instance. Currently, we detect handles by static analysis applied on the public source code of the analysed library. Our static analysis uses some simple heuristics like the ones used in [36]. For example, we only consider functions as constructors if they follow our naming convention (see below). We can combine our static analysis with dynamic analysis like temporal specification mining [137] or the data-flow analysis used by the dynamic learner of SwitchBlade (see Chapter 2.4.2).

Our static analysis is based on the following assumption:

**Assumption 5.1.** Each ADT is represented by a handle implemented by a unique C type. The constructors and destructors of an ADT fulfill our naming convention.

First, we extract all signatures of the functions implemented by a library using Doxygen [125]. Our heuristic is that we examine only functions that fulfill our naming convention for potential constructors and destructors:
Constructors The function name must contain alloc, create, new or open.

Destructors The function name must contain close, destroy, delete or free.

A constructor passes a handle to its caller as return value or via call-by-reference. We treat every C type that fulfills all of the following properties as handle:

- There is at least one potential constructor function that returns this C type or has one argument that is pointer to this C type.
- There is exactly one potential destructor that has this C type as argument.

So our tool does not extract the ADT itself but the implementing handle.

For example, the libapr [2] contains the ADT socket. Autocannon has extracted the handle type apr_socket_t* that represents this the socket ADT. The ADT has two constructors:

```c
1 apr_status_t apr_socket_create_ex(apr_socket_t**, int, int, int, apr_pool_t*)
2 apr_status_t apr_socket_create(apr_socket_t**, int, int, apr_pool_t*)
```

Both constructors return the handles by reference. We have found 44 functions (including the destructor) that have an argument of type apr_socket_t*. These functions will be tested with test values we generate with the help of the found constructors.

We create test values for the handle types by calling the constructors of the handle. In addition to the constructors, we need input vectors to call them for creating test values. Before we create test values for handle types we first do the fault injection for the constructors. Then, we do look into the truth tables used to compute the protection hypotheses of the constructors. We extract all input vectors for which the constructor function did not crash. These input vectors are inserted into the type template for the handle type to call the constructors.

Data Structures

The data structure template complements the feedback giving test types. Because C is weakly typed a data structure (defined by keyword struct) can also be treated as buffer. Therefore, argument types that are data structures are among others tested with the feedback type to determine the size of the underlying buffer. Usually, the feedback cycle starts with buffers of size 1 byte. However, data structures have a fixed size. It would be reasonable that most often instances of this data structure need to used as test values. Therefore, we start a feedback loop with the size of the data structure. In other words, the size of a data structure s is put into the data structure template to generate an adaptive feedback test type for s*.
5.3.5 Type Characteristics

Autocannon uses static analysis on public source code to extract the type characteristics of a function’s argument types. With the help of this characteristics it maps argument types to Autocannon’s test types. The type characteristics are:

- **pointer?** True if the argument type is a pointer value.
- **sizeof** Size of the argument type in bytes.
- **converts_to_int?** True if argument type can be interpreted as an integer.
- **signed?** True if argument type is signed.
- **content_size** Size of the dereferenced type in bytes (for pointers, only).
- **ishandle?** True if at least one constructor function and exactly one destructor function for this type exist.

Pointers to bytes are treated as strings. They are mapped to the `meta_string` type. For other pointers, a data structure type template is instantiated and with the help of a generated meta type combined with `meta_pointer`. If the type is not a pointer, it is mapped to one of Ballista’s integer types depending on its size and if it is signed. An exception are signed types of size 2 and 4. They are assigned to `meta_short` and `meta_integer`, respectively. If the argument type is also a handle, the handle type template is instantiated and combined with the test type computed up to now.

For example, type `size_t` is not a pointer, has a “sizeof” 32 Bit (on x86), “converts_to_int”, is not signed and is not a handle. Therefore, `size_t` is mapped to `meta_integer`. An other argument type we already introduced is `apr_socket_t*`. It is a pointer, has a “sizeof” 32 Bit (on x86), “converts_to_int”, is not signed (actually this property will not be queried for pointers), has a “content_size” of 44 bytes (on x86) and is a handle. For `apr_socket_t*` Autocannon instantiates a feedback type with “content_size” and a handle type. Both types are combined with `meta_pointer` by creating a new meta type. The test values of this new meta type are the test values for argument type `apr_socket_t*`.

5.3.6 Reducing the Number of Test Cases

Ballista’s produce up to 1000 test values (including parents) for a test type. However, the average is per test type is much lower. Both the size of the set of input vectors and, hence, the number of test cases increase exponentially with the number of function arguments. Therefore, it might be infeasible to test all input vectors for functions with a large number of arguments. Ballista introduced an upper bound \( u \) on the number of tested input vectors. If the set of all input vectors is larger than \( u \), then a uniform distributed sample set of size \( u \) of the input vectors is computed.

Autocannon uses typically larger input vector sets than Ballista because of the use of meta types. The average size of an input vector set in our evaluation is larger than
5.3 Test Values

3.45 \cdot 10^{16} \text{ and the maximum is } 5.9 \cdot 10^{18}. \text{ Ballista iterates over the set of input vectors and decides per input vector with the help of a random number if this vector is tested. But it consumes too much time to do } 3.45 \cdot 10^{16} \text{ iterations. Therefore, we introduced a new sampling algorithm. The algorithm computes the sample by choosing randomly } u \text{ input vectors. In contrast to Ballista’s sampling, our new sampling approach has the disadvantage of possibly testing an input vector twice, But our algorithm is fast. The larger the set of input vectors the smaller the probability of testing an input vector twice. To combine both advantages, we introduced a threshold level. If the size of the set of input vectors is smaller than this threshold, Autocannon uses Ballista’s algorithm. Otherwise, our new sampling is used. Currently our threshold level is 100,000 input vectors.}

A small number of Ballista’s test values are not computable (e.g., a file handle with write permissions on a read only file). Thus, Ballista accepts that some input vectors cannot be tested. However, Ballista’s sampling algorithm does not take into account the computability of a test value. Whereas, Autocannon’s sampling skips uncomputable input vectors. Hence, Autocannon can test more input vectors than Ballista. In that sense, Autocannon has a better test coverage than Ballista, because it reduces the size of the set of input vectors by removing not computable test values. Test coverage is the number of performed tests over the size of the input vectors set.

Another way to reduce the size of the set of input vectors is to exclude special types that will not contribute to the results from the generic meta types. For instance, a function operating on strings but not on files does not need to be tested with file names. We perform static analysis using the LLVM framework \cite{65} to determine which special test types can safely be removed. The LLVM byte code of the library might be provided by its vendor (if the library should run on LLVM) or can be compiled from the library’s private source code.

As mentioned above, Ballista contains a predefined mapping from POSIX functions to their specialized Ballista test types. Our assumption is that we do not need to test a function \( f \) on special test types designed for a POSIX function, if this POSIX function is not called by \( f \). But if a function \( f \) calls some POSIX functions, we test \( f \) also with the special test types of the POSIX functions because \( f \) might pass some argument values directly to this POSIX functions. Of course, this assumption is not restricted to POSIX functions. In general, we can apply our reduction approach as long as a mapping exists from functions to special test types exclusively provided for these functions.

For each function to test we compute the transitive set of called POSIX functions. To simplify our analysis, we exclude functions, if they might perform indirect calls using function pointers. The set of called functions is compared with the list of POSIX functions specified by Ballista. We only include special test types that are related to the POSIX functions called by the analysed function into the set of tested types. To return to the previous example: a function \( f (\text{char*}) \) that transitively calls no other function than printf is not tested on file names because printf expects only format strings. To avoid exclusion of exceptional values, we only exclude types that are leafs in Ballista’s original type tree that is a subgraph of Autocannon’s type graph. Therefore, \( f \) is also tested on plain string and pointer values.
5 Robustness and Security Hardening of COTS Software Libraries

5.3.7 Other Sources of Test Values

Because each new input vector adds just another row to the truth table, the table can be extended by other inputs. One can do other kinds of fault injection, like the ones we use to evaluate Autocannon in Section 5.3.6. An additional source are traces of runs of some application utilizing the library to protect. The advantage of these traces is that they contain (mostly) input vectors for which the library function behaves robustly. Thus, these traces avoid false positives (see Section 5.5.2). The disadvantage is that one has to set up the application to do the tracing.

Currently, we do not use any additional sources of test values to keep the implementation effort at a reasonable level. However, we use bit flips and application traces for evaluating the generated hypotheses (see Section 5.6). It would be reasonable to use these techniques to help in the generation of the hypotheses. But then our evaluation would be less meaningful.

5.4 Checks

Checks are used in the analysis stage to build the truth table that is used to derive the protection hypothesis. While input vectors represent the rows of the truth table, checks represent the table’s columns. Checks are used to classify the input vectors in the truth table. Because they become part of the protection hypothesis, they are also used at runtime in the wrapper to check the current input vector.

Definition 5.1. A check is a predicate over one or more argument values.

For instance, for function foo(void* p), the check null?(p) computes whether p is or is not a NULL pointer. This example illustrates that checks must be instantiated for specific arguments of a function.

While checks are instantiated for given argument types, checks and test types are not coupled. In this way, our approach is easily extensible. One does not need to consider the test type system when adding new checks and vice versa. Checks may employ more than one argument of a function. Checks employing only one argument are called basic checks and other checks are compound checks. We have a set of check templates that are instantiated depending on the type characteristics of the functions arguments. Hence, the applicable checks depend on the argument types of a function. For example, some checks are only applicable on pointers, e.g., if a pointer is NULL, or even only to pointers to characters (char*). Checks that apply on integers (such as if an integer is negative) will not be instanciated for a pointer type. We use the type characteristics from Section 5.3.5 to instantiate for each argument type all applicable checks. Some check templates are additionally parameterized. For instance, for each handle type we parameterize a more general check template to derive a specialized check template for this handle type.

On one hand, we only add instantiated checks to the truth table. Consider function void foo(void*, void*). Function foo has no integer arguments. Hence, no integer checks will instantiated or added to the truth table of foo. On the other hand, checks
5.4 Checks

<table>
<thead>
<tr>
<th>Check</th>
<th>Type</th>
<th>Description (true, if ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>isnull</td>
<td>any pointer</td>
<td>pointer value is NULL</td>
</tr>
<tr>
<td>isstring</td>
<td>char*</td>
<td>pointer points to zero terminated buffer</td>
</tr>
<tr>
<td>isreadable</td>
<td>any pointer</td>
<td>buffer pointed to is readable</td>
</tr>
<tr>
<td>iswritable</td>
<td>any pointer</td>
<td>buffer pointed to is writable</td>
</tr>
<tr>
<td>onstack</td>
<td>any pointer</td>
<td>pointer points to stack</td>
</tr>
<tr>
<td>onheap</td>
<td>any pointer</td>
<td>pointer points to heap</td>
</tr>
<tr>
<td>isbufferstart</td>
<td>any pointer</td>
<td>pointer points to start of an allocated buffer on the heap</td>
</tr>
<tr>
<td>iszero</td>
<td>any integer</td>
<td>integer = 0</td>
</tr>
<tr>
<td>ispositive</td>
<td>any integer</td>
<td>integer &gt; 0</td>
</tr>
<tr>
<td>isnegative</td>
<td>any integer</td>
<td>integer &lt; 0</td>
</tr>
<tr>
<td>isfilename</td>
<td>char*</td>
<td>string represents an existing file name</td>
</tr>
<tr>
<td>isdirectory</td>
<td>char*</td>
<td>string represents an existing directory name</td>
</tr>
<tr>
<td>isformatstring</td>
<td>char*</td>
<td>string contains %n</td>
</tr>
</tbody>
</table>

Table 5.3: Basic checks of Autocannon.

can be instantiated multiple times. Since function foo has two pointer arguments, each pointer check will be instantiated twice, once for each argument.

All checks are tested on the input vector directly before testing the function. This introduces an additional testing overhead. The reason is that checks have to be independent of the test values to facilitate the extensibility. For example, one can in this way add new test cases that might satisfy checks that one has not been aware of. Technically, it would be possible to precalculate the results of all applicable checks on known test values off-line. In this way we could save some time otherwise needed in the fault injection stage (dynamic analysis stage). For instance, the test type meta_integer contains only a static set of integer values. However, we have not implemented this precomputation to simplify our implementation. Beside that, it would be difficult to apply compound checks off-line. This makes the on-line evaluation, i.e., as part of the fault injection, very convenient because Autocannon might generate new test types depending on its static analysis.

5.4.1 Check Templates

Our current implementation includes a set of predefined check templates. We use “check” to refer to check template whenever it is obvious that we mean a check template. Checks test general properties of the argument values independent of the concrete function’s argument types. We start with presenting the basic checks before discussing our compound checks.
5 Robustness and Security Hardening of COTS Software Libraries

Table 5.4: Compound checks of Autocannon. Argument p have a pointer type, arguments with prefix a have an integer type, and arguments with prefix s are of type char*.

<table>
<thead>
<tr>
<th>Check</th>
<th>Buffer pointer to by ( p ) has at-least...</th>
</tr>
</thead>
<tbody>
<tr>
<td>bufReadable(p, a)</td>
<td>a readable bytes</td>
</tr>
<tr>
<td>bufWritable(p, a)</td>
<td>a writable bytes</td>
</tr>
<tr>
<td>bufStrReadable(p, s)</td>
<td>strlen(s) + 1 readable bytes</td>
</tr>
<tr>
<td>bufStrWritable(p, s)</td>
<td>strlen(s) + 1 writable bytes</td>
</tr>
<tr>
<td>bufReadable2(p, a1, a2)</td>
<td>a1 * a2 readable bytes</td>
</tr>
<tr>
<td>bufWritable2(p, a1, a2)</td>
<td>a1 * a2 writable bytes</td>
</tr>
<tr>
<td>bufStrReadable2(p, s1, s2)</td>
<td>strlen(s1) + strlen(a2) + 1 readable bytes</td>
</tr>
<tr>
<td>bufStrWritable2(p, s1, s2)</td>
<td>strlen(s1) + strlen(s2) + 1 writable bytes</td>
</tr>
<tr>
<td>bufStrReadable3(p, s, a)</td>
<td>strlen(s1) + 1 + a readable bytes</td>
</tr>
<tr>
<td>bufStrWritable3(p, s, a)</td>
<td>strlen(s1) + 1 + a writable bytes</td>
</tr>
</tbody>
</table>

Basic Checks

Table 5.3 shows all basic checks of Autocannon. Column Type refers to the argument types for which the corresponding check can be instantiated. Any pointer gets instantiated for any type where the type characteristics pointer? is true. Any integer gets instantiated for any type where converts_to_int? is true. For char* pointer? must be true and content_size must be 1. Column Description states when a check evaluates to true.

Compound Checks

Compound checks test the relations between more than one argument value. The strcpy example from Section 5.2 contains the compound check \( \text{buf.write?}(\text{dest}, \text{strlen(src)} + 1) \). It relates the size of a buffer (pointed to by \( \text{dest} \)) and the length of string (\( \text{src} \)). This check evaluates to true, if \( \text{dest} \) points to a buffer where at least \( \text{strlen(src)} + 1 \) bytes are writable. We derive all compound checks from existing function specifications (e.g., the C standard library specification).

Table 5.4 gives an overview of our compound checks. All compound checks are relate the size of an argument, which is a pointer to a buffer, with one or two other function arguments. It is checked, if a buffers has at least the size of another string argument or an integer value. Additionally, the product of two integer arguments, the sum of two string lengths and the sum of a string length or an integer to the buffer size. The check \( \text{buf.write?}(\text{dest}, \text{strlen(src)} + 1) \) introduced above is implemented by bufStrWritable from Table 5.4.

Compound checks are instantiated for any possible argument combination of a function. For example, for function \( \text{char* strcpy(char* dest, char* src)} \) the following compound checks are instantiated by Autocannon:

- \( \text{bufStrReadable(dest, src)} \)
5.4 Checks

- `bufStrWritable(dest, src)`
- `bufStrReadable(src, dest)`
- `bufStrWritable(src, dest)`

Of course, Autocannon will also instantiate basic checks for `strcpy` after Table 5.3.

### 5.4.2 Parameterized Check Templates

Like we have type templates for generating specialized test values, we need specialized checks for testing for these test values. Parameterized check templates are checks used to test properties that cannot be predefined because these properties depend on the function’s argument types or the function’s semantic.

We have four kinds of parameterized checks:

- The feedback buffer check gets the parameter from the feedback experiments of the fault injection phase: It checks if a given buffer is at least as large as the smallest buffer for which the function did not crash for values of the feedback type. Consider, for example, function `void foo(char* p)`. Consider furthermore, in the fault injection with the feedback type for argument `p` Autocannon derived that `p` must point to a readable buffer of at least `n` bytes. Hence, Autocannon instantiates the check `bufReadable_n_bytes(p)` that checks if `p` points to a buffer of at least `n` readable bytes.

- The second check is similar to the feedback check, but it relies on static analysis only. It checks, if the buffer pointed to by a pointer is at least as large as the data structure of the corresponding argument type. For instance, for function `void bar(struct foo* p)` Autocannon instantiates `bufReadable_n_bytes(p)` and `bufWritable_n_bytes(p)` where `n` is `sizeof(struct foo)`.

- Another parameterized check is also derive with static analysis. If an argument type is an `enum`, then the `enum` values are extracted from the public sources. The check tests, if an argument value is within the set of possible `enum` values. Consider function `char foobar(enum Color c)`. Autocannon instantiates a check `isEnum_Color(c)` that checks if `c` has valid value of type `Color`.

- The counterparts of the ADT test types are ADT checks. For each argument that has a handle type a parameterized ADT check is instantiated. For function `int fileno(FILE* f)` Autocannon instantiates `isHandle_FILE(f)` that checks if `f` is an instance of ADT `FILE`.

At runtime an ADT check maintains a set of all valid handles for the corresponding ADT. For each ADT there is at most one check that might be instantiated multiple times (i.e., for different functions). Each parameterized ADT check has its own handle set. Therefore, each execution of a constructor function is intercepted. After running the original constructor, the returned handle value is put into the set. Every
destructor execution is also intercepted and the passed handle value is removed from
the handle set. The check for the handle queries the corresponding handle set. If
the value is in it, the check evaluates to true, otherwise to false.

5.5 Protection Hypotheses

The analysis stage yields one protection hypothesis per library function. A protection
hypothesis is evaluated on a given input vector.

Definition 5.2. An input vector is safe if the library function evaluated on it will behave
robust and secure. Otherwise, an input vector is unsafe.

Note that our definition of a safe input vector is similar to the definition of a safe
function call (see Chapter 4.4.2).

Definition 5.3. A protection hypothesis of a library function \( f \) is a function that maps
the set of input vectors of \( f \) to \{true, false\}. If it evaluates to true, the given input vector
is considered to be safe. Otherwise, the hypothesis evaluates to false.

To enforce that a library function is only executed on safe input vectors, Autocan-
non generates a protection wrapper. The protection wrapper is inserted between an
application and its dynamic linked libraries. Our current implementation achieves this
by utilizing the pre-loading feature of the dynamic linker [33] (similar to AutoPatch in
Chapter 4.6). We could also insert the wrapper by instrumenting the libraries or the
application.

The first part of this section shows how to derive protection hypotheses and the second
part discusses possible failures of the protection hypotheses. We review two kinds of
failure:

Definition 5.4. A false positive happens if a hypothesis rejects a safe input vector. A
false negative happens if a hypothesis accepts an unsafe input vector.

5.5.1 Minimizing the Truth Table

The protection hypothesis for a function \( f \) is derived from the truth table built while
testing \( f \). The truth table is not yet a boolean function. It might contain redundant
and even contradicting rows. Also some rows might be missing. A row \( r \) is redundant
if there is another row \( p \) with \( p = r \). Redundant rows are the result of input vectors
that are indistinguishable from each other by our checks and the functions behavior is
the same for both input vectors. Redundant rows have two sources: an under fitting set
of checks or an over fitting set of input vectors. Contradicting rows are rows that are
classified equally by the checks but for which the function behaves differently (i.e., one
crashes the other does not). They indicate that the set of checks is too small. Hence, this
can be countered by adding new checks. Note however that removing input vectors from
testing reduces the probability to find contradictions too. Because the lower the number
Table 5.5: Example truth table with redundant, contradicting and missing rows.

<table>
<thead>
<tr>
<th>Row no.</th>
<th>check_1</th>
<th>check_2</th>
<th>robust?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

5.5.2 Discussion

As described above a source of false negatives and false positives is if too few input vectors are tested. Simply adding new test types may not help: The resulting set of input vectors might become too large to test it within a feasible time. Counter-intuitively, we have found that reducing the set of input vectors is a good approach if the set of input vectors is too large. It increases the coverage provided by the tested sample of input vectors. Of course the reduction algorithm should only remove input vectors that do not contribute to the protection hypothesis. In Section 5.3.6 we show one example for a reduction algorithm using static analysis.

If the protection wrapper evaluates a protection hypothesis on an input vector to false, the original function is not called. Instead the wrapper passes the control back to the caller. Therefore, a value has to be returned. Our current implementation returns a
5 Robustness and Security Hardening of COTS Software Libraries

predefined error value depending on the return type of the wrapped function \[107\]. The predefined value is not guaranteed to be an legitimate return value of the original function. Instead, the AutoPatch approach described in Section 4.3 can be used to determine error return values that are guaranteed to be legitimate return values of the original function. However, we have not added this approach to keep the implementation effort for Autocannon at a reasonable level.

In spite of every effort to reduce false positives in the analysis stage, it might be possible that some generated hypotheses are too restrictive and produce false positives. Such an hypothesis will evaluate to false for valid input vectors when used to protect the application. To finally detect such hypotheses before generating the wrapper, we generate an intermediate wrapper. Instead of enforcing the protection hypotheses, it only logs whenever a hypothesis is evaluated to false. This wrapper is used to gather false positives in one or more test runs with real applications. Hypotheses that are found to produce false positives are excluded from the protection wrapper. In that way it might be possible that some functions are not protected at all. But our evaluation shows that only a minority of hypotheses had to be excluded from the protection wrapper, i.e., our false positive rate was already very low.

Because our approach does not guarantee the absence of false positives, we discuss now how to handle them. Note that in case of a protection hypothesis being evaluated to false, the protection wrapper returns an error code instead of calling the library function. First, if the application is robust enough, it might perform some graceful degradation in the presence of a false positive. Thus, the application will provide a reduced functionality. Second, if an application performs retries of failed library calls and if we are mainly worried about transient errors, we could use a similar approach as \[88\]: On re-execution of a failed function, we would allow its execution and if it passes, we white list arguments with the same check vector.

The false positives are a sign that we do not have enough checks to distinguish between unrobust and robust input vectors. But our approach allows us to add more checks in the future without any redesign. The false negative rate might also benefit from new checks. For the same reason it is useful to add more test types. But as long as a function is only tested on a small sample of all test types we need better strategies to choose the “right” input vectors for fault injection.

5.6 Evaluation

We have evaluated our approach by hardening the Apache Portable Runtime library (APR) \[2\]. Among the applications that use this library is the web server Apache. We have tested our protection wrapper for the APR by executing it with Apache. To reduce the overhead, we only have hardened functions that are called by our executions of Apache. These are 148 APR functions including some without arguments. Functions without arguments where not tested and no protection hypotheses are created for these function. Our test system, Autocannon, was able to perform about 1000 tests per minute on our computers. That means we can add about 1000 rows to the truth tables per minute.
5.6 Evaluation

Because functions can be tested independently from each other, it is possible to parallelize the analysis stage. We have done all experiments on two virtual machines with the same configuration (running on an Athlon 64 3200+ with 1 GByte of RAM and an Intel Core Duo with 2 GBytes of RAM, respectively, both with Ubuntu 6.06).

First, we discuss the results of Autocannon’s fault injection tests. We will focus on test coverage and the results of the dependability benchmark. To check the correctness of hypotheses, we run Apache with the APR. To test the effectiveness of the generated hypotheses we injected bit-flips into Apache’s execution.

5.6.1 Coverage

Some of the hardened functions have more than 3 arguments. Because the set of input vectors grows exponentially with the number of arguments, such functions can only be dynamically analyzed to a very small extent. We tested all functions on at most 10,000 test cases and less if the set of possible input vectors was smaller. This number of test cases was chosen to perform our evaluation in a feasible time. In Figure 5.5 we depict the coverage for four different test configurations. We tested the combinations with and without test types for handles and with and without reduction of test types by excluding special test types via static analysis. The functions are grouped into 15 coverage classes. The coverage of a function is the number of tested input vectors over the size of the set of all input vectors of the test system for this functions. The class of a function \( f \) is derived from the order of magnitude of \( f \)’s test coverage. The functions of class 0 have 100%
5 Robustness and Security Hardening of COTS Software Libraries

<table>
<thead>
<tr>
<th>Test config with Reduction</th>
<th>with Handle</th>
<th>Avg. Coverage</th>
<th>Avg. Incomplete Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>0.442</td>
<td>0.071</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>0.276</td>
<td>0.023</td>
</tr>
</tbody>
</table>

Table 5.6: The average coverage per test configuration, and the average incomplete coverage that is the average coverage over all functions that have a coverage < 1.

Figure 5.6: Comparing number of crashed and robust test cases.

coverage, functions of class 1 have 10%, functions of class 2 have 1% coverage, and so on. The Y-axis depicts the number of functions per class.

Adding the handle test types (see Section 5.3.4) has no visible impact. The average coverage of the two test configurations with handles differs slightly from the average coverage for test configurations without handles as shown in Table 5.6.

The impact of excluding some test cases via static analysis (see Section 5.3.6) is visible. In the two configurations, where the number of test cases is reduced with static analysis, more functions have a higher coverage than in the two configurations without reduction. Some functions do not benefit from the reduction because they perform indirect function calls via function pointers. These functions were excluded from the static analysis (see Section 5.3.6). We were able to reduce the number of test cases for 76 of the analysed 148 functions.

5.6.2 Autocannon as Dependability Benchmark

Autocannon is not only useful to generate protection hypotheses. The analysis stage of Autocannon alone is a flexible dependability benchmark for arbitrary C libraries. Figure 5.6 gives a summary of the benchmark results for the APR. The figure only covers the
functions that we have tested. More than half of the test cases resulted in robust behavior. The gap between the number of robust and unrobust test cases is about 70,000 function calls. Figure 5.7 shows how the robustness is distributed over the tested functions. The majority of the functions can be put in one of two sets:

- 67 functions crashed in at least 90% of the test cases.
- Whereas, 66 functions crashed in at most 10% of the test cases.

Between these two classes are 40 functions that crashed for more than 10% but for less than 90% of their test cases. Functions without arguments are not included in the Tests.

### 5.6.3 Protection Hypotheses

We evaluated the correctness of the generated protection hypotheses in two directions:

- Do we produce false positives?
- How good do we prevent crashes?

We generated two wrappers:

- $W$ was generated from fault injection experiments with lower test coverage, and
- $W_{SA}$ was generated from fault injection experiments with higher test coverage using static analysis.
We switched the handle test types on for both wrappers. In practice a developer can manually correct any protection hypotheses. However, for this evaluation we did not do so. Table 5.7 shows that W protects 77 functions whereas W_SA protects 95 functions. For all other functions no hypotheses could be generated.

We first executed Apache with the protection wrapper and logged all failed hypotheses. We assume that Apache does call all APR functions with valid arguments as long as no faults are injected by a third party. Thus, any failed hypothesis is a false positive. The results are shown in the first two data rows of Table 5.7 in row “FP normal Run”. W produced no false positives. W_SA had 10 functions leading to false positives (6.17 % of all function calls). 3 of these functions were also wrapped by W, but with different hypotheses.

Additionally, we implemented a bit-flip benchmark with fault injection to test our protection in the presence of failures. Therefore, we run Apache with our protection wrapper and injected bit-flips into this execution. We injected probabilistically bit-flips into argument values of the protected functions. Whenever a bit-flip was injected when calling a function \( f \), we compared the result of \( f \)'s hypothesis and the behavior of \( f \). Not all bit-flips resulted in a crash of \( f \). Apache was restarted after the bit-flip to prevent further propagation of the fault. The results show a trade-off between W and W_SA (see Table 5.7 last two rows). A successfully predicted crash is prevented by not calling the crashing function. Thus, W prevents 56.81% of all crashes and W_SA prevents 51.39% of all crashes. However, W has also more false positives than W_SA.

### 5.7 Conclusion

We have presented a flexible approach to hardening arbitrary libraries for robustness and security. Our contributions are:

- A new dependability benchmark that can measure the robustness of arbitrary library utilizing static analysis, and

- A truth table based approach to derive protection hypotheses from the benchmark’s results.

Therefore, the benchmark’s test data is classified using checks. The difference to previous work, such as HEALERS, is that our approach is easily extensible. One can add new
checks and test types for the dependability benchmarks as needed.

Our evaluation shows that Autocannon as dependability benchmark can find bugs in real world software (Thesis 7: “Fault injection finds failures even in mature software.”). The protection wrapper implements the filter patch pattern. Unsafe input values to a function are filtered. Our protection hypotheses were able to predict up to 56.85 % of the crashes in our evaluation (Thesis 6: “Automatic bug removal based on patch patterns can decrease the number of failures.” and Thesis 9: “Automatic filtering can mask a high percentage of failures injected by bit-flip faults.”). The drawback is that our hypotheses misclassify a low number of robust argument values as unrobust. But we believe that we can overcome this issue by adding more appropriate checks and test types.


6 Conclusion

In this thesis we introduce automatic hardening as a tool to tolerate software bugs at the end-user site. Automatic hardening techniques use either error tolerance or bug removal.

With SwitchBlade (Chapter 2), we contribute to error tolerance. SwitchBlade uses speculation in a novel way to detect malicious 3rd-party control flow manipulations (Thesis 1: “Speculation can be used to reduce the perceived performance overhead of error detection mechanisms.”). We show that fine-grained system call interposition techniques alone cannot be used to reliably detect control flow manipulations (Thesis 2: “The system call model of an application depends on the environment.”). But in combination with taint analysis fine-grained system call interposition provides good means to detect control flow manipulation based attacks (Thesis 3: “System call model enforcement combined with taint analysis can detect control flow manipulation with in average low performance overhead, low false positive, and low false negative rate.”). We use SwitchBlade to protect the web-server Apache. By small changes to Apache’s code base, SwitchBlade is not only able to detect control flow manipulation in a 3rd-party plug-in, but also tolerates these attacks by dropping the current connection. For applications that are not able to tolerate detected control flow manipulations by themselves, SwitchBlade can be combined with Rx [94] or ASSURE [108].

ParExC, presented in Chapter 3, contributes to the state of the art of parallelized runtime checking (Thesis 4: “Parallelization of runtime checks can (partly) mitigate the performance costs of expensive runtime checks.”). ParExC parallelizes runtime checks speculatively (Thesis 1). Our measurements indicate that parallel runtime checking can compete with and sometimes even outperform checking of already parallelized applications (Thesis 5: “Parallelizing runtime error checks themselves can scale better than parallelizing an application with runtime checks.”). ParExC itself is an error detection approach. To achieve error toleration it needs to be combined with approaches such as Rx and ASSURE. However, these approaches make use of check-pointing and replay. Both techniques are already part of the ParExC framework. Thus, we believe a combination of ParExC with the orthogonal approaches of Rx and ASSURE is feasible.

In Chapters 4 and 5, we contribute to the state of the art of bug removal. In both chapters we use fault injection to find bugs in real world code (Thesis 7: “Fault injection finds failures even in mature software.”). We introduce patch pattern to remove some of the found bugs (Thesis 6: “Automatic bug removal based on patch patterns can decrease the number of failures.”). However, both chapters are orthogonal to each other. AutoPatch (Chapter 4) uses a technique called error mapping to remove bugs in error handling code (Thesis 8: “Automatic error mapping can mask failures.”). Error mapping is similar to the error virtualization used by ASSURE. If 3rd-party libraries do not check their input carefully, they can crash the whole application. Autocannon (Chapter 5) uses filtering
to remove “missing input checking” bugs in 3rd-party libraries (Thesis 3: “Automatic filtering can mask a high percentage of failures injected by bit-flip faults.”).

Future research in automatic hardening should try to target distributed applications (for instance streaming application). While AutoPatch and Autocannon are generic enough to be applied out of the box to distributed applications, SwitchBlade and ParExC are mainly focused on protecting applications running on a single node. Distributed applications introduce new challenges (for instance partial failures and partitioning), but also new opportunities (such as node replication and distributed speculation).

6.1 Publications

Chapters 2 to 5 are partly based on published papers. In this section we want to clarify the mapping between this thesis and those papers.

Chapter 2 is an extended version of


This paper was republished as


ParExC’s StackLifter (Section 3.4) will appear in


The Speculative Variables (Section 3.5) were introduced in


The comparison of ParExC with checking applications parallelized by Software Transactional Memory (Section 3.6.4) is published as

- Martin Süßkraut, Stefan Weigert, Martin Nowack, Diogo Becker de Brum and Christof Fetzer: *Parallelize the Runtime Checks – Not the Application* at the Workshop on ExploitingConcurrency Efficiently and Correctly – (EC)$^2$ (CAV), 2009 [121].

The Chapter 4 about AutoPatch is widely based on


Section 4.3 about Learning Library Level Error Values was publication as

- Martin Süßkraut and Christof Fetzer: *Learning Library-Level Error Return Values from Syscall Error Injection* in the proceedings of the Sixth European Dependable Computing Conference (EDCC), 2006 [118].
Section 4.5 about Fast Fault Injections with the Help of Virtual Machines was published as


Chapter 5 about Autocannon is an edited and extended version of

References


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES

derivation of application-specific error detectors and their implementation in hardware.
In Proceedings of the Sixth European Dependable Computing Conference
(EDCC 2006), October 2006.

[89] David A. Patterson. Recovery oriented computing: A new research agenda for a

[90] D. S. Peterson, M. Bishop, and R. Pandey. A flexible containment mechanism for
executing untrusted code. In Proceedings of the 11th USENIX Security Symposium,

[91] Christopher J. F. Pickett and Clark Verbrugge. Return value prediction in a Java
virtual machine. In Proceedings of the Second Value-Prediction and Value-Based

[92] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fin-
gerprinting zero-day attacks. In Proc. ACM SIGOPS EUROSYS, Leuven, Belgium,
April 2006.

[93] Niels Provos. Improving host security with system call policies. In 12th USENIX

[94] Feng Qin, Joseph Tucek, Yuanyuan Zhou, and Jagadeesan Sundaresan. Rx: Treat-
ing bugs as allergies—a safe method to survive software failures. ACM Trans.

[95] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng
Wu. Lift: A low-overhead practical information flow tracking system for detecting
security attacks. In MICRO 39: Proceedings of the 39th Annual IEEE/ACM Inter-
national Symposium on Microarchitecture, pages 135–148, Washington, DC, USA,

[96] 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 Operating Systems Design & Implementation, pages 21–21, Berkeley, CA, USA,

[97] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran,
Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic informa-
tion flow tracking. In SPAA ’08: Proceedings of the twentieth annual symposium
on Parallelism in algorithms and architectures, pages 35–45, New York, NY, USA,
2008. ACM.

In NDSS. The Internet Society, 2004.
REFERENCES


REFERENCES


REFERENCES


REFERENCES


List of Figures

1.1 Contributions of this thesis associated to automatic hardening . . . . . . . 4

2.1 SwitchBlade architecture: in normal mode, all system calls are checked against a system call model. Violations result in a switch to taint mode, in which the last requests are replayed using a fine-granular data flow and control flow analysis. To facilitate replays, all network traffic is routed through a proxy. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.2 Execution of an application under SwitchBlade. Violations of the model result in the re-execution of the affected request in taint mode. If the violation was caused by a false alarm, the model is updated appropriately. 16

2.3 Syscall Model of Arithmetic Unixbench Benchmark . . . . . . . . . . . . . . 17

2.4 Programs might not always close handles. The system call model can sometimes remove values from variable sets even if they are not freed by the program. In this case, the file descriptor is removed from set v20 on the last write call . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

2.5 Extract from system call model for Apache worker processes. . . . . . . . 21

2.6 Model size of grep grows when we inject errors in the form of returning error codes for random system calls during tracing. The number of states grows from 36 to 79 and the number of transitions from 46 to 129 . . . . . . . 23

2.7 Model sizes for Apache when learned with different client programs . . . . 24

2.8 Overlap of the Apache system call models generated with 5 different client programs: left graph shows overlap in nodes and the right the overlap in edges . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

2.9 Part of the system call model for Apache. Transitions and edges missing in the Apache model generated with wget (dotted lines) in comparison to the unified Apache model . . . . . . . . . . . . . . . . . . . . . . 25

2.10 Overhead of SwitchBlade’s normal mode (i.e., system call enforcement) and executing the same requests in SwitchBlade’s taint mode compared to a native execution without SwitchBlade. We measured the average connection time as seen by a client connecting to the proxy for different kinds of content . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

2.11 Evolution of the Apache system call model for a sequence of 200 requests . . 34

2.12 System call model of ghttpd including model of the exploit (bold) . . . . 35

2.13 Change of the size of the models when changing from a small input file to a larger input file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
LIST OF FIGURES

2.14 Performance overhead of system call model enforcement compared to taint analysis. We expect that the applications run most of the time in model enforcement mode. .................................................. 37

2.15 Size of Apache model depending on the length of the backtrace. ........... 38

2.16 Growing size of the system call model of Vim. Each step contains an action like saving the file, calling external applications, and using the online help. 39

3.1 Our parallelization approach executes a fast variant on one core. The execution of the fast variant is partitioned into epochs. Each epoch is re-executed with a slow variant with more functionality. The re-execution happens in parallel to the fast variant on multiple cores. ......................... 46

3.2 The ParExC work-flow: The StackLifter generates the two initial code bases for the fast and the slow variant. Both variants can be instrumented to remove or add functionality, respectively. Both variants are linked together with a generated framework code that manages the switch from the fast variant to slow variant at epoch boundaries. .............................. 48

3.3 Overview of ParExC’s architecture at runtime. Deterministic replay and speculative system call execution is provided as part of the OS kernel. The checker runtime has access to speculative variables to manage the checker’s book-keeping state. Both, the fast variant and the slow variant (through the checker) have access to the epoch management. ......................... 49

3.4 Translating the call stack at runtime. ........................................... 56

3.5 Sample control flow graph. ......................................................... 66

3.6 The temporal order of A (malloc) and B (the write access) are different between predictor and executors. Checks use speculative variables to defer the check of the write. ......................................................... 67

3.7 Runtime measurements on an 8-core Intel Xeon Server with 16GB main memory. ................................................................. 74

3.8 Speedup of out-of-bounds (OOB) with ParExC relative to OOB without ParExC. ................................................................. 75

3.9 Runtime of sequential vs parallel DFI checker for two STAMP benchmarks. 76

3.10 Runtime and Speedup of parallelizing sanity checks and assertions in the BOOST words example. ........................................... 76

3.11 Throughput/runtime of the OOB checker with ParExC and Tanger/TinySTM. The error bars show the minimum and the maximum of our measurements. ................................................................. 77

3.12 Overhead of the system call speculation in the fast variant and the deterministic replay in the slow variant for Vacation. ................. 78

3.13 Overhead of the StackLifter instrumentation with three optimizations for Vacation. ................................................................. 79

3.14 Time to perform a stack lifting for different stack depths. ................... 80

4.1 Caller/Callee relationship together with our patch. .......................... 87

4.2 The data flow of analyzing an application for patch generation. ........... 88
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Two possibilities for error injection into the application: (1) the direct approach: from a library into the application, and (2) the indirect approach: from the operating system via the library into the application.</td>
</tr>
<tr>
<td>4.4</td>
<td>An error injection run with three error injection points with two, three, and two different error values to injection. The execution of the original application pauses while the fault injection runs.</td>
</tr>
<tr>
<td>4.5</td>
<td>Architecture of Fault Injection using Virtual Machines.</td>
</tr>
<tr>
<td>4.6</td>
<td>Shows the error mapping patch pattern on a call group: errors of unsafe function calls f1, f3, and f6 are mapped to safe call f4. Unsafe function call f5 can not be patched by this pattern.</td>
</tr>
<tr>
<td>4.7</td>
<td>Number of unsafe calls and unsafe calls with static arguments.</td>
</tr>
<tr>
<td>4.8</td>
<td>Robustness test of unpatched and unpatched applications with 4 different error injectors.</td>
</tr>
<tr>
<td>4.9</td>
<td>Runtimes relative to the unpatched application in %.</td>
</tr>
<tr>
<td>5.1</td>
<td>Workflow of Autocannon.</td>
</tr>
<tr>
<td>5.2</td>
<td>Autocannon injects faults into the input of a library.</td>
</tr>
<tr>
<td>5.3</td>
<td>A part of Ballista’s test type system with a meta type.</td>
</tr>
<tr>
<td>5.4</td>
<td>Feedback loop: For buffer test types the test system uses the addresses of illegal memory accesses to refine test values.</td>
</tr>
<tr>
<td>5.5</td>
<td>The coverage of executed test cases compared to the set of all possible input vectors in different test configurations.</td>
</tr>
<tr>
<td>5.6</td>
<td>Comparing number of crashed and robust test cases.</td>
</tr>
<tr>
<td>5.7</td>
<td>Variation in percentage of unrobust test cases per function.</td>
</tr>
</tbody>
</table>
## List of Tables

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Checks done by TaintCheck to detect and block exploits</td>
<td>28</td>
</tr>
<tr>
<td>2.2</td>
<td>Result of Wilander’s testbed [133] running natively and under SwitchBlade.</td>
<td>32</td>
</tr>
<tr>
<td>2.3</td>
<td>Results of testing SwitchBlade against real world exploits.</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>Table based approach to generate robustness and security checks. For a check the 0 means that a given check evaluates to false. A 1 means true.</td>
<td>121</td>
</tr>
<tr>
<td>5.2</td>
<td>Part of truth table for the function <code>strcpy(char* dest, const char* src)</code>.</td>
<td>122</td>
</tr>
<tr>
<td>5.3</td>
<td>Basic checks of Autocannon.</td>
<td>131</td>
</tr>
<tr>
<td>5.4</td>
<td>Compound checks of Autocannon. Argument <code>p</code> have a pointer type, arguments with prefix <code>a</code> have an integer type, and arguments with prefix <code>s</code> are of type <code>char*</code>.</td>
<td>132</td>
</tr>
<tr>
<td>5.5</td>
<td>Example truth table with redundant, contradicting and missing rows.</td>
<td>135</td>
</tr>
<tr>
<td>5.6</td>
<td>The average coverage per test configuration, and the average incomplete coverage that is the average coverage over all functions that have a coverage &lt; 1.</td>
<td>138</td>
</tr>
<tr>
<td>5.7</td>
<td>Correctness of our Approach with different protection wrappers: W is generated from test cases without reduction, and W_SA is generated from test cases with reduction.</td>
<td>140</td>
</tr>
</tbody>
</table>
Listings

3.1 API example and the problem of switching code bases ............................... 57
3.2 Input to the StackLifter ........................................................................... 59
3.3 StackLifter output: Code base of fast variant .......................................... 60
3.4 Framework code to combine both code bases at runtime ......................... 60
3.5 StackLifter output: Code base of slow variant ....................................... 61
3.6 Function call transformation for fast variant .......................................... 62
3.7 Indirect call transformation for slow variant .......................................... 62
3.8 Save register block ................................................................................ 63
3.9 New entry block .................................................................................... 63
3.10 Restore register block .......................................................................... 64
3.11 SSA-Example: before StackLifter ......................................................... 64
3.12 SSA-Example: after StackLifter – in the slow variant ......................... 64
3.13 Example of a phi node in LLVM ............................................................ 65
3.14 Source code for Figure 3.6 .................................................................. 67
3.15 Allocation and memory access in different epochs ............................... 71

4.1 The structure of a function wrapper part of the argument recording wrapper ................................................. 94
4.2 Structure of an error injection wrapper function ..................................... 95
4.3 Statements at the start of a function ....................................................... 98
4.4 Tail recursion in assembler generated by gcc ........................................ 99
4.5 Integrating Error Injection with a VM .................................................... 102
4.6 Pseudo code of a wrapper function of a generated patch ....................... 106
4.7 grep-5.2.1 src/search.c:152 .................................................................. 111
4.8 coreutils-2.5.1 lib/hash.c:537 ............................................................... 112
4.9 coreutils-2.5.1 lib/hash.c:578 ............................................................... 112
4.10 coreutils-2.5.1 lib/hash.c:591 .............................................................. 113
4.11 coreutils-2.5.1 lib/hash.c:602 .............................................................. 113
4.12 grep-5.2.1 src/dfa.c:3423 ...................................................................... 113
4.13 grep-5.2.1 src/dfa.c:3622 ...................................................................... 113
4.14 grep-5.2.1 src/dfa.c:3633 ...................................................................... 114
4.15 grep-5.2.1 src/dfa.c:3240 ...................................................................... 114