Safe Loading and Efficient Runtime Confinement: A Foundation for Secure Execution

Mathias J. Payer

Diss. ETH No. 20442

Safe Loading and Efficient Runtime Confinement: A Foundation for Secure Execution

A dissertation submitted to
ETH Zurich

for the degree of
Doctor of Sciences

presented by
Mathias J. Payer
Master of Science ETH in Computer Science, ETH Zurich
born April 29, 1981
citizen of the Principality of Liechtenstein

accepted on the recommendation of
Prof. Dr. Thomas R. Gross, examiner
Prof. Dr. Srdjan Capkun, co-examiner
Prof. Dr. Steve Hand, co-examiner

2012

Abstract

Protecting running applications is a hard problem. Many applications are written in a low-level language and are prone to exploits. Bugs can be used to exploit the application and to run malicious code. A rigorous code review is often not possible due to the size and the complexity of the applications. Even a detailed code review does not guarantee that all bugs in the application are found.

This thesis presents a model for the secure execution of untrusted code. The model assumes that the application code contains bugs but that the application is not malicious (i.e., malware). The application is safe if the model protects from all attack vectors through code-based or data-based exploits in the untrusted code. The model verifies all code prior to execution and ensures that no unchecked control flow transfers are possible. An important design decision is to use a dynamic approach for the implementation with minimal impact on the original applications. Binary only applications are executed without static recompilation or changes to the compiler toolchain (e.g., no recompilation is needed and features like dynamically loaded libraries, lazy binding, or hand written assembly code are still usable).

A dynamic, transparent sandbox in user-space loads and verifies code using binary translation. A secure loader starts the sandbox and bootstraps the application and all needed libraries in the sandbox. The sandbox checks the application code before it is executed and adds security guards during the translation. The combination of the secure loader and the sandbox protects from code-oriented exploits. System calls are redirected by the sandbox to a policy-based system call authorization layer that verifies every system call towards a policy. Every control flow transfer in the application code is verified using a dynamic control flow model. Control flow transfers to illegal locations or instructions that are not legal in the application stop the program. The combination of a system call policy and control flow integrity protects the application from code-based and data-based exploits.

A prototype implementation is used to evaluate the performance and effectiveness of the proposed model. We show that the overhead for our prototype implementation is low and that the model protects from all code-based exploits. The control flow model restricts the attack space for data-based attacks and restricts control flow transfers of the application to well-known and valid locations. The small and modular trusted computing base enables code reviews and allows additional security modules (e.g., a module that detects file-based race conditions).

Zusammenfassung

Der Schutz von laufenden Anwendungen ist ein komplexes Problem. Viele Anwendungen wurden in einer systemnahen Sprache geschrieben und sind fehleranfällig. Diese Programmierfehler können ausgenützt werden um eine Anwendung anzugreifen und schadhafte Befehle auszuführen. Eine detaillierte Überprüfung des Quellcodes ist, wegen der Grösse und Komplexität der Applikationen, oft nicht möglich. Ausserdem garantiert eine detaillierte Überprüfung nicht, dass alle Fehler in der Applikation gefunden wurden.

Diese Doktorarbeit präsentiert ein Modell für die sichere Ausführung von ungesichertem Programmcode. Das Modell trifft die Annahme, dass der Programmcode der Anwendung Fehler enthalten kann, die Anwendung selbst jedoch nicht bösartig ist (d.h. die Anwendung ist selbst keine Malware). Die Anwendung gilt als gesichert, falls das Modell vor allen Angriffsvektoren durch Code-basierte und Daten-basierte Angriffe im ungesicherten Programmcode schützt. Das Modell stellt sicher, dass jede Codesequenz geprüft wird, und garantiert, dass keine ungeprüfte Verzweigung möglich ist. Eine wichtige Designentscheidung ist es, einen dynamischen Ansatz für die Implementierung zu wählen, welcher nur minimalen Einfluss auf die Anwendung hat. Ausführbare Programme ohne Quellcode werden ohne statische Kompilierung oder Veränderungen des Kompilierers ausgeführt. Dieser Ansatz ermöglicht die Ausführung von geschützten Programmen ohne erneute Kompilierung und erlaubt Funktionen wie dynamische Bibliotheken, verzögertes Laden und handoptimierten Assemblercode.

Eine dynamische, transparente Sandbox im Userland lädt und verifiziert den Programmcode mittels binärer Übersetzung. Ein abgesichertes Programm startet die Sandbox und initialisiert die Anwendung und alle benötigten Bibliothekten innerhalb der Sandbox. Die Sandbox überprüft den Programmcode bevor er ausgeführt wird und fügt während der Übersetzung dynamische Sicherheitschecks hinzu. Die Kombination von sicherem Startprogramm und der Sandbox schützt vor allen Code-orientierten Angriffen. Systemaufrufe werden von der Sandbox zu einer regelbasierten Autorisierung für Systemaufrufe umgeleitet. Jede Verzweigung im Programmcode wird anhand eines Kontrollflussmodells verifiziert. Verzweigungen zu illegalen Zielen oder Instruktionen beenden die Anwendung. Die Kombination einer regelbasierten Autorisierung für Systemaufrufe mit den Kontrollflusstests schützt die Anwendung vor Code-basierten und Daten-basierten Angriffen.

Die Effektivität und Effizienz des Modells wird anhand einer Prototypenimplementierung evaluiert. Wir zeigen, dass der zusätzliche Berechnungsaufwand klein ist und dass das Modell vor allen Code-basierten Angriffen schützt. Das Kontrollflussmodell verringert die Angriffsmöglichkeiten von Daten-basierten Angriffen und beschränkt die möglichen Verzweigungen auf bekannte und gültige Ziele. Der kleine und modulare Prototyp ermöglicht eine detaillierte Überprüfung des Quellcodes und erlaubt ausserdem zusätzliche Sicherheitsmodule (z.B. ein Modul, welches zeitliche Angriffe auf das Dateisystem erkennt).

Acknowledgments

This thesis would not have been possible without the help and input of many people. First of all, I would like to thank my adviser Thomas R. Gross for the helpful discussions during the design of the secure execution model. His input and feedback on countless paper drafts and conference submissions is greatly appreciated. I would also like to thank my co-advisers Steven Hand and Srdjan Capkun for their feedback on this thesis.

I owe my deepest gratitude to my wife Anna Barbara Payer for the unlimited personal support I get from her and for her patience whenever I come home late due to some conference deadline. My parents Ursula Payer and Josef Payer and my sister Corina Payer have given me all the support and encouragement I could have hoped for.

Special thanks go to all the current and former group members: Albert Noll, Christoph Angerer, Christian Tuduce, Faheem Ullah, Florian Schneider, Michael Pradel, Mihai Cuibus, Niko Matsakis, Oliver Trachsel, Stephanie Balzer, Susanne Chech Previtali, Valeri Naoumov, Yang Su, and Zoltan Majo. Thank you for all these great discussions and the helpful feedback during our lunch breaks and over coffee.

I would also like to thank all the students that I was allowed to supervise during my time as a doctoral student: Antonio Barresi, Boris Bluntschli, Christian Oberholzer, Enrico Kravina, Georg Schätti, Gianmatteo Costanza, Jonas Pfefferle, Ken Lee, Marcel Wirth, Martin Bill, Noah Heusser, Olivier Saurer, Peter Suter, Philipp Wolfensperger, Stephan Classen, and Tobias Hartmann.

Last, but by no means least, I thank all my friends in Switzerland, Liechtenstein, Germany, Austria, the U.K., the U.S., and all over the world for their encouragement and support throughout my time as a doctoral student.

Contents

List of Figures

List of Tables

List of Listings

1

Introduction

Secure execution of applications in user-space is a hard problem. Several statistics show that the number of lines of code is proportional to the number of bugs in a program [PCC+96, Sch00, McC04]. Some bugs are security relevant. A bug is security relevant if it can be exploited to gain access to a system or to escalate privileges of an attacker (e.g., if the bug enables an attacker to redirect control flow to alternate code, or to inject new code into the process that is then executed).

In addition, the number of lines of code in applications usually increases from one release to another (e.g., bugs that are fixed, added functionality, and other changes). Code reviews can reduce the number of bugs and especially for critical sections it is important to review the code rigorously. Nevertheless, the *Trusted Computing Base (TCB)* [1] includes the complete code (i.e., the application and all used shared libraries) that is running inside a process plus the privileged kernel code. It is not practical to rigorously review or verify large code bases due to the state and complexity explosion.

The current security practice is reactive. Exploitable bugs are analyzed by the vendor after detection. The vendor releases a patch after the exploit is reproduced and a fix is constructed. This patch is then distributed to the clients. The clients then apply the patch after local testing. All running instances of the software are vulnerable until the patch is applied. Other related approaches prohibit specific patterns (e.g., system call combinations, control flow transfers, or input data) in the application. These patterns must be predefined and the detection mechanisms are limited, e.g., techniques like return-oriented programming [PB04, Sha07, Ner07], or jump-oriented programming [PB04, BJFL11] are hard to detect.

A limitation of current operating systems is that privileges are given based on a per-user basis. Every user can run multiple (different) applications. Every running application has the same privileges as the user. A specific application does not need all privileges of a specific user but rather only a subset. In general different applications need different subsets of privileges. For example, a web-browser should be able to access the Internet but should not be able to access private files, while a text editor should be able to open documents but should not be able to transfer data to a different host on the Internet.

[1] The Trusted Computing Base is the set of software that is critical for the security of the system.

Many partial solutions to protect the execution of applications (e.g., Address Space Layout Randomization (ASLR) [PT03,BDS03,BBSD05], Data Execution Prevention (DEP) [vdVM04], stack canaries [CPM+98,HK01], Software-based Fault Isolation (SFI) [WLAG93, SD01, SD02, KBA02, FC08, PG11], and policy-based system call authorization [GWTB96, KBA02, Pro03, GPR04, Bau06, FS08, WALK10, PG11]) exist and protect from specific attack vectors. But all these techniques have their weak points and can be circumvented by targeted attacks.

SFI has been embraced by many projects to address the problem of the trusted computing base. SFI is a technique that executes untrusted application code[2] in a sandbox. This sandbox controls every executed instruction and dynamically adds new security guards into the executed code. These guards can protect from code injection attacks (stack-based and heap-based code injection) that place new code in the address space of a running application.

User-mode SFI is restricted to code that is executed in user-space and does not restrict individual system calls. Many dynamic interception tools [SD02, KBA02, BGA03, SSNB06] rely on the standard loader to gain control of the execution flow, yet the loader is not part of the sandbox. This mismatch can be exploited by a potential attacker.

The dynamic loader is the first piece of code that is executed when a new process is created. The dynamic loader maps the application into memory and resolves all symbols that are used in the different shared libraries. These relocations and symbol lookups enable a program to use libraries and to implement different techniques like position independent code. After the program is prepared for execution (i.e., after the loader has finished with the initial relocation and loading), the program turns into a process and the initialization code (i.e., the main function) of the application is executed.

The loader has access to all symbols and relocated objects at runtime and shares this information with the executed program, e.g., to resolve new symbols dynamically, or to dynamically load additional libraries. The standard dynamic loader offers a broad functionality, e.g., debugging primitives, library overwrites, symbol overwrites, or alternative paths. The combination of the rich functionality that the standard loader provides and the fact that all applications (e.g., privileged applications, applications reachable over the network, or local applications) use the same loader make it a promising attack vector. Recent attacks [Dan10, Orm10b, Orm10a, Ros10, Bro11] illustrate the problem. This problem can be solved by adding the loader to the trusted computing base and making the loader part of the security framework.

A limitation of an SFI-based sandbox (without a special secure loader) is that the sandbox treats application code as a black box. The black box approach enables control flow-based attacks that use return and jump instructions to redirect control

[2]Untrusted application code is code that can contain vulnerabilities, but that is not malicious code.

flow to alternate code locations that are already available in the application. A clever combination of multiple such control flow redirects enables the construction of a chain of gadgets that executes malicious code without injecting new code into the application. Applications need a model that enumerates all allowed control flow transfers that is enforced by the sandbox. The enforcement of such a model is called control flow integrity.

1.1 Attack model

The attack model defines the constraints for an attack, the properties of a successful attack, and the limitations of the secure execution platform. The execution model explains which applications can be protected and describes changes to the original memory layout of the application. The flow of execution in an application can be seen as an explicit graph where control flow transfers are the edges and the nodes are individual basic blocks, i.e., a dynamic control flow graph. The execution model describes how a safe user-mode sandbox transforms this control flow graph at runtime to ensure that no "unwanted" locations are reached.

A potential attacker tries to escalate privileges by executing injected or constructed code. A local user escalates the available privileges to a higher privileged account (e.g., by triggering an exploit in a "Set User ID upon execution (SUID)" application to gain super-user access). A remote user (without the ability to execute arbitrary commands, e.g., using a web service) gains user-level access by escalating the available privileges to a local user account. Attackers use the application and maliciously crafted input that triggers security relevant bugs in the application.

1.2 Requirements for a secure execution platform

A platform for the secure execution of untrusted application code must ensure that no *code-oriented*[3] or *data-oriented*[4] exploits are possible in the running application. Such a platform is implemented as a dynamic sandbox that precisely controls the executed code. Mandatory factors for such a secure execution platform are:

1. The loader is part of the secure platform and ensures that no untrusted code is executed during the initialization of the application.
2. All application code is executed in an uncircumventable sandbox.
3. Security guards extend the sandbox and stop code-based intrusion vectors.
4. A fine-grained execution model controls the application code and limits illegal, or unintended control transfers of the application.

[3]Code-oriented exploits inject new executable code into a process and execute it.
[4]Data-oriented exploits reuse existing code with alternate and malicious data.

5. The application can only execute system calls that are needed to fulfill the purpose of the application.

The security guards of the sandbox protect the application from all code injection attacks. A code injection attack is an exploit form that places new (malicious) code in the memory space of an application and redirects control flow to that injected code. Code can be injected (as data) through, e.g., a buffer overflow, but the sandbox never executes the injected code. The sandbox either detects an illegal code region that contains code when a "control flow instruction"[5] attempts to transfer control to a data region or the kernel generates a protection fault if the application tries to write to a code region. A shadow stack [FS01, PcC03] in the sandbox domain ensures that return-oriented programming [Sha07] attacks are not possible. The sandbox dynamically removes indirect control flow transfers whenever possible to reduce the opportunities for jump-oriented programming [BJFL11]. The sandbox uses per-application system call policies to protect from remaining attack vectors. Data-based attacks and jump-oriented programming attacks are stopped whenever an illegal system call is executed. Attacks against the sandbox are limited by protecting internal data. All data structures of the sandbox domain (including the secure loader) are write-protected during the execution of translated code.

The sandbox kills the application if an attack is detected. Denial of service attacks (e.g., an attacker can repeatedly kill an application by trying to exploit a vulnerability) are outside of the scope of the attack model. Several mitigation techniques exist to restart failed services but they are not the topic of this thesis.

1.3 The foundation of a secure dynamic execution platform

This thesis presents a novel *model-driven* approach to application security that fulfills the requirements presented in Section 1.2. The secure platform for the execution of untrusted application code combines a secure loader, a user-space sandbox, and a user-space policy enforcement mechanism for system calls and arguments. Figure 1.1 gives a high-level overview of the secure execution platform.

Our approach replaces the standard loader with a security-aware loader. The secure loader is a part of the secure execution platform and offers a restricted secure API to the application. The secure loader is the foundation for a secure separation of the sandbox and the (translated) application in user-space by ensuring that no untranslated code is executed in the startup phase of the application. The secure loader ensures that the sandbox is initialized before any application code is loaded and runs all application code in the sandbox.

[5] A control flow instruction is any instruction that changes the control flow of the program, e.g., jump instructions, indirect jump instructions, call instructions, indirect call instructions, or return instructions.

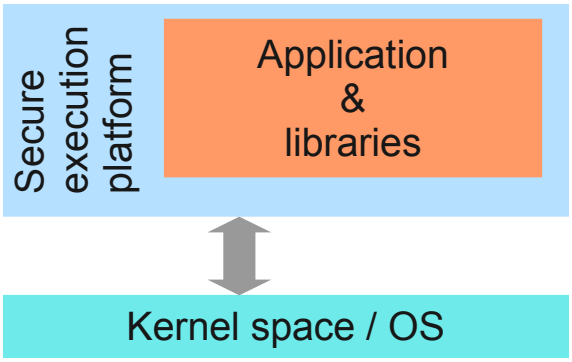

Figure 1.1: Overview of the secure execution platform. The complete application is executed in the secure execution platform.

The sandbox realizes SFI through dynamic binary translation. All application code is translated before it is executed. SFI adds dynamic guards during the translation process and ensures that no invalid code (e.g., code in data regions or stack regions) can be translated. Invalid instructions, malicious instructions, or errors during the translation (e.g., illegal library imports, or illegal targets for control flow transfers) terminate the application. The sandbox ensures that no code injection exploits are possible. Attempted code injection exploits are detected and the application is terminated.

An extension of the control flow transfer checks verifies all target locations according to a given control flow model. The model contains a set of all valid targets for each source location. The control flow transfer is only allowed if the current target is in the set of valid target locations for the current source location. The dynamic verification of the model implements a form of dynamic control flow integrity. These extended control flow transfer checks extend the protection of the sandbox from code injection exploits to any code-oriented exploits. Data-oriented exploits reuse already existing code in a way that was not intended by the programmer, e.g., by combining multiple short instruction sequences into malicious code using unintended control flow transfers. If a data-oriented exploit is detected then the application is terminated with a security exception.

The tight integration of a secure loader into the sandbox is used to dynamically generate the model used for the control flow transfer checks. The loader shares information about all loaded symbols (e.g., functions) with the sandbox and constructs a control flow transfer model that lists all valid targets for each control flow transfer. Targets of function calls are limited to symbols in the same shared object or to symbols imported from other shared objects. Jump instructions are limited to the region of the current symbol. The security model peeks into the application black-box and uses third-party information (symbol information from the linker) to verify all control structures of the process image. The additional information

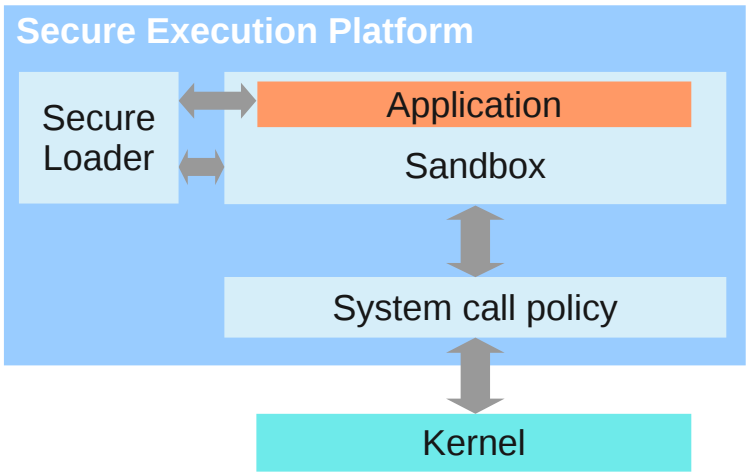

Figure 1.2: Detailed overview of the secure execution platform.

offers a fine-grained and detailed view of the application and the used interfaces. This model-driven approach combined with SFI allows fine-grained control of the executed code.

All system calls of the application are treated as special instructions. System calls are redirected to an authorization layer that checks the system call and all arguments of the system call. For each system call the sandbox verifies that the system call and all its parameters are allowed according to a given per-application policy. The policy limits the system calls (and arguments) that an application can execute. The system call authorization layer protects from data-based exploits. Data-based exploits overwrite data structures of the application. These changed data structures are later used by legit code and result in an illegal operation, e.g., a system call with malicious parameters.

Figure 1.2 gives a detailed overview of individual modules for the secure execution platform including the secure loader. All application code is wrapped and executed in a sandbox layer. The sandbox controls all interactions of the translated application code with the operating system. The application can load additional libraries only using the secure loader. System calls are executed only through the system call interposition API. This API checks that the arguments correspond to a well-defined policy. Our approach uses compiler techniques to dynamically rewrite machine code. The translated code is checked for potentially malicious instructions. All control flow transfers are hardened using additional translation-time or runtime checks. Using this isolation layer all system calls are redirected to a policy-based authorization framework.

An extension of this framework implements a check for dangerous patterns in system calls (e.g., policy violations) and is also used to check the program state (e.g., an extension of the secure execution framework protects running applications from file-based race conditions).

This comprehensive technique extends different fields of system-based security research (SFI, security guards, system call policies, policy enforcement, and race-detection in file accesses). Only such a holistic approach enables the protection from all known attack vectors, whether control flow-based or data-based.

1.4 Thesis statement

This thesis presents different building blocks of a secure execution platform for unsafe or untrusted applications. A binary translation-based sandbox enables SFI that is used to isolate traps and confine programming errors in the application. A novel secure loader that is part of the trusted computing platform forwards additional information to the secure execution platform. This additional information is used to construct an execution model for the application that encodes valid targets for all control flow transfers.

The observation that vital guards for the secure execution of untrusted code are missing in current platforms leads to the thesis statement:

> *"A secure dynamic execution platform that combines dynamic control flow integrity, a secure loader mechanism, and a sandbox to run untrusted code enables full protection from code-oriented and data-oriented exploits."*

The rest of the thesis presents the different building blocks to prove the thesis statement. The following chapters introduce the different security layers and present an evaluation of the security features.

1.5 Libdetox

Libdetox is a prototype implementation of the concepts presented in Section 1.3 for the Linux kernel. Libdetox is used to validate the thesis statement in Section 1.4.

Libdetox provides a secure loader as an alternative to the standard libc loader. The secure loader initializes the sandbox first, with dependences of the application being resolved afterward. The sandbox then uses dynamic binary translation to rewrite the application code before it is executed. During the translation process dynamic guards are added to the application code. The libdetox platform uses information from the secure loader to construct a detailed model of allowed control flow transfers similar to a control flow graph. This model is then checked in the control flow transfer guards in the translated code. The combination of the secure loader, the sandbox, and the control flow transfer checks using the detailed model enables protection from all code-oriented exploits.

A second line of defense uses a per-application policy to check every single system call and all arguments. This mechanism protects from data-based exploits.

The interaction between a secure loader and an extended sandbox enables a novel model for program instantiation. Security is a first class citizen and is provided right from the start.

1.6 Contributions

The contribution of this thesis is the design of a platform for the secure execution of untrusted application code. This platform follows a modular design that builds on a loader module, a sandbox module, and a system call policy module. The different modules share and exchange information and are all part of the trusted computing platform. We evaluate an implementation prototype to prove the practicability of the proposed design. A detailed list of contributions follows:

1. The design of a platform for the secure execution of untrusted code.
   (a) The design of a secure loader module that controls the startup of the execution platform. The secure loader ensures that the application cannot execute unchecked code during the startup phase and forwards crucial control flow information to the sandbox.
   (b) The design of a sandbox module that verifies all application code and adds security guards to the executed application code.
   (c) A system call policy that checks the communication between the application and the operating system by verifying every system call and the corresponding arguments.
2. An extensive evaluation of a prototype implementation of the secure platform. The complete platform and individual modules are evaluated in both performance and security aspects.
3. A case study that shows the extensibility and flexibility of the secure execution platform by adding a dynamic file-based race detection toolkit to the platform.

1.7 Publications

Portions of this work have been published at different conferences. The basic structure of the binary translator is published at AMAS-BT 2009 [PG09] and Systor 2010 [PG10]. The basic binary translator is called fastBT. The paper presented at 26c3 [Pay09] focuses on security implications for binary translators. A discussion of attack vectors through the eyes of SFI is presented at 27c3 [Pay10]. The paper in VEE 2011 presents libdetox [PG11] which extends the basic binary translator fastBT with additional security modules that protect the sandbox from attacks

against the sandbox itself and provide an additional policy-based system call interposition layer. The design of the secure loader and the integration into the secure execution framework is published at IEEE S&P 2012 [PHG12]. The combination of libdetox and secure loader builds the secure execution platform. The dynamic race detection module dynaRace [PG12] extends libdetox with a detection engine for file-based race conditions. DynaRace is published at VEE 2012 [PG12].

1.8 Outline

The thesis is organized as follows. Chapter 2 presents background information and different exploit classes that one needs to understand the context of this thesis. Chapter 3 discusses related work. Chapter 4 explains the design of the platform for the secure execution of untrusted code and the three modules that build up the platform. Chapter 5 describes the implementation of libdetox. Chapter 6 evaluates the security design and the prototype implementation of the execution platform. Chapter 7 shows a case study that extends the execution platform and implements a dynamic race detection toolkit that protects unaware applications from file-based race conditions. Chapter 8 presents future directions for research that extend the execution platform and shows different optimization opportunities. Last but not least Chapter 9 concludes this thesis.

2

Background information

This chapter presents background information that describes material needed to follow the design decisions of this thesis and presents details to put this work into context.

The ideas and concepts of this thesis apply in general and do not depend on a specific platform. The implementation prototype, some specific attack classes, and some low-level optimizations use specific features of the x86 Instruction Set Architecture (ISA) described in Appendix A.

Successful exploits on running applications require an attack vector, e.g., a bug in an application and typically specially crafted input data. Section 2.1 and Section 2.2 describe different attack vectors and possible exploit classes. Section 2.3 offers a comparison between Erlingsson's attack classification and the different exploit classes introduced in Section 2.2.

All dynamic applications use the standard loader to bootstrap the process image. The standard loader on regular Linux systems has several security weaknesses, as discussed in Section 2.4.

Binary translation, the art of rewriting code, is used as a transparent way of rewriting code before its execution. Our virtualization system uses binary translation to implement an additional abstraction layer between the application and the operating system where our security features are enforced. Section 2.5 lists the design criteria for binary translation. The concepts in this thesis apply for all operating systems but the prototype implementation assumes a Linux-based environment. The important details of the Linux loader and the Executable and Linkable Format (ELF) are listed in Appendix B. A Linux-oriented implementation must use the specific low-level Linux system calls. Appendix C describes the interface to these system calls.

2.1 Attack vectors

Exploits interact with a running application to trigger bugs or vulnerabilities. Exploits generate specially crafted input data that force the program to reach alternate locations and cause unintended behavior. Exploits are used to gain control of an

application, to escalate privileges, or to start a denial of service attack.

An exploit is implemented in two stages: (i) inject alternate code that will be executed or data that will be interpreted in an unintended way, (ii) redirect the control flow of the program to execute the injected code or to interpret the injected data. An attack vector describes a way how an exploit can attack a running application.

This section explains various different attack vectors. Exploits are enabled through unchecked and unvalidated assumptions on input data, e.g., the programmer expects that the user always passes less than 20 characters in the name field. If the maximum length of the name is neither enforced by the programmer nor by the runtime system then the missing bound check can be used to exploit the application.

An attack vector is a prerequisite for an exploit, being used to prepare the later stages of an exploit. Examples of possible attack vectors are:

Buffer overflow: A buffer overflow is the result of an operation that copies data into a buffer that is not large enough to hold that data. A buffer overflow overwrites data structures following or preceding the buffer location. Buffers can be located either on the stack of a thread or on the heap of the application.

User-controlled format string: If an unchecked user-controlled string is passed to any `printf` function of the standard libc then malicious format characters in the user string are interpreted which can lead to random memory reads and random memory writes.

Dangling pointer: A dangling pointer in a data structure is used to read or write unintended memory locations. An attacker can carefully plan the allocation and deallocation sequences and use a dangling pointer to overwrite data.

Arithmetic overflow: The precision of integer numbers is limited when using low level instructions. If the result of an arithmetic operation is larger or smaller then the largest or smallest representable number an overflow occurs. The ISA usually flags an overflow and the operation wraps around. Missing overflow checks can lead to incorrect results that are used in future computations.

2.2 Exploit classes

This section introduces the different exploit classes and discusses the necessary requirements for successful exploitation. All exploit classes have in common that they redirect control flow to new or alternate locations that would not be reached in an unaltered run. Control flow can be altered through buffer overflows, format string attacks, or data attacks.

All exploits use the fact that the programmer and/or the runtime system are unable to check the bounds of a buffer or to detect a type overflow (e.g., integer overflow) to overwrite either data structures or code.

2.2.1 Code injection

Code injection attacks place new machine code in the running application and redirect the control flow to the newly placed code. A code injection attack writes additional code into an executable region of the application's memory image and transfers control to that injected code [Ale96]. Code injection attacks often use a buffer overflow (e.g., for a C-based string or array) to inject the code and to overwrite a stored instruction pointer in one step. The injected code carries the malicious payload.

Code injection is one of the oldest attack vectors and has been used for many years. Until recently Intel IA32 did not support the separation of code and data. The CPU tried to interpret any memory region as executable, enabling code injection attacks into data regions. The 64-bit extension x64 includes the PAE feature that enables separation of data and code. Only code on pages that have the executable flag set is executed by the CPU. If an exploit redirects control flow to a data page then an exception is triggered. Nowadays most systems support $W \oplus X$[1]; a memory page is either writeable or executable. Due to $W \oplus X$ this attack vector is only applicable under special circumstances (e.g., executable trampolines on the stack, shared memory regions with wrong permissions, or exploitable just-in-time compilers).

Stack-based code injection

This attack exploits missing or incomplete bound checks of a local array on the stack. The array is filled with user-controlled code. Because the stack grows downwards it is possible to overwrite the variables that are higher up on the stack, including the base pointer and the return instruction pointer. The attack adjusts the return instruction pointer so that it points to the code that was placed in the buffer. The function therefore does not return to the caller but to the code on the stack.

[1] $W \oplus X$ is also called Data Execution Prevention (DEP) [vdVM04]

```
int is_foobar(char *cmp) {
  // assert(strlen(cmp) < MAX_LEN)
  char tmp[MAX_LEN];
  strcpy(tmp, cmp); // no bound check
  return strcmp(tmp, "foobar");
}
...
// user_str is > MAX_LEN
if (is_foobar(user_str))
...
```

Listing 2.1: A potential stack-based overflow

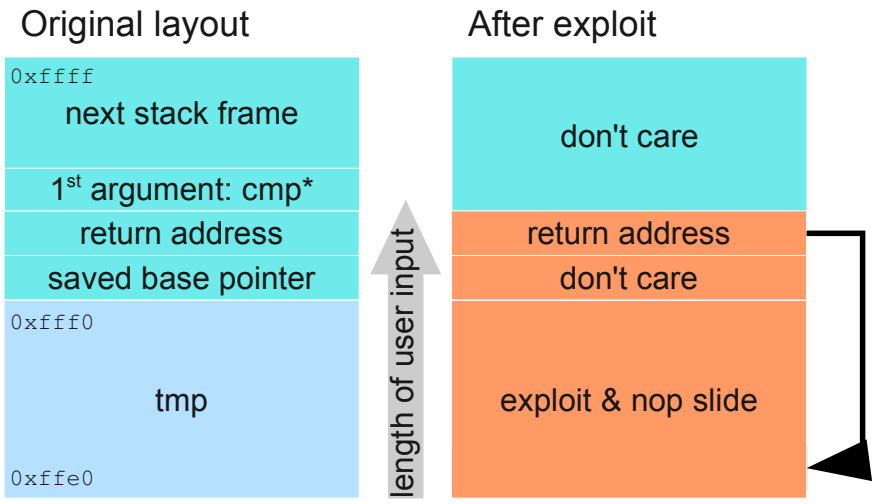

Figure 2.1: Stack before and after a stack-based code injection exploit.

Listing 2.1 shows a simple code sequence that is prone to a stack-based overflow where code injection is possible. Figure 2.1 shows the stack layout before and after an overflow. The length of the user supplied input overflows the length of the variable on the stack and overwrites the base pointer and the return instruction pointer. The new return instruction pointer then points to the beginning of the temporary array on the stack. The exploit code is executed when the function returns.

Constraints for this attack are that code on the stack must be executable, a bound check for a buffer on the stack must be missing or faulty, and the runtime system must not verify the return instruction pointer.

Heap-based code injection

This form of attack places the malicious code in an array on the heap and redirects control flow to the injected code. The control flow redirection is achieved through overwriting the return instruction pointer, overwriting a function pointer (or `vtable` entry for C++), adjusting standard libc destructors, or fiddling with the memory allocator data structures.

Listing 2.2 shows a simple code sequence with a vulnerable `struct` and a code sequence that is prone to a heap-based code injection. Figure 2.2 shows the vulnerable struct before and after an exploit. The buffer in the vulnerable struct is filled with a user supplied nop-slide[2] and exploit code. The function pointer is overwritten and points somewhere into the nop-slide of the struct's buffer. The exploit code is

[2]A sequence of nop instructions is used if the exact address of the buffer is not known but only the region of the target buffer. The exploit transfers control somewhere into the nop-slide.

```
typedef struct vuln_struct {
  char buf[MAX_LEN];
  int (*cmp)(char*);
};
int is_foobar_heap(vuln_struct *s, char *str) {
  // assert(strlen(cmp) < MAX_LEN)
  strcpy(s->buf, str); // no bound check
  return s->cmp(s->buf, "foobar");
}
...
vuln_struct *st = \
  (vuln_struct*)malloc(sizeof(vuln_struct));
// user_str is > MAX_LEN
if (is_foobar_heap(st, user_str))
...
```

Listing 2.2: A potential heap-based overflow

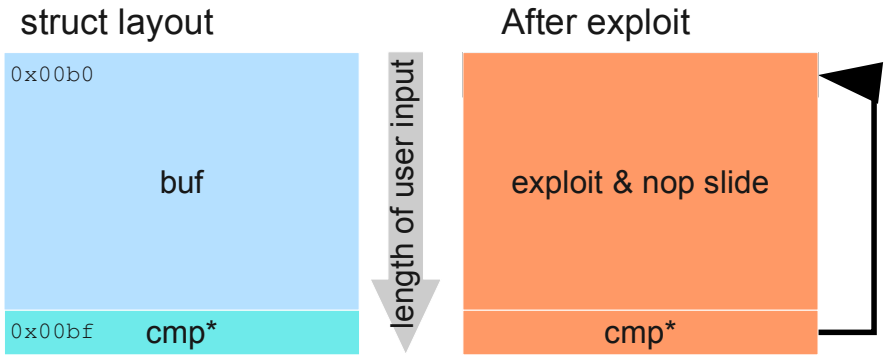

Figure 2.2: Vulnerable struct before and after a heap-based code injection exploit.

executed when the function pointer is called.

The location where the code is placed must be executable and a bound check for a buffer must be faulty or missing. Additionally, the control flow must be redirected to the code placed on the heap.

2.2.2 Return-oriented programming

Return-oriented programming (ROP) [PB04, Sha07, Ner07] uses pre-existing code sequences to execute malicious computation. A return-oriented attack constructs a set of stack invocation frames that are executed one after the other. Each stack

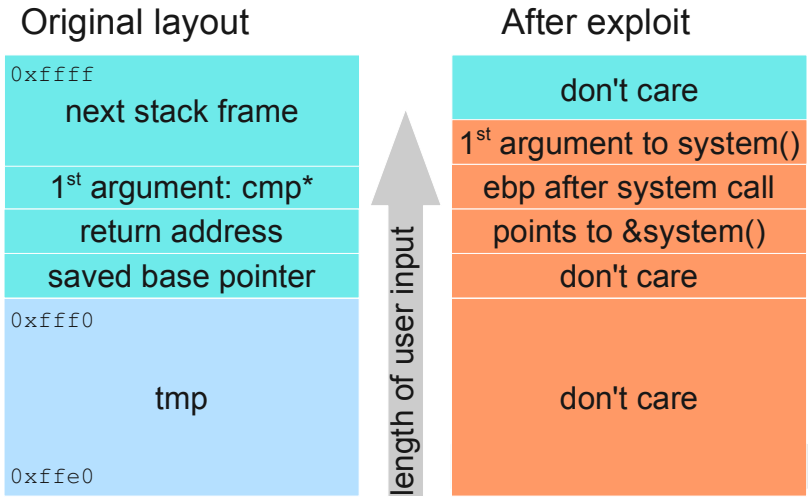

Figure 2.3: Stack before and after a stack-based ROP injection.

invocation frame prepares a set of parameters on the stack and targets a gadget[3] that uses the parameters and executes arbitrary computation. ROP is a code-reuse technique that reuses the stack pointer as an indirect instruction pointer.

A stack-based buffer overflow as shown in Listing 2.1 can be used to prepare the stack for an ROP attack. Figure 2.3 shows a simple ROP attack that uses a buffer overflow on the stack. The buffer and the base pointer are overwritten with garbage data, the return instruction pointer is redirected to a function in the standard libc (`system()` in this case). On the stack the function pointer is followed by the overwritten `ebp` and the arguments to the function.

This attack relies on a stack-based overflow and works only if the return instruction pointer is not verified before it is dereferenced.

2.2.3 Jump-oriented programming

Jump-oriented programming [PB04, BJFL11] (JOP) is a generalization of return-oriented programming. Return-oriented programming relies on the control of the stack pointer and of the return instruction pointer. Jump-oriented programming relies on the control of the instruction pointer and any other register. Both programming styles inject data to modify the control flow of the application using gadgets. Jump-oriented data is not limited to stack overflows but uses modified indirect control flow transfers to construct a chain of gadgets.

[3] A gadget is a code snippet (not necessarily a function) that already exists in the memory image of the application.

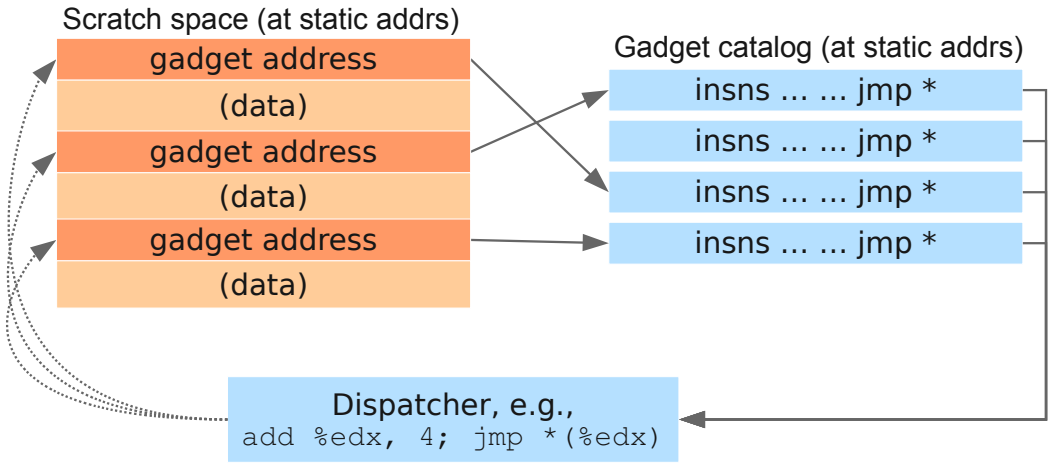

Figure 2.4: A generic JOP attack.

Just like return-oriented programming jump-oriented programming relies on a set of gadgets that are located in legacy code (e.g., the standard libc). These gadgets are chained using a dispatcher gadget that reads an attacker controlled table with a set of gadget locations. For example, the dispatcher can be implemented as an indirect jump through an attacker-controlled register. Figure 2.4 shows a generic JOP attack with a scratch space somewhere on the application's heap. A successful exploit prepares the scratch space in a data region and redirects the control flow of the application to the beginning of the scratch space.

Two forms of jump-oriented programming are described in related work. Checkoway et al. implement return-oriented programming without return instructions [CDD+10] while Bletsch et al. [BJFL11] and Ping et al. [CXM+11] go one step further with the definition of jump-oriented programming and the description of the dispatcher gadget.

2.2.4 Format string attacks

A format string attack [OWA, Haa10, Pla10] exploits a user-controlled string that is passed to a function of the `printf`-family. The `printf`-family parses the first string argument for control tokens (of the form `%T`) to determine the number of variable parameters that follow. Many programmers forget to check user-controlled strings for these control tokens and pass the string directly to the function (e.g., `printf(usr_str)`). A safe implementation would use a static parameter to pass a single string (e.g., `printf("%s", usr_str)`).

The `%n` token is a special control token that reverses the order of input. All other tokens specify a format that is used to print the current argument. `%n` writes the count of already printed characters to the address given by the current argument. Any argument on the stack can be used as a target address for `%n` with careful

```
void foo(char *arg) {
  char text[1024];
  if (strlen(arg) >= 1024) return;
  strcpy(text, arg);
  printf(text);
}
...
foo(user_str);
...
```

Listing 2.3: A potential format string attack

encoding of the format string. The format string itself can be used to store pointers to specific addresses if it is placed on the stack. The number of written bytes can be controlled with additional parameters (e.g., `printf("%NNc");` prints NN bytes and increases the counter used for `%n`).

The malicious format string can use other tokens like `%p` to read specific pointers on the stack, and `%s` to read specific stack addresses as strings. An attacker uses these parameters during the construction of the format string.

The combination of input tokens (e.g., `%p`, `%s`, and `%x`) are used to gather information about the stack layout (if the binary is not available). The gathered information about the stack frame is then used to construct an exploit string that consists of `%NNc` to set single byte values (i.e., to increase the number of written bytes) and `%hhn` (hh represents a half half word and equals one byte) to write single bytes to given memory addresses. The exploit string then writes arbitrary values to arbitrary memory locations. These random writes are used to redirect control flow to injected code. The injected code is often in the string itself.

Listing 2.3 shows code with a potential format string exploit. Using a combination of `%x` to increase the count of printed characters and `%n` that writes the number of already printed characters to a given location it is possible to read and write arbitrary memory locations. Assume that in Listing 2.3 the return instruction pointer is at `0xffffd37c` and the text array is 11 words higher up on the stack relative to the printf call. An input string of `(0xffffd37c)(0xffffd37e)%12$2043x.%12 $hn%11$32102x%11$hn` will overwrite the return instruction pointer with `0x0804856a` by writing two half words namely $0x0804 - 9 = 2043$, and $0x856a - 2043 = 32102$.

Depending on the environment a format string exploit overwrites the return instruction pointer, Global Offset Table (GOT), or the list of destructors. All of these variations alter the control flow at some point in time.

The requirement is that the user string must contain escape sequences like `%n` or `%s` that are then parsed and expanded by `printf`.

```
void foo(int len, char *pack) {
  char *response;
  if (len > 0) {
    response = malloc(len*sizeof(char*));
    memcpy(response, pack, len);
  }
}
...
foo(user_len, user_packet);
...
```

Listing 2.4: A potential arithmetic integer overflow

2.2.5 Arithmetic overflow

Arithmetic data types always have specific bounds. An 1 byte data type can only store 256 different values. If an operation (e.g., an addition) exceeds these bounds then the variable wraps around and continues on the other end (e.g., $127 + 1 = -128$ for an 1 byte, signed data type, or $255 + 1 = 0$ for an unsigned 1 byte data type). These overflows can be used to bypass bound checks (e.g., before memory is allocated).

Listing 2.4 shows a potential arithmetic overflow. If `len` has the value $0x40000000$ then $len > 0$ holds but the result of the multiplication in the `malloc` call is 0. The following `memcpy` will overwrite data structures on the heap, resulting in a heap-based overflow.

Requirements for an arithmetic overflow are lax or implicit type conversions, sign errors, rounding errors, type overflows due to arithmetic operations, and pointer arithmetic.

2.2.6 Data attacks

A data attack exploits a missing or faulty bound check to write data to an user-controlled address. This random write is used to, e.g., redirect control flow to injected code or to set up a secondary attack.

Listing 2.5 is prone to a potential data attack. The attacker controls the position and the value that is written. A simple calculation relative to the position of the data array enables a random write to (almost) any memory location. The target location and possible values are often limited due to address calculation and range checking in real-life programs.

Requirements for a data attack are unfiltered or only partially filtered user input and missing or faulty bound checks.

```
void foo(int pos, int value, int *data) {
  data[pos] = value;
}
...
foo(user_pos, user_value, data);
...
```

Listing 2.5: A potential data attack

2.2.7 Mixing x86_64 and x86 code

Modern operating systems support 64-bit x86_64 and 32-bit x86 code. An application can even use both modes interchangeably. Mixing x86_64 and x86 code is used to trick static verifiers or to escalate privileges. Most static verifiers and system call authorization frameworks are limited to either x86_64 or x86 code. An application that uses both instruction sets to execute system calls or specific control transfers can to escape the control of the guards [Eva09].

2.2.8 Overwriting other data structures

Other data structures (e.g., system call parameters, data structures of the memory allocator, data structures of the dynamic loader, or destructors) can be overwritten by random writes.

When these data structures are used later on in the program (e.g., when data is freed or the destructor of a shared library is called) then the exploit can carry out its malicious payload.

2.3 Comparison to Erlingsson's attack classification

Erlingsson describes a series of attacks and defenses in [Erl07]. According to Erlingsson a successful attack discovers what assumptions were made by the programmer and crafts an exploit outside those assumptions. We relate the following four attacks to the attack vectors introduced in Section 2.1 and the exploit classes listed in Section 2.2.

2.3.1 Return address clobbering

This attack uses a buffer overflow on the stack to overwrite the return instruction pointer or the stack base pointer. This attack is equivalent to stack-based code injection or return-oriented programming.

Stack-based code injection relies on an executable stack and return-oriented programming relies on available gadgets in loaded libraries. Both attacks rely on a missing bound check and missing stack protection. See Section 3.6 for compiler extensions that can protect the stack.

2.3.2 Corrupting heap-based function pointers

This attack corrupts data structures on the heap by using a heap-based buffer overflow. Exploited data structures are virtual tables of object-oriented languages (e.g., C++) or function pointers. A variant of this attack corrupts heap metadata (e.g., malloc's data structures).

We differentiate between heap-based code injection, jump-oriented programming, and overwriting other heap-based data structures.

2.3.3 Executing existing code via bad pointers

This attack is based on the knowledge of the position of specific code sequences. We differentiate between return-oriented programming where the control structures for the attack are on the stack and jump-oriented programming where the control structures can be anywhere in the application's memory.

2.3.4 Corrupting data that controls behavior

This type of attack does not rely on control-data and is a data-only attack. The attack uses the boundaries of the legal control flow graph. Specific data is replaced by using random writes or data attacks. These attacks use random writes to replace, e.g., specific parameters for system calls. We classify these attacks as data attacks.

2.4 Security of the standard loader

The standard loader `ld.so` is the first code that is executed when an application starts. This piece of code is used to load all the application and all used shared libraries. The standard loader is not designed for security. This lack of design results in several problems if the standard loader is used in a security-relevant context. Bugs in the standard loader lead to direct privilege escalation. If the application and the sandbox share the same loader then the sandbox can be attacked through the loader. All dynamically loadable applications rely on features of the loader to dynamically resolve references or to load additional modules. If the loader is translated alongside the application then the application must have the privilege to map code as executable. It is a security risk if (loader) code in the unprivileged domain is allowed to map executable code.

2.4.1 Exploiting the standard loader

The standard execution model uses the same loader for all applications, no matter if they are regular user applications, privileged applications, or remotely accessible applications.

The standard loader supports a wide range of dynamic functionality (e.g., debugging, dynamic library replacement, and tracing of method calls) and a huge feature-set (e.g., many different relocation types). Large parts of the functionality are not needed or are even harmful for privileged programs. Extra checks in the code ensure at runtime that it is not possible to (i) preload alternate libraries or to (ii) replace the standard search path for libraries if an application uses the Set User ID upon execution (SUID) flag.

Missing or faulty checks for privileged applications or other bugs in the standard loader [Dan10, Orm10b, Orm10a, Ros10] can therefore be exploited to escalate privileges for SUID applications.

These problems can be mitigated or reduced if the functionality and feature-set of the loader is restricted to the bare minimum of necessary features to execute current applications.

2.4.2 The late interception problem

Many dynamic interception tools [KBA02, SD02, BGA03, SSNB06, FC08] use `LD_PRELOAD`[4] to gain control of the application. The application, the standard loader, and the sandbox share the same memory space. The data structures of the loader contain pointers to the sandbox as well as to the application. The loader is in the application domain and the loader functionality can be used to gather information about the sandbox (i.e., resulting in an information leak) or to break the integrity of the sandbox. A potential exploit uses the data from the loader (e.g., the GOT section of the sandbox) to compromise the sandbox itself and to redirect sandbox functions to malicious code.

The standard loader treats a sandbox that uses `LD_PRELOAD` just like any other shared object and enables the application to read information about the shared object (e.g., address space, Procedure Linkage Table (PLT), and GOT sections). The standard loader does not guarantee that the sandbox initialization function is the first sequence of instructions that is executed after the loader finishes its initialization. For example, the `INITFIRST` flag can be set by multiple libraries but only the initialization code of one library is actually executed first. Preloaded libraries are relocated first but the standard loader executes the initialization code of the last loaded library that sets the `INITFIRST` flag first.

[4] `LD_PRELOAD` is an option for the dynamic loader that injects an additional library into the process space of the application; this option executes the initialization code of the injected library prior to the main function of the application.

Another example is a symbol of the `STT_GNU_IFUNC` relocation type. The standard libc uses `STT_GNU_IFUNC` to select the fastest version of a function for the current hardware at runtime. Such a scenario may trigger executing of code before the `LD_PRELOAD`-based sandbox is initialized. The sandbox is deprived of control of its environment if application code is executed before the sandbox is initialized.

2.4.3 The loader black box

In related work [KBA02, BGA03, SD02, SSNB06, PG11] the sandbox is either (i) unaware of the standard loader and translates the code of the standard loader as part of the application, or (ii) does not support dynamic loading [ABEL05, MM06, EAV+06, YSD+09]. Solutions that are unaware of the loading process treat library loading as a black box.

The loader plays a privileged part during the runtime of all applications that use shared libraries. The dynamic loader manages information about all loaded shared objects (libraries) and about all exported symbols that can be used in other objects. The sandbox uses functionality of the loader to discover the loaded shared objects and the exported functions. The sandbox also relies on information about executable regions and data regions that is exported by the standard loader.

The loader is a crucial component of the application as it can load and map new code into the running process. If the loader is translated as a black box then the application must have the privileges to load and map any code (e.g., using the `mmap` system call, or the `mprotect` system call to map memory regions as executable). On the other hand if the sandbox provides a transparent and secure loader API then the privilege to map executable memory regions can be abstracted into the trusted sandbox domain. The sandbox can control the application and limit the loading process to predefined libraries.

An extension of the secure loader can be used to implement a clear separation between the different shared objects. Privileges and permissions (e.g., specific system calls and parameters to the system calls) can be tuned and specified on a per-object basis and are no longer enabled for all parts of an application.

2.5 Binary translation

Binary translation is a technique that translates code from a source ISA to the destination ISA. Binary translation enables the emulation of a specific ISA. Possible applications for binary translation are (i) debugging: additional logic is added to the translated code; (ii) testing: assertions are evaluated during the execution of the program; and (iii) security: additional checks are added to the translated code. The source ISA and destination ISA can be the same.

There are two forms of binary translation, *static, ahead-of-time binary transla-*

tion and *dynamic, just-in-time binary translation*. Static binary translation converts the code of an application before execution into the target ISA and adds all instrumentation at compile time. Static binary translation does not incur translation overhead during the execution of the application, but the generated code may introduce overhead. Dynamic binary translation converts code during the execution of the application (typically on the order of a basic block) and incurs a runtime translation overhead. Many dynamic binary translators use code caches to lower the translation overhead: translated code is placed in a cache, and subsequent executions of the same region benefit from the already translated code. A key advantage of dynamic binary translators over static binary translators is that they translate only code that is executed. For example, a hotspot compiler reduces the execution time of code that is executed frequently compared to static ahead-of-time translation.

The interception mechanism of dynamic binary translators uses specific features of, e.g., the Linux loader to gain control of the application control flow. In addition, the ELF format is used to gather information about the application.

Static binary translation is unable to translate dynamic code (e.g., dynamically loaded libraries, dynamically generated code, and self-modifying code). Security-related binary translation must therefore use dynamic binary translation to cover all possible code locations.

Two key aspects in designing a fast binary translator are a lean translation technique and low additional runtime overhead for any translated code region. Indirect control transfers are the source of most of the dynamic overhead [PG09]:

1. **Indirect jumps:** The target of the jump depends on dynamic information (e.g., a memory address, or a register) and is not known at translation time. Therefore a runtime lookup is needed.

2. **Indirect calls:** Similar to indirect jumps the target is not known and a runtime lookup is needed for every execution.

3. **Function returns:** The target of this indirect control transfer is on the stack. The stack contains untranslated addresses only and a runtime lookup is needed.

2.6 Summary

This chapter presents background information that is needed to understand the context of this thesis. Information on attack vectors from Section 2.1 helps in understanding the different threats that Software-based Fault Isolation (SFI) must protect against. Section 2.4 discusses the security of the standard loader. The loader in the context of SFI faces two problems: the late interception problem where the loader executes code before the sandbox is initialized, and the black box problem where the SFI system is not aware of the loader. Section 2.5 introduces information about binary translation in general.

3

Related work

This chapter discusses related work for this thesis. The prototype implementation of the secure execution platform relies on dynamic binary translation. Important related work that lead to the development of a dynamic binary translator is described in Section 3.1.

Software-based Fault Isolation (SFI) can be implemented using binary translation. Binary translation instruments the executed code and adds the specific security guarantees. Section 3.2 discusses related work that implements SFI in user-space. System call authorization described in Section 3.3, is a technique which implements security profiles at the boundary between applications and the kernel. Section 3.4 discusses "full system virtualization": a coarse-grained view that implements security in a virtual machine monitor outside the operating system.

Static program verification in Section 3.5 is an alternative to dynamic SFI. A static binary rewriter and a verifier ensure that the binary code is unable to circumvent the static sandbox. Control Flow Integrity (CFI) and XFI (an extension of CFI) are static program rewriters with specific security guarantees.

Section 3.6 describes compiler approaches that mitigate several attack vectors for system software. Many of these compiler extensions can also be implemented in a dynamic protection system that builds on SFI.

Section 3.7 summarizes the different related work and highlights the most important security solutions in comparison to our approach. Different features are evaluated based on general approaches, implementation techniques, and security features.

3.1 Binary translation

Binary translation has a long history [Gil51, DS84, May87]. In this section we focus on current binary translators that are similar to our basic binary translator described in Section 5.2, the baseline binary translator of our prototype implementation. Dynamic binary translators can be separated into two groups. The first group uses a rewriting scheme based on low-level translation tables (e.g., Mojo [WKCG00], HDTrans [SSNB06, SSB07], and our approach: libdetox [PG09, PG10, PG11]). The

second group parses the machine code stream into an intermediate representation (IR) and uses common compiler techniques to work on that IR (e.g., DynamoRIO [BGA03], PIN [LCM+05], and Valgrind [NS07]).

IR-based approaches offer the advantage of more sophisticated optimization possibilities, but the IR translation comes at a higher translation cost. IR-based translators can use profiles to generate runtime information but there is no structure or type information at the machine-code level. Therefore it is hard to achieve good performance with IR-based translation systems. Some binary translation systems (e.g., [KF01, HKZ+06]) use a dual approach of profiling and adaptive optimization to limit the dynamic compilation overhead. Fx!32 [CHH+98] uses emulation for infrequently executed code and profile generation and offline static recompilation of hot code to speed up overall execution.

The most important overhead of dynamic binary translators is the handling of indirect control transfers. Hiser et al. [HWH+07] cover different indirect branch mechanisms like lookup inlining, sieves, and a translation cache.

3.1.1 HDTrans

HDTrans [SSNB06, SSB07] is a lightweight, table-based instrumentation system. A code cache is used for translated code as well as trace linearization and optimizations for indirect jumps. HDTrans resembles our basic binary translator most closely with respect to speed and implementation, but there are significant differences.

HDTrans requires hand-coded low-level translation tables that specify translation properties and actions for every single instruction. This setup requires a deep understanding of the translator and relations between different optimizations. Our basic binary translator raises the level of interaction with the translation system. Low-level translation tables are generated using a table generator that specifies transformations on a high level. Transformations can be specified on the properties of individual instructions (e.g., memory access, specific registers, or extension groups like SSE and FPU), or on a specific instruction opcode.

HDTrans uses a hash table of jump code blocks called a "sieve" to make the indirect jumps faster. The target is hashed and then a jump into the sieve is issued. The code block in the sieve then takes care of the jump to the correct location whereas our basic binary translator uses a simpler hash table and hand optimized assembly code that is emitted into the code cache. Instead of a sieve, our basic binary translator uses a lookup table that maps locations in the code cache to locations in the original program.

Return instructions are handled differently from normal indirect control transfers to take care of the `call/ret` relationship. HDTrans uses a (small) direct mapped cache to speed up return instructions, with a fall-back to the sieve. Our basic binary translator uses a similar approach with the return cache that uses a bidirectional protocol.

Compared to our basic binary translator, HDTrans translates longer chunks of code at a time, stopping only at conditional jumps or return instructions. This strategy can result in longer stalls for the program. Trampolines to start the translator for not already translated targets are inserted into the basic blocks itself. Therefore the memory regions for these instructions cannot be recycled after the target is translated.

3.1.2 Dynamo and DynamoRIO

Dynamo is a dynamic optimization system developed by Bala et al. [BDB00]. DynamoRIO [BDA01, BGA03, HS06] is the binary-only IA32 version of Dynamo for Linux and Windows. The translator extracts and optimizes traces for hot regions. Hot regions are identified by adding profiling information to the instruction stream. These regions are converted into an IR, optimized and recompiled to native code. DynamoRIO is also used in a project that includes a manager for bounded caches that clears no longer needed blocks out of the code cache [HS06]. The main purpose of the DynamoRIO project is shifting from a binary optimizer to a binary instrumentation system.

Compared to our basic binary translator, DynamoRIO translates machine code into an IR and uses a binary optimization framework and heavy-weight transformations to generate code. Our basic binary translator, on the other hand, uses a lightweight table-based approach.

3.1.3 PIN

PIN [LCM+05] is a dynamic instrumentation system that exports a high-level instrumentation API that can be used during the runtime of a program. The system offers an online high-level interface to the instruction-flow of the instrumented program. A user program can define injections or alterations of the instruction stream of a running application. PIN uses the user supplied definition and dynamically instruments the running program. PIN employs code transformations like register reallocation, inlining and instruction scheduling to make the instrumented code fast. Nevertheless, the online high-level interface results in relatively high overhead. Unfortunately, the instrumentation system of PIN (except the library, which is available as open source) is distributed in binary only.

In contrast to PIN, our basic binary translator offers a high-level interface at *compile time*. A table-based translator that is then used at runtime is generated using the high-level interface. The alteration possibilities at runtime are limited for our basic binary translator, but arguably a security framework does not need this flexibility.

3.1.4 Valgrind

Valgrind is described in [NS07] as heavy-weight dynamic binary analysis tool. The main applications are checkers and profilers. Machine code is translated into an IR, which is instrumented using high-level plugins. Valgrind keeps information about every instruction and memory cell that is used at runtime. This feature enables the implementation of new and different instrumentation tools that are not possible in other environments. Valgrind is designed primarily for flexibility and not for performance. This makes Valgrind perfect for basic block profiling, memory checking and debugging. Due to its high overhead, Valgrind is unsuited for large and performance critical programs.

3.2 Software-based Fault Isolation (SFI)

Vx32 [FC08] implements a user-space sandbox built on binary translation that uses segmentation to hide the internal data structures. Due to the use of segmentation the Vx32 system is limited to 32-bit code. Interrupts, system calls, and illegal instructions are translated to traps that call special handler functions. These design decisions lead to high execution overhead as there are no optimizations for indirect control transfers. The traps for system calls offered by Vx32 are targeted towards a reimplementation of the system calls and are not intended for a policy-based system call authorization framework.

Strata [SD01, SD02] is a safe virtual execution environment using software dynamic translation. It uses dynamic binary translation to isolate user-space programs and implements a basic system call interposition API. This API is used to instrument individual system calls. The translation framework is neither limited to a single system nor to a single architecture. Strata uses binary translation to enforce a non-executable stack but the current version has no security guards that mitigate against return to libc attacks or heap-based overflows.

Program shepherding [KBA02] uses the DynamoRIO [BDA01, BGA03, HS06] framework to safeguard running applications. A single policy is hardcoded and enforced using binary translation. The binary translator adds additional checks to restrict code origins and to control the targets of indirect control transfers.

Our approach implements a user-space sandbox and extends this sandbox with additional security guards that check the execution of application code at runtime. System calls in our approach are not replaced by software traps but are validated through a policy-based system call authorization framework. Additionally, libdetox does not depend on specific hardware features like segmentation, a feature that is not present on x86_64 and hinders portability. The policies used in the sandbox can be refined and changed without the need to recompile the safe execution platform.

3.3 System call authorization

System call interposition uses either a kernel module or `ptrace` support to implement a control mechanism on the level of system calls. These systems share several drawbacks. For one the protection mechanism is coarse-grained and does not detect the execution of malicious code until a system call that is not part of the policy is executed. These systems miss exploits that target opened files or use the policy's allowed system calls. In addition, these systems are often prone to Time Of Check To Time Of Use (TOCTTOU) attacks. These attacks evade detection by security systems that are only based on system call validation. A second concern is the overhead introduced due to context switching (more than 60x overhead for simple system calls according to [Pro03]). An expensive context switch has to be performed whenever a policy rule is checked. A third drawback is that these systems rely on trusted code in the kernel to stop the monitored program, and this setup poses an additional security risk.

Janus [GWTB96] is a system call interposition framework that uses the Sun Solaris process tracing facility (`ptrace`) to allow one user mode process to filter the system calls of a second process. This framework builds on kernel support and has two drawbacks: (i) the traced application is already in the kernel when it is stopped, a situation that poses a potential security problem, and (ii) the overhead of switching between an inspecting process and the corresponding application is high. Consh [AKS99], SubDomain [CBKH+00], MAPbox [AR00], and AppArmor [Bau06] extend the idea of a `ptrace` interposition framework by implementing policy-based authorization. Tal Garfinkel analyzed practical problems of system call interposition in [Gar03].

The Linux kernel offers an API for security modules [WCS+02] that can be used to implement many kernel-based coarse-grained security extensions. Systrace [Pro03] uses a kernel module to implement a system call policy. Some global system calls are validated in the kernel with low overhead (around 1% according to [Pro03]), but for all other system calls the program is stopped and a user-space daemon decides based on the parameters if the system call is allowed or not. Switchblade [FS08] enforces a system call policy using an in-kernel system call model and dynamic taint analysis. Capsicum [WALK10] offers capabilities and limits the set of system calls that an application can execute. Ostia [GPR04] prevents TOCTTOU attacks by using a proxy and a delegation model to delegate system calls to different processes or threads. Alcatraz [LSVS09] is a hybrid isolation approach that offers unrestricted read access to a sandboxed application but redirects all writes to a buffer which can be examined before it is committed. Unfortunately, this approach does not protect against data leaked over the network or processes that use local root exploits to gain privileges.

Our sandbox uses a fine-grained level of control that checks individual instructions and makes it impossible to execute injected code. Each thread of an application runs in its own sandbox. Each sandbox is secured against the execution of malicious

code, and as a second thread cannot execute code that races between the system call argument check and the execution of the system call; this restriction removes the threat of TOCTTOU attacks.

3.4 Full-system virtualization

Full system translation virtualizes a complete physical system, including the operating system and all hardware. QEMU [Bel05] offers full encapsulation with relatively high overhead (4-10x according to [Bel05]), while other frameworks like VMware [DBR02, Bug04] and Xen [BDF+03] rely on kernel or hardware support. Livewire [GR03] extends VMware and builds an intrusion detection system on top of the virtual machine monitor. Ho et al. [HFC+06] use a Xen-based virtualization approach to implement taint analysis that falls back to QEMU to analyze individual instructions. Aftersight [CGC08] logs non-deterministic inputs to the virtual machine and decouples analysis from normal execution.

The drawbacks of full system translation are the complexity of accessing shared data and handling interactions between different applications. Additionally, since the granularity of control for a checking system is coarse-grained (i.e., on the level of a complete virtual machine instead of a process), an attacker might control the complete virtual machine.

3.5 Static program verification

The Google Native Client [YSD+09] executes x86-code in a sandbox. The native client uses the same instruction padding techniques as presented in the SFI system by Wahbe et al. [WLAG93]. The instruction set is limited to a safe subset of the IA32 Instruction Set Architecture (ISA), making illegal operations impossible. A verifier checks if the program is valid before the program is executed without any additional isolation. Such a system limits the possible range of used instructions, the programs must be linked statically, and no dynamic libraries can be used. Programs must be compiled with a custom-tailored compiler and special libraries.

PittSField [MM06] implements a static binary translation and checking tool used for SFI. The static rewriting algorithm (i) aligns targets for control transfers on 16 byte boundaries, (ii) changes control transfer instructions so that targets are always 16 byte aligned, and (iii) separates data and code regions by adding additional instructions to force pointers to point to data or code. This static translation results in moderate overhead of 13% for instruction alignment and 21% for data verification [MM06].

3.5.1 Control flow integrity

CFI [ABEL05] restricts a running application to a predefined Control Flow Graph (CFG). A binary rewriter uses a whole-program analysis to statically detect the CFG. This CFG is then enforced for all control flow transfers. The binary rewriter adds dynamic checks to all dynamic control flow transfers using a canary concept where the source location checks for a canary at the target location.

CFI protects against all possible exploits that redirect the control flow to alternate locations. This approach requires that all possible target addresses are known at translation time. The source and target locations are interlocked and any deviation will stop the program. CFI is also combined with a shadow stack that verifies the return addresses of the application. The shadow stack is in the same address space as the program but only verified code is allowed to push and pop data from the stack.

A drawback of the CFI approach is the knowledge about which locations are reachable for dynamic control flow transfers. E.g., for jump tables or an array of function pointers it is hard to specify the correct bounds using only information that is available in the static binary. An additional problem is that CFI uses whole-program analysis. Whole-program analysis requires that all libraries are available at translation time, and the translation will produce an *unmodifiable static binary*. The CFI-protected binary is unable to load dynamic code, e.g., shared libraries or plugins. Another drawback of CFI is potential code duplication. If some code is a potential target from multiple sources then either all sources must share the same canary (which reduces security or the effectiveness of CFI) or the binary rewriter must emit duplicate code with different canaries for each source location.

An alternative to CFI is a dynamic approach using a dynamic binary translator to enforce CFI. This dynamic approach is not limited to statically identifiable target locations as it discovers new targets on the fly and can adapt to newly loaded code. An integration with the dynamic loader also enables the use of address space layout randomization of the different shared libraries that is not possible for statically compiled binaries.

3.5.2 Hacking control flow integrity

CFI protects from malicious changes of the control flow using specific static canaries that are checked whenever a dynamic control flow is dispatched. These canaries form a contract between the source and the target location.

A possible attack that circumvents CFI exploits the static behavior of CFI. Prerequisites to break CFI are control over a register for a dynamic control flow transfer, an attack vector in the application that results in a random memory read and a random memory write, and either a call to `mprotect` or a writeable code region (e.g., modifyable code of the dynamic linker, or a page sharing code and writeable data).

The CFI paper assumes non-executable data but the application includes the `mprotect` system call (if the standard libc library is used in the program). `Mprotect` can be used to circumvent the non-executable data assumption.

The random memory read is used to read the canary. The canary and the shellcode are then written to either the page with shared code and data or to a data page. If the page has no execute permissions then the parameters to an `mprotect` call must be forged to toggle the executable bit of the target page. At last the target of the control flow transfer is changed to the newly generated valid target consisting of the correct canary and the injected shellcode. The static control flow integrity check verifies that the static canary matches (but not the actual target location) and allows the control flow transfer. This attack breaks CFI.

3.5.3 XFI

XFI [EAV+06] is an extension of control flow integrity intended for kernels or kernel drivers. CFI is combined with an extended shadow stack, a memory access checker, and a verifier. Whenever a new XFI-protected module is loaded the verifier checks that the code contains all the necessary guards. CFI and the shadow stack are used to protect from any control flow-based attacks. These guards ensure that the control flow of the kernel module cannot deviate from the statically defined CFG.

The memory access checker uses additional guards for all memory reads and writes. This guard ensures that all accesses are in well-defined regions, e.g., writes are only allowed to the heap or to the stack of the program. This memory access checker comes at a cost: every group of memory accessing instructions using the same register or the same pointer is rewritten using an additional bounds check. Instead of a simple memory access there are multiple compare instructions and a jump.

XFI solves some problems of CFI but shares similar drawbacks. All protected code must be static and the complete CFG must be known during the translation. In addition, the memory access checker relies on a static setup of all memory regions. The static setup inhibits dynamic loadable code, relocatable code, and features like address space layout randomization.

3.5.4 Hacking XFI

The strict programming model that XFI ensures removes all threats from code based attack vectors. The control flow cannot be diverted to alternate code and no new code can be injected into the memory image of the module. In addition, potential attack vectors can only change the data of the application but not the code.

Return-oriented attacks or jump-oriented attacks are limited to the set of already available locations. An attack may still change the register or pointer with the target location but the target must match a predefined target. Attacks can also write

to the heap and to the stack. Random writes can be used to change parameters of legitimate calls and system calls. For example, a `switch` statement that switches between different functions (e.g., system calls) could be exploited to execute a different function out of the set with adapted parameters, or - in a user-space context - the parameters of an `execve` system call could be swapped to some malicious executable. This attack also applies to CFI.

3.6 Secure compiler extensions

Secure compiler extensions use the available type information or program structure to add security checks to the generated code. Additional runtime checks are added for, e.g., format strings [CBB+01], memory allocations [CBJW03], and checks for return instructions [CPM+98,BST00,HK01]. WIT [ACR+08] uses points-to analysis at compile time to generate a CFG of the application and sets of possible locations for each memory write. The effectiveness of WIT is limited by the completeness of the points-to analysis. The CFG and write set is enforced at runtime using additional checks. Byte-Granularity Isolation (BGI) [CCM+09] is a compiler extension that implements multiple protection domains in a single address space. Every byte belongs to a module and can only be accessed by a subset of all modules. The compiler adds additional write checks to enforce the write integrity. LXFI [MCZ+11] implements SFI and kernel module separation by embedding additional checks into the compiler generated code.

A naive version emits runtime checks in all cases; static verification in the compiler can reduce the amount of necessary runtime checks. Each of these systems removes one attack vector or reduces the probability of one form of attacks. The drawback of these approaches is that they have only a partial picture of the runtime image of the application. They concentrate only on one attack form and leave other attack vectors open.

3.7 Summary of different protection techniques

The research on a secure execution platform builds on ideas from different fields of research. Security can be enforced at different levels of granularity and through different approaches. This section summarizes approaches that are similar to SFI as well as SFI itself.

The basic SFI framework must be fast, extensible, and secure. Many different instrumentation frameworks exist, and one must be aware of the limitations that several optimizations pose to security.

Apart from user-space isolation there exist other possibilities to secure running systems. Dynamic systems add additional guards and checks to a running application. These systems all work at different levels of granularity. Full system translation

encapsulates a complete running system and works at a coarse-grained level of granularity; system call interposition encapsulates the application at the system call level and works at the granularity level of individual applications and their system calls.

Static protection reduces the potential overhead but either restricts the instruction set or introduces complicated static analysis. Static verification allows only a sub-set of the instruction set or imposes other additional checks. Compiler extensions can be used as a quick fix to patch a specific static problem.

Table 3.1 presents a summary of related work and distinguishes features, design and implementation details of these different approaches.

Product/Feature	Techniques used[a]	System call interposition	System call policies	Full ISA supported	Completely transparent translation	Stack exploit protection (ret2libc)	Control flow integrity	No special kernel-module needed	No application changes needed	Separate secure stack for monitor[b]	Separate shadow stack for application	TOCTTOU-aware using safe-guards	Source code available
libdetox [PG11]	1	x	x	x	x	x	x	x	x	x	x	x	x
basic BT [PG09, PG10]	1	x	x	x	(x)	(x)	(x)	x	x			(x)	x
Vx32 [FC08]	1	x		[c]				x					x
Strata [SD01, SD02]	1	x		?	?	?		x	x				
Prog. sheph. [KBA02]	1	x	(x)[d]	x	x	(x)[e]	(x)	x	x	x			
Janus [GWTB96]	3	x	x		x			x	x				x
AppArmor [Bau06][f]	3	x	x		x				x				x
SysTrace [Pro03]	3	x	x		x				x				x
Switchblade [FS08]	3	x	x										
Capsicum [WALK10]	3	x	x										
Ostia [GPR04]	3	x	x										
NaCl [YSD+09]	2					x		x					x
PittSField [MM06]	2					x		x					x
CFI/XFI [ABEL05, EAV+06]	2			[g]		x	x	x			x		
StackGuard [CPM+98]	4					x		x	(x)				x
libverify [BST00]	4					x		x	(x)				x
Propolice [HK01]	4					x		x	(x)				x
PointGuard [CBJW03]	4							x	(x)				(x)
WIT [ACR+08]	4					x	x	x	x				

Possible feature classifications: x marks an existing feature, (x) marks a limited feature, a blank marks a missing feature, a ? indicates that not enough information is available.

[a] 1: dynamic BT; 2: static BT; 3: kernel module or kernel support; 4: compiler extension

[b] The monitor uses a separate stack to, e.g., check permissions or translate new code.

[c] Only a subset of IA32 ISA is implemented, FPU, MMX, SSE, and 3 byte opcodes are missing.

[d] Static hard-coded policy in implementation, only open and execve system calls are intercepted.

[e] Return instruction may only target instructions immediately after any call instruction.

[f] MAPbox [AR00], SubDomain [CBKH+00], and Consh [AKS99] use a comparable approach.

[g] According to the paper at least FPU, MMX, and SSE extensions are missing.

Table 3.1: Summary of related work.

4

Design and security guidelines

This chapter explains the design details and security implications of our secure execution platform. We call the protection offered by the secure execution platform *user-space security without sacrifices*. The proposed model implements multiple layers of security that build on each other.

Figure 4.1 shows the different components of the secure execution platform. The three layers are (i) a secure loader that is part of the trusted computing platform and that loads application and libraries, (ii) a dynamic Software-based Fault Isolation (SFI) layer that controls all executed code and enforces dynamic Control Flow Integrity (CFI), and (iii) a policy-based system call authorization platform that checks every system call of the application. These three layers form a sandbox in which all application code is executed.

Our secure loader offers the foundation for the secure execution platform. The secure loader enables a novel model for program creation. Instead of using the standard Linux loader, we use a secure loader replacement that is part of the trusted computing base. This loader offers the same core functionality as the standard loader but is security hardened and tightly integrated into the libdetox security framework (the code execution sandbox). The secure loader ensures that no application code

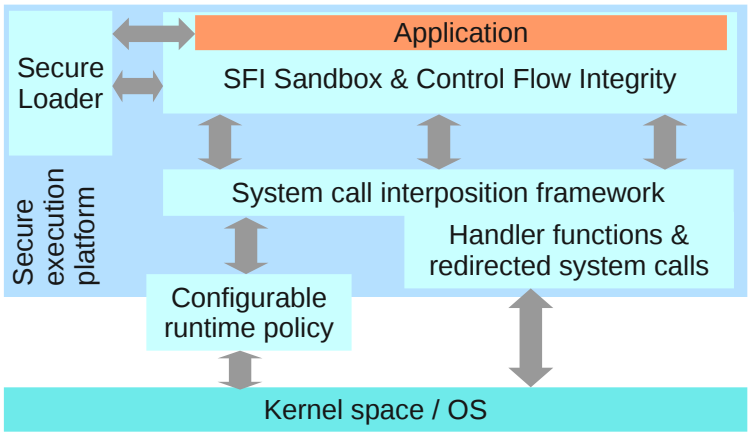

Figure 4.1: Component-based overview of the secure execution platform.

can be executed before the secure sandbox is set up and enables secure loading of libraries and secure lazy binding of library functions.

All application code (including any dynamically loaded shared libraries) is executed under the control of a user-space virtualization environment. A small online dynamic binary rewriting toolkit implements SFI and ensures that programs cannot escape out of the virtual machine. Special guards in the SFI layer ensure non-executable data regions and non-writable code regions. Another advantage of dynamic SFI is that all direct and indirect control flow transfers can be checked dynamically. These dynamic checks enable on-the-fly verification for all control flow targets (dynamic CFI). The bare SFI layer ensures that all executed code is checked and that the security guards are added before the code is executed.

We use the tight coupling between the loader and the application to extract a detailed model of all control flow transfers. This model is then enforced using dynamic control flow checks. These checks implement control flow integrity and prohibit code-based attacks, return-oriented programming, and jump-oriented programming.

Process sandboxing through dynamic binary translation is the first line of defense and ensures that (i) no injected code is translated or executed, (ii) application or library code cannot be overwritten, (iii) all control flow transfers must adhere to a predefined control flow graph, (iv) program code cannot escape or terminate the sandbox, and (v) program code cannot overwrite any data structures owned by the sandbox. Features (i) and (ii) are enforced by the non-executable data and non-writable code security concepts. Feature (iii) implements control flow integrity. The binary translation framework translates all code before it is executed and uses special guards to ensure that the code conforms to the security specification and that, e.g., control transfers always target valid code. For performance reasons, target addresses of individual translated instructions are not checked. This second layer enables dynamic control flow integrity. Dynamic CFI is combined with non-executable data regions and non-writable code regions.

A third layer of protection redirects all system calls to a system call authorization framework that checks the system calls and their parameters according to a tight user-defined per-application policy. This last line of defense protects from possible data exploits (e.g., integer overflows, type errors, format string attacks) where parameters of system calls are changed to malicious values.

The remainder of this chapter is organized along the three layers that build up our security concept. Section 4.1 describes the design decisions and features of the secure loader that is used to bootstrap the secure sandbox and the application. Section 4.2 explains the basic SFI layer that translates all code on the fly. Section 4.3 introduces a method to generate dynamic control flow graphs. These control flow models are then used to implement dynamic CFI, and Section 4.4 describes how the model for dynamic CFI is generated using readily available Executable and Linkable Format (ELF) information and symbol tables. Section 4.5 introduces policy-based system call interposition that checks all system calls in user-space.

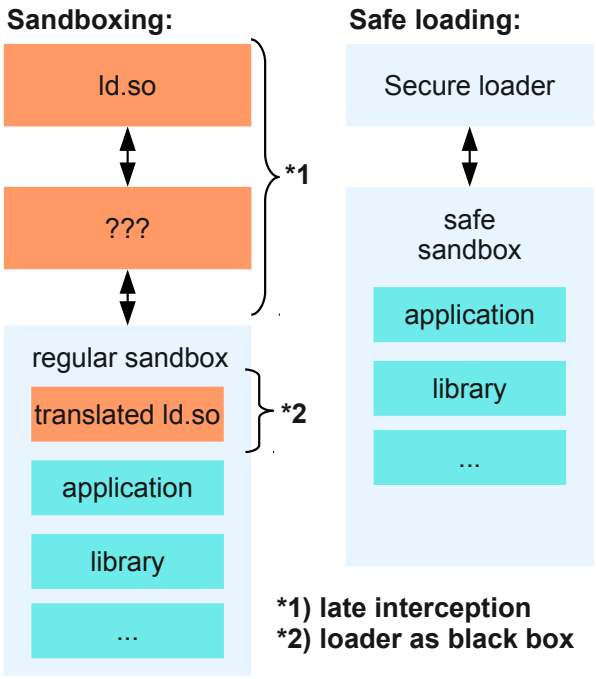

Figure 4.2: Comparison between a regular sandboxing approach and safe loading as provided a secure loader. The left-hand side shows two problems: 1) the late interception problem and 2) the loader black box problem.

4.1 Safe loading in a trusted runtime environment

The secure loader presents a new model for process creation (turning an executable and all associated libraries into a running program) and is the first technique that takes complete control over an application in user-space. The standard dynamic loader is replaced by a secure loader that is part of the sandbox domain. As a result the application domain no longer needs the permissions to map executable code. A secure loader must be safe (the implementation is reviewed and bug-free) and secure (the design does not provide attack vectors in the offered functionality). This approach bridges programming languages and operating systems: a language-independent loader is used to secure and confine binary-only applications in their execution running on an operating system. Figure 4.2 provides a comparison between the standard runtime environment and the secure execution platform.

The secure loader runs as part of the trusted computing base. The secure loader is the only entity that is allowed to load new code, and the application is only allowed to access loader functions through an API. The sandbox and the loader are tightly coupled and share information about the program. The loader analyzes segment and section information of the application and all dynamically loaded objects and enables per object privileges. The loader resolves objects, symbols, and relocations for the sandbox that then embeds resolved addresses in the translated code.

The tight coupling of the loader and the sandbox enables module separation. Control transfers between modules are inlined directly into the translated code. The translated source object contains a direct reference (that is unreadable from the application) to the target object and no call through the Procedure Linkage Table (PLT) and Global Offset Table (GOT) is needed. The PLT data structure is only kept for reference reasons.

The secure loader replaces the standard loader[1] and solves the problems of the standard loader that are discussed in Section 2.4. The secure loader ensures that the SFI library is initialized first and treated specially so that symbols are neither added to the global scope nor accessible through any API functions.

4.1.1 The sandbox

The secure execution platform defines two privileged domains in user-space. The *sandbox domain* a trusted domain that contains the trusted computing base (the loader and the sandbox) and the *application domain* an untrusted domain that contains the application code and all needed libraries.

The sandbox domain ensures that no unchecked code is executed in the application domain. Application code is examined by the sandbox before execution, and additional security guards are added to ensure that the executed code cannot escape out of the sandbox.

Binary translation is a key component for user-space SFI. A dynamic translation system translates and checks every machine code instruction before it is executed. Translated code is placed in a code cache. Every direct control transfer is resolved during the translation. Every indirect control transfer is intercepted at runtime and only translated branch targets are reached. The translator can change, adapt, or remove any invalid instruction and can intercept system calls before they are executed.

An important requirement of the sandbox is that return addresses on the stack remain unchanged. This requirement adds additional complexity when handling return instructions, as they are translated to a lookup in the mapping table[2] and an indirect control transfer. On the other hand, an unchanged stack ensures that the original program can use the return instruction pointer on the stack for (i) exception management, (ii) debugging, and (iii) return trampolines. Additionally, the user program does not know and cannot discover that it runs in a sandboxed environment, and the address of the code cache is only known by the binary translator.

[1] Usage of the secure loader is encoded on a per-application basis in the ELF information of the application itself.

[2] The mapping table is a sandbox-internal data structure that relates translated and untranslated code.

4.1.2 Solving the loader's security problem

Combining a secure loader and a safe sandbox to a trusted execution environment solves the problems from Section 2.4. The loader must be separated from the application, and the application may not access internal data structures directly. The privileged sandbox domain is an optimal location for the secure loader. The secure loader and the sandbox share information about all loaded shared objects and symbols. The shared information enables the sandbox to restrict control flow transfers between shared objects.

Restricting privilege escalation attacks

The secure loader implements a subset of the features of the standard loader. The subset is complete enough to run in practice any programs compiled with a recent version of the compiler toolchain.

The secure loader protects privileged programs (e.g., programs running with "Set User ID upon execution (SUID)" privileges) and programs accessible from the Internet (e.g., a web server). The secure loader does not read any environment variables and has no user-configurable settings. Library paths, debugging features, and loader settings can only be changed by modifying the source code. The secure loader does not allow any changes in the production environment.

The removal of these user-settable features protects from attacks mentioned in Section 2.4.1. Privileged applications do not need these features, and hence removing the features altogether is more secure than executing additional checks before accessing the features (as done by the standard loader).

Protecting all executed application code

The initialization code of the secure loader is the first code that runs when an application is started. This initialization code initializes the sandbox as well.

The secure loader can execute all application code under the control of the sandbox because the loader is part of the privileged sandbox domain. The secure loader tells the sandbox to translate an entry point to application code whenever the standard loader would pass control to application code. Whenever the application requests an action from the loader (e.g., resolving symbols, loading additional modules, or loading PLT entries) the secure loader switches into the sandbox domain, verifies the correctness of the request, and returns the result to the application domain. The secure loader cleans all references to internal data from any returned structures. The application uses the loader features through a well-defined API and can no longer read or write internal loader data.

This procedure ensures the safety of the secure loader, the sandbox, and the internal data structures at all times. Consequently the problems mentioned in Section 2.4.2 do not exist for the secure loader.

Opening the loader black box

Placing the loader in the sandbox domain solves the loader black box problem from Section 2.4.3. The sandbox and the loader are in the same trust domain and form the trusted computing base. Loader and sandbox can share data structures and exchange information about executable code regions, data regions, and symbol locations.

The loader is no longer translated by the sandbox as a part of the application but is an integral part of the sandbox. The application no longer needs privileges to map executable code into the application memory space but transparently uses the loader API provided by the sandbox. All applications remain unchanged but calls to the loader are redirected to the secure loader API in the sandbox domain. Applications that do not use the standard loader to load modules and executable code are not supported[3].

4.1.3 PLT inlining

The tight integration of the secure loader into the trusted computing base enables PLT inlining. In traditional systems, the PLT is used to enable position independent code for shared libraries. The binary translator in the sandbox can remove the PLT code and inline the resolved target addresses directly into the generated code.

This optimization reduces the number of indirect control flow transfers (these control flow transfers account for the main overhead in dynamic binary translation) and hides the location of other objects from the application.

The addresses are encoded directly in the code cache and the application has no access to the instructions in the code cache. This feature enables module separation and raises the bar for security exploits because a potential exploit is unable to determine the locations of specific functions in other objects. Finally, the total number of indirect control flow transfers is reduced, limiting the opportunities for jump-oriented attacks.

4.2 Software-based fault isolation layer

The dynamic translation system uses binary translation to check every machine code instruction *before* it is executed. Every direct control transfer is translated, and every indirect control transfer is intercepted, and only translated branch targets are reached. The translator can change, adapt, or remove any invalid instruction and can intercept system calls *before* they are executed. During the translation process code can be instrumented and augmented with additional security features.

[3]Hard-coded exceptions in the sandbox domain can be used to support applications that use a non-standard way to load executable code. We are not aware of applications that need such exceptions.

Security features like non-executable stack, stack guards, control flow evaluation, or argument checking for specific functions are added without the need to recompile the application. Even patches can be applied at runtime to fix bugs in running applications. Fault isolation detects code injections and terminates the program before the data structures are corrupted. Data-based exploits, on the other hand, are not detectable by fault isolation and are caught using policy-based system call interposition.

The binary translator is the foundation of the security guarantees of the presented sandbox. The binary translator should be modular and small to keep the trusted computing base small. This section presents design criteria for a modular and flexible binary translation fault isolation layer that is needed to implement the additional security guards and *system call authorization*. An important feature of the binary translator is that return addresses on the stack remain unchanged. This adds additional complexity when handling return instructions as they are translated to a lookup and an indirect control transfer. On the other hand, an unchanged stack ensures that the original program can use the return instruction pointer on the stack for (i) exception management, (ii) debugging, and (iii) return trampolines. Additionally, the user program does not know that it runs in a virtualized environment, and the address of the code cache is only known by the binary translator.

4.2.1 SFI security guards

Binary translation guarantees that only translated instructions will be executed but does not prevent individual instructions from overwriting memory regions or executing specific system calls. Dynamic binary translation enables the implementation of additional security guards by rewriting and encapsulating specific instructions.

An important requirement of the binary translator is that no pointers to internal data structures are left on the stack. The binary translator therefore uses a hidden stack in the sandbox domain that is separate from the application stack of the translated user-program to dynamically translate new basic blocks and dispatch indirect control transfers.

A custom tailored exploit could target the binary translator itself. If the program were able to locate the internal data structures of the binary translator (e.g., the code cache), it could modify the executed code by directly changing instructions in the code cache and so break out of the isolation layer. Therefore any pointers to internal data are only allowed on the secure stack, and no pointers are pushed to the application stack. Additionally, the translator guarantees that application code that tries to access internal data structures is not translated by virtualizing, e.g., addresses or registers.

The basic binary translator is extended by the following security guards that harden the user-space isolation sandbox and to ensure that application code cannot escape out of the sandbox:

Non-executable data: implements a form of executable space protection for x86 on a section basis in user-space. This protection enforces the permissions for regions defined in ELF headers of the programs and loaded libraries, even if they are smaller than a page. The guard checks if the target area is defined in the program or an imported library and if it actually contains code. If there is a violation then the program is terminated. This guard ensures that data regions cannot be executed and protects against the execution of code injected through stack-based and heap-based buffer overflows.

Executable bit removal: this guard marks the code of the untranslated program as *non-executable* (by using `mprotect` calls). The application does not know the location of the internal code cache and therefore cannot overwrite parts of the code cache with injected code. This guard ensures that only code from the binary translator and translated code in the code cache can be executed.

Return address verification: This guard checks that the return address is not modified by implementing a *shadow stack* that is only accessible in the protected sandbox domain where the return address is verified for each return instruction. The shadow stack contains pairs of addresses, the original location and the translated counterpart. If the address on the original stack does not correspond to the address on the shadow stack then the program is terminated. This guard protects against stack-based overflows and return to libc attacks and is comparable to solutions like StackGuard [CPM+98], libverify [BST00], Propolice [HK01], and FormatGuard [CBB+01].

Signal handling: the binary translator keeps a mask of installed signals on a per-thread basis. Whenever a new handler for a specific signal is installed, the code of the handler is wrapped into a trampoline that guarantees the secure execution of the signal and ensures that the signal processing code cannot escape the binary translator. This guard protects against errors in the trap handling and enables the sandbox to catch memory accesses to unmapped memory regions (e.g., probing for the location of the code cache).

Secure context transfers: the control transfer from the binary translator to translated code uses an indirect control flow transfer to ensure that no pointers to any internal data structures of the binary translator are exposed on the stack. This guard hides the internal data structures of the sandbox from the application.

Address Space Layout Randomization (ASLR): the binary translator allocates all internal data structures on random addresses using an internal `mmap` implementation. On IA32 the instruction pointer cannot be read directly (e.g, through a register) and indirect control transfers (e.g., `call` instructions) are replaced by a secure sequence of virtualized instructions during the translation process. All translated indirect control flow transfers point into the original code region and are replaced with a lookup in the mapping table. The

translated code is therefore unable to recover a pointer into the code cache or any other internal data structure of the binary translator. The internal `mmap` implementation uses the ASLR feature that is available in the Linux kernel [PT03, BDS03, BBSD05]. The probabilistic security guarantees of ASLR are limited if ASLR is used in isolation [SPP+04].

Dedicated sandbox stack: all code executed in the sandbox domain uses a dedicated secure stack. Application code has no access to the sandbox stack. Execution switches to the sandbox stack whenever sandbox code is executed. A dedicated stack helps separating the application domain and the sandbox domain.

Protection of internal data structures: the sandbox adds an additional heavyweight guard that uses `mprotect` to disable write access to all internal data structures (e.g., mapping table, code cache, internal translator data) of the binary translator whenever translated code is executed. This way translated code is unable to change translated code or any other internal data structures of the binary translator. Even if this guard is not active, all pointers to the internal data structures are pruned from the stack. An exploit is unable to detect the internal data structures due to the virtualization guarantees of the translator.

The current guard configuration does not allow applications with self-modifying code. An application with, e.g., a JIT compiler could be handled by specifying ahead of time exactly which regions the compiler will be emitting code and using an additional guard for these regions; see Section 5.3.5 for more details.

4.3 Dynamic Control Flow Integrity (CFI)

Control flow transfers must transfer to valid targets as specified by the application. No location that is not pre-existent in the application must be reached. Control flow transfers must conform to a pre-existing Control Flow Graph (CFG). This CFG defines an unmodifiable set of reachable targets for each source location.

As long as the control flow transfers correlate with the pre-defined CFG no code-based attack took place. If the control flow of the application deviates from the CFG then an exploit happened or is about to happen and the program must be stopped.

A dynamic instrumentation framework is in the unique position to check all control flow transfers before they are executed. For each control flow transfer source location a set of valid target locations is defined. The target array is then used in the check; if the current target location is in the array then the transfer is valid and can be executed, otherwise the program is terminated with a security exception.

The dynamic translator checks static control flow transfers (e.g., `jmp` and `call` instructions) at translation time. For dynamic control flow transfers (e.g., `jmp in-`

`direct` and `call indirect` instructions) the dynamic translator emits a trampoline that checks the pair of source and target location whenever the control flow transfer is executed. The dynamic control flow transfers result in additional overhead for each execution. This verification overhead can be reduced by using, e.g., local lookup caches.

4.3.1 Comparing dynamic and static CFI

Dynamic CFI shares one limitation with static CFI (see Section 3.5.2 and Section 3.5.4), namely that a possible attack that controls a register can change the control flow to an alternate location. This alternate location must also be in the set of valid target locations for the source control flow transfer. This possible attack vector can be used to set up data attacks. Static CFI/XFI is unable to detect these data attacks. To be effective a data attack must change parameters of a system call at one point in time. These changed system calls (and arguments) can then be used to escalate privileges, or to launch additional attacks. Our framework includes a policy-based system call authorization that detects data-based attacks in system calls and terminates the application (see Section 4.5).

Dynamic CFI has two advantages compared to static CFI. First, ASLR is still possible. ASLR enables random placement of code and data in the process image, making it harder for an exploit to discover valid alternate target addresses. Second, dynamic loading of libraries and lazy binding is also possible. These two facts enable faster startup times because not all code must be relocated and loaded when the application starts.

4.4 Model generation for dynamic CFI

Control flow integrity relies on an exact CFG that specifies every possible control flow transfer. This CFG can be generated at different stages during the compilation of the program. Compilers have complete information of all possible targets for each location. A compiler can be extended to emit a CFG for each compiled module along with the binary or assembly code that can be used for CFI (see "Future Directions" in Section 8.1). If the source code is not available then a possible alternative is to reverse engineer the CFG based on the available binary (as done by CFI/XFI [ABEL05, EAV+06]). The reverse engineering approach has the limitation that not all valid locations can be located statically (e.g., the size of static jump tables is not known, dynamically computed and populated jump tables cannot be statically detected, and the target address can be the result of complex computation.

We generate a model using an alternate approach. By reusing already available information in the ELF headers of the application and the libraries we generate a detailed execution model for an application which is basically a CFG of the complete program. Appendix B describes the ELF format. The symbol table and the dynamic

symbol table define all symbols (including name, location, and size) of a shared object. The dynamic symbol table is a subset of the symbol table and contains only imported and exported symbols.

Stripped objects only contain the dynamic symbol table and remove the symbol table. This less fine-grained information reduces the accuracy of our model. For maximum effectiveness of our model we assume that the shared objects are not stripped and that both the symbol table and the dynamic symbol table are available.

The information about imported and exported symbols is used to implement a CFI guard for function calls, while the information regarding the location and size of individual functions is used to implement a section guard for jump instructions.

Section guard: only (direct and indirect) function calls and function returns are allowed to transfer control to a function in a different code region. All other control transfers (like jumps or indirect jumps) must target code in the same function. This guard prohibits unintended control flow transfers.

Call guard: call instructions are verified to transfer control to an existing function (symbol) that is either imported by the current object and exported by the target object or defined in the same object. If the call does not target a symbol defined in any of the loaded libraries or the program itself then the call is not allowed. This guard prohibits ROP attacks and the redirection of function pointers to unintended code.

The SFI sandbox checks all control flow transfers and ensures that they conform to the constructed runtime model. Static control flow transfers (e.g., direct jumps, conditional jumps, direct calls) are checked when the code is translated. No additional check is needed during the execution of the translated code (check once, execute often). Dynamic control flow transfers (e.g., indirect jumps, indirect calls, and return instructions) are translated into a runtime check. A control flow transfer check is executed every time before the dynamic control flow transfer is executed. Figure 4.3 shows a flow diagram for inter-module function calls and inner-module function calls. These secure control flow transfers protect from undesired control flow transfers (ROP attacks and code injection).

4.4.1 Inter-module function calls

Inter-module calls describe transitions from a method in one object to a method in another object. Our execution model allows only call instructions to transfer control between modules; jump instructions must stay inside the module.

Inter-module transfer checks ensure that the instruction is a call instruction and that the target of the instruction is reachable from the current location according to the fine-grained execution model. The target location must be imported in the current object and the symbol must be exported in the target object. Failed checks terminate the program.

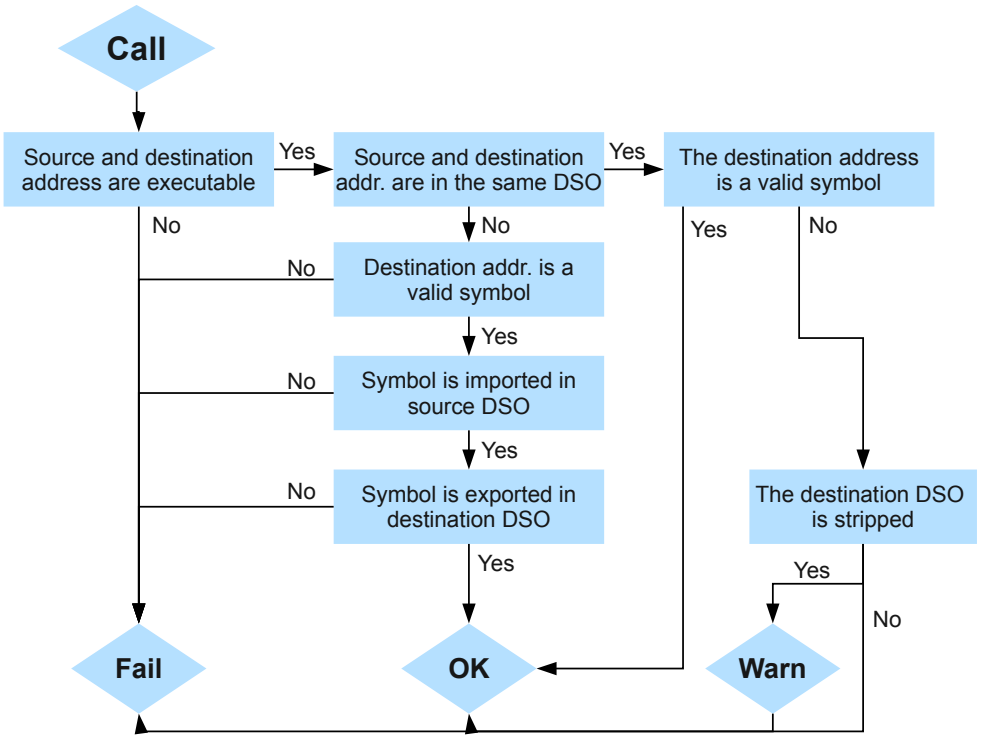

Figure 4.3: Control flow transfer check for call instructions. A DSO is a shared library.

4.4.2 Intra-module function calls

Inner-module transfers describe transitions inside from one method to another method in the same object.

Inner-module transfer checks ensure that the instruction is a call instruction and that the target of the instruction is a function symbol inside the current object according to the fine-grained execution model. If the current object has no complete symbol table information (but only the dynamic symbol table of exported symbols) and the target is not an exported symbol then a warning is emitted because our approach is unable to detect inner-module method boundaries. The application is terminated if the destination address is not in an executable section. If symbol tables are available and the check fails then the program is terminated.

4.4.3 Jump instructions

Jump instructions (direct, indirect, and conditional) are allowed to transfer control inside a method. If jump instructions transfer control into another object or method then the program is terminated. Figure 4.4 shows the control flow transfer check.

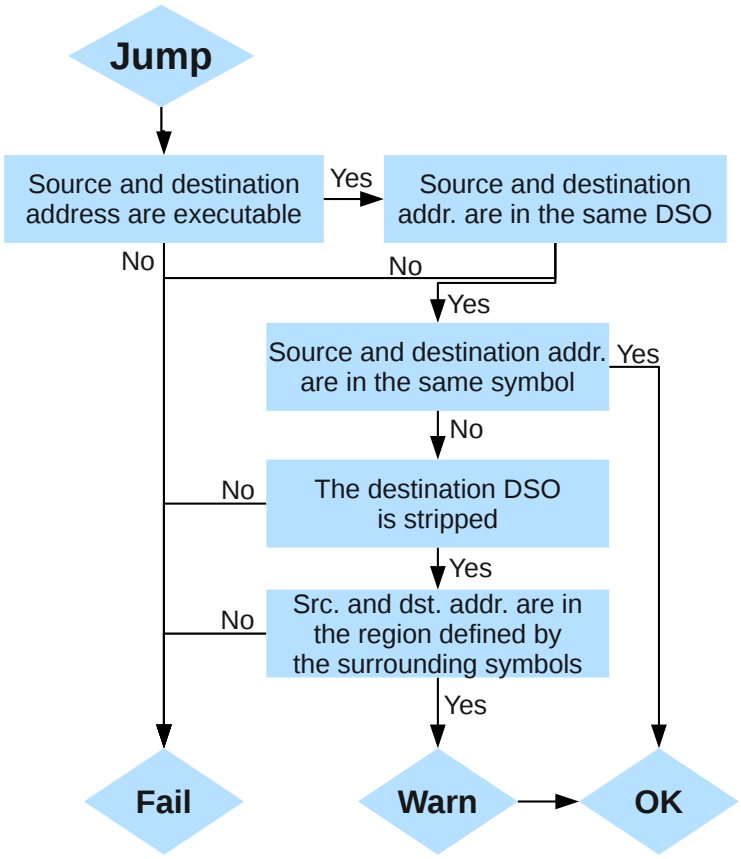

Figure 4.4: Control flow transfer check for jump instructions. A DSO is a shared library.

If the shared object is stripped (and the resolution of function boundaries is limited) then the control flow transfer check is limited to the boundaries of outer exported functions or, more generally, the defined executable region.

This ensures secure control flow transfers but prohibits the use of functions like `setjmp`, `longjmp`, and `goto` between functions. Failed checks terminate the application.

4.4.4 Return instructions

Return instructions are secured using a shadow stack; this stack is in the protected domain and not accessible from application code. The shadow stack contains information about call frames (return instruction pointer, location of the return instruction pointer on the original stack, and translated instruction pointer in the code cache). Return instructions are translated into a check that ensures that the return instruction pointer on the stack is equal to the return instruction pointer on

the shadow stack. If these two pointers are equal then control is transferred to the saved location on the shadow stack.

This ensures that stack overflows are unable to change the execution flow of the application. The program is terminated if an illegal control flow transfer is detected.

4.4.5 Limitations of our model

A complete CFG knows the exact set of target locations for each instruction that changes the control flow, whereas our model is less exact to enable dynamic computation of function call targets and jump targets. The control flow transfer checks validate every single control flow transfer but are limited by the breadth and exactness of the model. See Section 5.4.2 for a discussion of possible limitations in the current implementation due to potentially missing information, e.g., in the ELF headers.

A call instruction can target any imported function that is exported in another shared object, and all functions in the current object. The number of imported symbols in a shared object is usually small. A potential exploit based on the redirection of function calls can target any valid function that is imported or defined in the source shared object. Control transfers to other functions must always target the beginning of the function. It is not possible to transfer control somewhere into a function. This guard limits the potential uses of function call redirection attacks to alternate parameters for existing imported functions. All other possible redirects are detected by the control flow transfer checks and the faulty application is terminated.

Jump instructions can target any valid instruction inside the same symbol. A jump instruction can target any valid instructions of a function. Jump instructions cannot transfer to or into a different symbol. This limits gadget search for, e.g., jump-oriented programming to the current function. For a potential exploit to be successful the function would need to include a dispatcher gadget and all useful gadgets to carry out the exploit. CFI/XFI has a similar problem with, e.g., large switch statements that can be used for a jump-oriented programming attack. The probability that such an exploit can be carried out is low. The dynamic translator can flag a warning if an application contains code that would allow such an attack (e.g., by counting the number of indirect control flow transfers in a function correlated with the number of individual jump targets).

On the other hand, it is an advantage of this model that not all possible targets for a source location have to be known during the translation or at startup.

4.5 Policy-based system call interposition

All the potentially dangerous functionality of a program is performed by system calls (e.g., I/O, network sockets, privilege escalation). A mechanism that restricts

a program's use of system calls is a useful and important extension to fault isolation and dynamic CFI. Code-based exploits are handled by the SFI layer. Data driven attacks where no malicious code is executed (e.g., integer overflows and type errors) are caught whenever a system call is executed that does not conform to the application's policy.

Policy-based system call interposition relies on *SFI* and the rewriting and replacement of system calls. All system calls through `sysenter`, and `int 80` instructions (Linux uses and supports both schemes [Gar06]) are rewritten by the binary translator to execute a validation function before the system calls are allowed. The sandbox offers an extensible *system call interposition framework* that makes it possible to allow or disallow system calls based on the call stack, the system call number, and the parameters.

The sandbox validates system calls through handler functions and by a policy that is loadable at runtime. A policy has the advantage that combinations of allowed and disallowed parameters can be specified in a simple way. Handler functions, on the other hand, enable in-depth verification of arguments and can use state (e.g., a list of previous `mmap` calls, arguments, and call locations) to track application behavior throughout the execution of different system calls. The combination of a policy to handle simple and static combinations of system calls and handler functions for complex system calls enables an even tighter and more dynamic security model than policies alone.

4.5.1 Special handler functions

The privileges of a program can be managed by specific handler functions on a per system call basis. Every system call can use a different handler function that analyzes *call stack* and *arguments*. The handler functions are a part of the sandbox domain and have full control over the application. Handler functions may *allow* the system call, *abort* the program, or redirect the system call and return a *fake value*.

These redirected system calls can be used to implement different functionality in user-space. If a system call is redirected then a user-space function is executed whenever the system call is called. This function runs in the context of the sandbox and can execute arbitrary other system calls (redirected system calls can, e.g., emulate or isolate vulnerable system calls). More generally redirected system calls add additional validation of arguments that are passed to the kernel.

The sandbox uses additional handler functions to check all `mmap`, `mprotect`, `open`, and `openat` system calls. For `mmap` and `mprotect` the sandbox checks if the arguments overlap or touch any internal data structures that the binary translator uses. If there is a conflict then the application is terminated. For the `open` and `openat` system calls the sandbox uses `stat` to check if the file is in the black list or tries to access protected files like `/prof/self/maps` that would leak information about the sandbox.

```
mode:whitelist /* deny unlisted syscalls */
open("/dev/arandom", O_RDONLY):allow
open("/dev/urandom", O_RDONLY):allow
time(null):allow
getuid32():return(0) /* return static uid=root */
close(*):allow /* close open files */
write(1,*,*):allow /* stdout */
access("/etc/*",*):allow
// implicit: access("*",*):deny
```

Listing 4.1: An example of a small white-listing policy

The handler functions are used as an extension of the policy system. Handler functions enable additional control logic to guard and tighten the allowed actions of the application.

4.5.2 Policy-based system call authorization

The *system call authorization framework* for policy-based system call authorization builds on *process sandboxing* and extends the system call interposition framework. The sandbox loads a user-defined per-application policy at startup to decide for each individual system call if it is allowed or not.

If a system call and its arguments do not match the policy (e.g., the configuration is not present in the policy for white-listing, or is present for black-listing), the isolation system assumes that there is an error, bug, or security problem and terminates the user process. Additionally, it can signal the system operator that an authorization fault occurred.

The policy file contains a list of system calls and parameter-sets that are allowed or denied. This setup allows a combination of white-listing and black-listing of different argument combinations per system call. Arguments are encoded as integers, pointers, strings, paths, null, or asterisk for an unspecified value that matches any input. Partial paths can be matched with an appended asterisk (e.g., `"/etc/apache2/*"`). Effective path arguments can additionally be evaluated using `stat` system calls. Possible actions for each combination are to allow the system call, to abort the program, or to return a predefined integer value. See Listing 4.1 for an excerpt from a policy file. This policy allows two specific files to be opened, and execution of the `time()`, `close()`, and `write()` system calls. `getuid()` returns 0 (root), and `access()` is restricted to `/etc/*` only. See Figure 4.5 for an overview of runtime data structures needed for the policy-based authorization.

The redirected system calls can be used to change and test the behavior of a program if certain system calls return special values (e.g., many programs behave differently if they are run as root, so returning a fake value for `getuid` is useful in

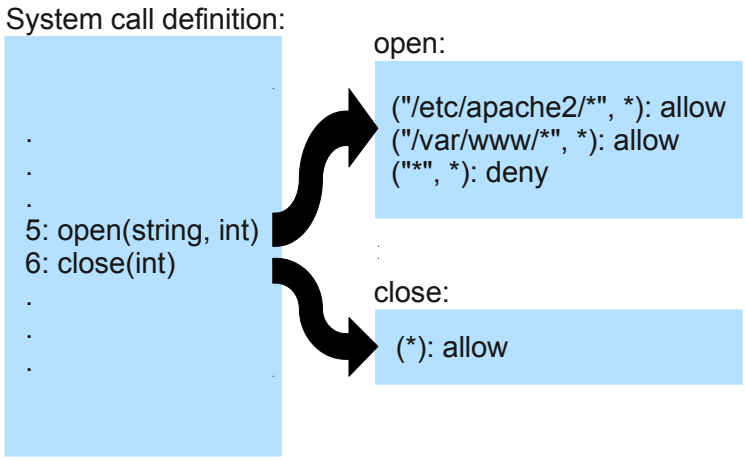

Figure 4.5: Runtime data structures for a given policy with examples for the `open()` and `close()` system calls.

some cases). The current policy is limited to returning a static fake value, although recall that the system call interposition framework can be used to call any user supplied function to handle a specific system call. This functionality enables, e.g., additional stateful security checks, the emulation of (obsolete or unsafe) system calls, virtualization or reimplementation of specific system calls in user space, or even kernel emulation in user-space.

Depending on the first line of the policy file, either white-listing or black-listing is used. Black-listing can be used, e.g., for testing or implementation of new features. For security policies we assume that a white-listing approach is taken. An unmatched combination of parameters for a specific system call either aborts the program if white-listing is used, or is allowed if black-listing is used. White-listing specifically allows system calls and implicitly denies all other system calls. Black-listing denies or redirects specific system calls and implicitly allows all unspecified system calls.

4.6 Summary

This chapter presents the design of the secure execution framework that implements two execution domains in user-space: the trusted sandbox domain and the untrusted application domain. The secure execution framework consists of three components: a secure loader, a SFI sandbox, and a policy-based system call interposition framework.

The secure loader extracts detailed runtime information (e.g., location of symbols, code regions, and import/export information of individual modules) about the application and ensures that all untrusted application code is executed in the

sandbox. Accesses to the loader API by the application trap into the privileged domain.

The sandbox checks all application code before it is executed and weaves security guards into the translated code. The information from the secure loader is used to restrict control flow transfers inside the application.

The policy-based system call interposition framework restricts the interaction between the application and the kernel. A per-application policy defines which system calls and parameters can be executed by the application.

5

System architecture and implementation

The system architecture of the secure execution platform is split into independent modules. The implementation follows a Unix-like design concept that uses different components that can be plugged together depending on the needs of the user.

Most components can be used as either a replacement for already existing products with additional security features (e.g., the secure loader) or as an alternative runtime environment for applications (e.g., the user-space sandbox, or the policy-based system call authorization layer).

The different components can be plugged together into a complete protection mechanism. This system is then called the secure execution platform. Individual components communicate through a well-defined API in the trusted domain.

The secure loader in Section 5.1 can be used as a secure replacement of the standard loader. The loader can then be used to execute all application code under the control of a user-space sandbox. Section 5.2 describes a generic table-based user-space binary translator that is used as a basis for the Software-based Fault Isolation (SFI) layer. The binary translator is hardened for security[1] and is extended by the security guards to enforce fine-grained user-space SFI; see Section 5.3 for implementation details.

The user-space sandbox can then be used in combination with the secure loader. The standard loader should not be used in a security related context because of the problems mentioned in Section 2.4.

Section 5.4 discusses the implementation of the dynamic control flow integrity checks. The control flow checks use a model of allowed targets for each source location. This model can be generated using different approaches defined in Section 5.4.1. If the secure loader is available then the loader forwards information about individual loaded shared objects into the model used by the control flow transfer checks.

[1] Security hardening is a process to reduce the vulnerability surface of a software system. Security hardening techniques include removing unneeded external dependencies, removing unneeded code, simplifying both the interface and the implementation, and adding validation for untrusted input.

Policy-based system call authorization relies on an SFI system that redirects all system calls to an interposition layer. This interposition layer is extended to include a policy parser that enforces per-application policies. Section 5.5 presents implementation details of the system-call interposition layer.

The code base of the secure execution platform is small; the secure loader consists of around 5,400 lines of code (including 2,100 lines of comments) and the sandbox platform consists of around 15,200 lines of code (including 3,200 lines of comments and 4,900 lines for the full IA32 translation tables).

5.1 Secure loader

The security framework integrates the information from the loader into the security guards. The secure loader initializes the sandbox before any application or library code is loaded or executed. All application and library code is then executed in the sandbox. The source code of the prototype implementation is available as open-source.

The secure loader uses Executable and Linkable Format (ELF) information and symbol table information [SCO96] and implements all needed functionality to load most programs (e.g., OpenOffice, and the SPEC CPU benchmarks).

The SFI platform is tightly coupled with the secure loader. The loader first maps libdetox into the address space and initializes the SFI platform. This special treatment ensures that the SFI platform is initialized and that the application has no access to or knowledge of the sandbox domain. The next steps are the relocation of the application and all needed shared objects. The loader controls all data that is passed to the application and runs all user code under the control of the SFI platform.

5.1.1 Application and library loading

The secure loader implements the most common subset of features from the standard loader. Some features (e.g., overwriting library search paths) are removed out of security concerns. Unimplemented features result in an error message and graceful termination of the program. The current implementation prototype covers the core functionality needed to execute many programs of Ubuntu 11.04[2]. Further options (e.g., obscure relocation patterns, additional callbacks from the application into the loader, and access to internal loader data[3]) can be added if needed.

The standard loader has no protection for internal data structures and leaks

[2]We executed a selected set of applications to demonstrate the practicality of this approach but did not test the full set of applications run in our department's infrastructure.

[3]E.g., GDB uses undocumented direct access to the internal data from the loader; this feature can be implemented as a proxy that projects information out of the secure loader if needed.

pointers to the internal data structures to the application. The API of the secure loader that is accessible from the application (e.g., `dlopen`, `dladdr`, and `dlsym`) ensures that no protected internal data is leaked to the application. The sandbox write-protects all internal data whenever (translated) application code is executed.

The secure loader must handle the startup of new applications. First of all the loader is completely independent from any libraries (even the standard libc) and is just mapped into memory. This loader then examines the ELF headers of the application and maps the runtime sections of the application to a fixed address in memory. Then the list of needed libraries is examined and entries are added to a "to-process-list". The loader dequeues one entry at a time and loads and initializes this library at random addresses. If the library depends on other libraries then they are added at the end of the "to-process-list". This algorithm conforms to a breadth-first traversal of the dependence graph of the application starting with the application as the root node.

References to needed libraries only contain the name of the library but not the path. When the loader locates a new library several paths are examined: first a per-Dynamic Shared Object (DSO) variable that specifies one or more search paths per DSO, then the standard search paths defined in `/etc/ld.so.conf`. The standard libc loader also supports additional search directories using the `LD_LIBRARY_PATH` environment variable and the local cache file `/etc/ld.so.cache`. Out of security reasons the secure loader does not support runtime-configurable paths; the search path for libraries is restricted to the compile-time configuration.

5.1.2 Symbol resolving

The loader resolves symbols using the symbol tables in the different shared objects. Every shared object contains the `.dynsym` table with all exported symbols. If the loader needs to resolve an imported symbol then the loader checks different lookup scopes. The loader defines three different lookup scopes that are checked one after the other:

1. *Loader scope:* this scope contains the symbols that are exported by the secure loader. The loader scope is checked first and symbols in this scope cannot be overwritten.

2. *Local scope:* the local scope of a DSO contains its own symbols and the symbols of all libraries that the DSO depends on. This scope is a (smaller) subset of the global scope.

3. *Global scope:* shared objects that are in the initial set of objects loaded during the startup of the application (e.g., all objects in the dependence graph) or shared objects that are loaded at runtime with the `RTLD_GLOBAL` flag set are in the global scope.

A special feature is symbol versioning where symbols can be defined multiple times with different versions. The correct symbol is then selected based on a matching version.

The secure loader supports the GNU IFUNC relocation format (`STT_GNU_IFUNC`) where a piece of code is executed to determine the correct location of the symbol. This feature is, e.g., used in libc to select between multiple implementations of a function. The test function checks if a specific CPU feature is available and returns the most optimized version for the current environment. The loader then uses this function pointer and forwards it to the requesting DSO where the function pointer can be embedded in the Global Offset Table (GOT).

5.1.3 Memory protection

One of the advantages of a secure loader is that all loader-related data structures can be write-protected. The secure loader manages two kinds of data structures: internal data structures and application data structures.

Internal data structures contain information about the different relations between shared objects, scope information, and other details about the loaded objects. This information is updated by the secure loader whenever new shared objects (e.g., additional shared libraries) are loaded and initialized. The secure loader maps these data structures read-only whenever application code is executed.

Shared objects contain data structures that are only changed by the loader and are only read by the application. If we take the standard libc 2.1.3 as an example we see in Table 5.1 that there are 34 ELF sections that are mapped to memory. 11 sections are mapped writable (`.tdata`, `.tbss`, `.fini_array`, `.ctors`, `__libc_subfreeres`, `__libc_atexit`, `__libc_thread_sub`, `.data.rel.ro`, `.dynamic`, `.got`, `.got.plt`, `.data`, `.bss`) and 1 section (`.dtors`) is marked read-only but on the same memory page as `.ctors` and is therefore writeable as well. Most of these sections are used only during the initialization of the shared object. The sections `.data.rel.ro`, `.dynamic`, `.got`, and `.got.plt` are critical for the loader and can be used in attacks against a sandbox. The standard loader maps `.data.rel.ro` as read-only after the initialization but the other sections remain writeable. Out of the writeable set of sections only `.data` and `.bss` are used by the libc code.

The secure loader write-protects all sections except `.data` and `.bss` dynamically to protect the application from modification attacks in these sections. If the secure loader needs to update write-protected structures (e.g., a GOT entry) then the write-permission is set temporarily during the update in the sandbox domain.

5.1.4 Loader optimizations

The secure loader currently implements two optimizations: lazy binding and Procedure Linkage Table (PLT) inlining.

Nr	Name	Type	Size	Flags
0		NULL	0	
1	.note.gnu.build-i	NOTE	24	R
2	.note.ABI-tag	NOTE	0x20	R
3	.gnu.hash	GNU_HASH	0x3c38	R
4	.dynsym	DYNSYM	9200	R
5	.dynstr	STRTAB	005acd	R
6	.gnu.version	VERSYM	0x1240	R
7	.gnu.version_d	VERDEF	0x3d8	R
8	.gnu.version_r	VERNEED	0x40	R
9	.rel.dyn	REL	0x2a20	R
10	.rel.plt	REL	0x40	R
11	.plt	PROGBITS	0x90	RX
12	.text	PROGBITS	0x1088d4	RX
13	__libc_freeres_fn	PROGBITS	0xfc8	RX
14	__libc_thread_fre	PROGBITS	0x182	RX
15	.rodata	PROGBITS	0x1b808	R
16	.interp	PROGBITS	0x13be68	R
17	.eh_frame_hdr	PROGBITS	0x333c	R
18	.eh_frame	PROGBITS	0x132b4	R
19	.gcc_except_table	PROGBITS	0x5c1	R
20	.hash	HASH	0x3484	R
21	.tdata	PROGBITS	0x8	RWT
22	.tbss	NOBITS	0x38	RWT
23	.fini_array	FINI_ARRAY	0x4	RW
24	.ctors	PROGBITS	0x14	RW
25	.dtors	PROGBITS	0x8	R
26	__libc_subfreeres	PROGBITS	0x70	RW
27	__libc_atexit	PROGBITS	0x4	RW
28	__libc_thread_sub	PROGBITS	0xc	RW
29	.data.rel.ro	PROGBITS	0x1afc	RW
30	.dynamic	DYNAMIC	0xf0	RW
31	.got	PROGBITS	0x174	RW
32	.got.plt	PROGBITS	0x2c	RW
33	.data	PROGBITS	0x97c	RW
34	.bss	NOBITS	0x3068	RW
	...			
68	not allocated			

Table 5.1: These sections of the standard libc are mapped at runtime using the given flags (X - execute, W - Writeable, R - Readable, T - Thread local storage). Command used to get this information: `readelf -S /lib32/libc-2.13.so`.

Lazy binding reduces the number of relocations that must be calculated when a library is loaded. With this optimization active symbols in the data region are relocated ad hoc; symbols in the PLT region are only resolved and relocated when the function is first executed. This optimization is also implemented in the standard loader.

The implementation of PLT inlining follows the design in Section 4.1.3 and uses the close relationship between the secure loader and the sandbox. The sandbox intercepts all call instructions and checks for each instruction if the call is a PLT call. The secure loader then resolves the static target address of the PLT target. The original call and indirect jump of the PLT call are then replaced by a translated call instruction to the resolved target. This translation removes an indirect jump including an indirect control flow check for every PLT call that is executed.

During the loading process weak symbols of prior DSOs can be overwritten by symbols in the current DSO. If the weak symbol points to a function and this function was inlined (through a PLT slot) then the sandbox flushes its code cache and retranslates all code.

5.1.5 Handling of the sandbox

The secure loader handles the sandbox in a special way. The loader resolves the additional sandbox code before any shared library or application code is loaded and initialized. The symbols of the libdetox framework are also resolved in a protected namespace that is only accessible by the secure loader and the sandbox.

Any application or library code that is then executed during the initialization phase is executed under the control of the sandbox, enabling security right from the start.

5.1.6 Integration of the secure loader into the sandbox

The SFI sandbox is based on the libdetox [PG11] open-source project. Changes to the original implementation are an API for the secure loader, an alternate sandbox stack for functions in the sandbox domain, and a new shadow stack to store information about the application stack in the sandbox domain.

The sandbox uses the secure loader to lookup information on the different sections and hence decides if code is in an executable section of a shared object or in some other region. The same information is used to implement PLT inlining as described in Section 5.1.4.

Our sandbox uses specific entry and exit trampolines to simplify the transition between the application domain and the sandbox domain. The entry trampoline handles the transition from translated application code to privileged sandbox code. The application stack remains unchanged, registers are spilled to a thread-local storage area in the sandbox domain and the stack is swapped to a sandbox stack. Code

running in the sandbox domain uses the sandbox stack to store local information. The exit trampoline returns from the sandbox domain to the application domain. The trampoline restores registers, switches back to the application stack, and continues the execution of the translated code.

Events that trigger a switch from the application domain to the sandbox domain are:

Lookup misses in the mapping table: if an indirect control flow transfer cannot be resolved with the inlined assembler code (e.g., a quick lookup in the first entry of the mapping hash table) then the control flow transfer code escalates to the sandbox domain and requests a slow-path lookup.

Untranslated code: if the translated application code branches to untranslated code an exception is triggered and the sandbox either translates the untranslated code and continues execution or faults.

Signals and exceptions: the sandbox installs special handlers to catch all signals and exceptions. These handlers check the signal or exception, resolve the original instruction pointer[4], check if the signal or exception is legit, and pass the information to the application.

System calls: system calls trigger a switch to the sandbox domain. A handler copies the arguments of the system call into the sandbox domain and checks the combination of system call and parameters using a per-application policy. The system call is evaluated in the sandbox domain to protect from *time of check to time of use attacks* by concurrent threads.

5.1.7 Implementation alternatives

We discuss two implementation alternatives that offer a similar level of security to the combination of a secure loader with a sandbox. The first alternative uses static recompilation. All libraries are compiled to a statically linked binary, guards are added during the recompilation, and the loader is no longer needed. This approach has several drawbacks: (i) there is no second protection domain; exploits can get control of the user-space and then execute arbitrary system calls, (ii) static recompilation is limited to statically known targets and code locations (e.g., handling of dynamic jump-tables for switch statements is not possible), (iii) a secure static runtime environment must restrict the Instruction Set Architecture (ISA) and the dynamic control flow transfer instructions to limit the dynamic options of the IA32 ISA.

A second alternative implements a sandbox without changing the loader. The sandbox is hidden from the application using loader tricks that alter the data struc-

[4]The kernel passes an instruction pointer to the code cache that must be resolved to a pointer in the application domain before the signal or exception is passed to the application.

tures of the loader, or the sandbox is added as a binary blob and injected into the process image by an external process. This implementation approach has the disadvantage that it is hard to hide the sandbox from the loader/application and to remove all traces from the sandbox in the loader data structures. A second disadvantage is that loader code is translated as well, including when new symbols are resolved. This disadvantage results in an unsolved loader black box problem.

5.2 A generic dynamic binary translator

A generic dynamic binary translator processes basic blocks of input instructions, places the translated instructions into a code cache, and adds entries to a mapping table. The mapping table is used to map between addresses in the original program and the code cache. Only translated code is executed. The translator is started and intercepts user code whenever the user program branches to untranslated code. The user program is resumed as soon as the target basic block is translated. See Figure 5.1 for an overview of such a basic translator.

The user stack contains only untranslated instruction pointers. This setup is needed to support exception management, debugging, and self-modifying code. Each one of these techniques accesses the program stack to match return addresses with known values. These comparisons would not work if the stack was changed, because the return addresses on the stack would point into the code cache, instead of into the user program. Every return instruction is replaced with an indirect control transfer to return to translated code. This design decision enables a user-space virtualization

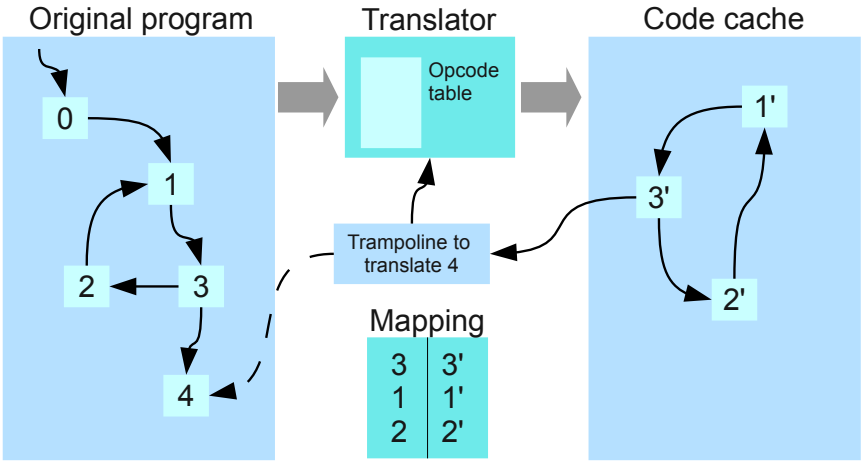

Figure 5.1: Runtime layout of the binary translator. Basic blocks of the original program are translated and placed in the code cache using the opcode tables. The mapping table maps addresses in the program code to translated addresses. Trampolines are used to dynamically invoke the translator for untranslated basic blocks.

principle. User-space virtualization ensures that the program cannot easily discover that it is executed in a virtualized environment.

5.2.1 Translation tables

The translation engine is a simple table-based iterator. The translator is invoked with a pointer to an untranslated basic block, allocates space for the translated instructions, adds an entry into the mapping table, and translates the basic block one instruction at a time using the translation tables.

Instructions of the IA32 architecture have a variable length from 1 to 17 bytes. These instructions are decoded using multidimensional translation tables. The translation tables contain extensive information about all possible encodings for all IA32 instructions and parameters. The decoding of a new instruction starts with the first byte of the instruction, which is used as an index into the first translation table. The indexed row contains information about the instruction or a forwarding pointer to the next table with the next instruction byte if the instruction is not finished. The decoding process continues until the instruction is decoded. The final indexed row contains the information about the instruction. Table 5.2 shows an example.

...	...	...
flags	*next_tbl (or NULL)	&action_func
...	...	...

Table 5.2: Layout of an opcode table, the opcode byte is used as table offset. The flags contain information about parameters, immediate bytes, and registers.

After the decoding of the instruction the instruction and parameters are passed to the corresponding action function. The action function handles the transcription of the instruction to the code cache. The function can alter, copy, replace, or remove the instruction.

The translation process stops at recognizable basic block boundaries like conditional branches, indirect branches, or return instructions. Backward targets are not recognizable by a single-pass translator and multiple concatenated basic blocks are translated as one long extended basic block. In this case the second part of the extended basic block is translated a second time as a shorter basic block. The additional translation overhead for these split basic blocks is negligible and smaller than the overhead of a multi-pass translator.

At the end of a basic block the translator checks if the outgoing edges are already translated, and adds jumps to translated targets. For untranslated targets a trampoline is constructed that starts the translation of the target. Trampolines are short sequences of code that transfer control flow to a different location.

5.2.2 Predefined actions

A translator needs various action functions even to support the identity transformation. The main challenge is to redirect control flow into the code cache but maintain the illusion that the program is executed from the original location.

Several action functions remove individual instructions or copy an instruction into the code cache. Other actions are needed to keep the execution flow inside the code cache:

Jump: Translates a direct jump in the original program. This action uses the mapping table to look up the translated target and encodes a branch to the translated target into the code cache.

Call: This action emits code to push the original location onto the stack and emits a branch to the translated call target.

Jump conditional: Translates a conditional jump. Code that branches to translated jump targets is emitted into the code cache. A trampoline is constructed for untranslated targets.

Jump indirect: This action encodes a runtime lookup that translates the original target into a target in the code cache and emits a branch to the translated target.

Call indirect: Emits code that pushes the original location onto the stack followed by a runtime lookup and dispatch of the indirect target into a location in the code cache.

Function return: This action emits a translation of the return address on the stack into a target in the code cache.

5.2.3 Code cache

It is likely that a code region is executed multiple times during the runtime of a program. It makes sense to keep translated code in a code cache for later reuse. Code is translated only once, minimizing the overall translation overhead.

As reported in [BA05] and [BKGB06], code sharing between threads is low for desktop applications and moderate for server applications. We therefore propose a *per thread cache*-strategy to increase code locality and to potentially extract better thread local traces.

Important advantages of thread local caches are that (i) accesses to the code cache need not be synchronized between threads, and the high overhead of locking is avoided, and (ii) it is possible to emit hard-coded pointers to thread local data structures which remove the need to call lookup functions. This feature is used in

the translator itself for *syscall* and *signal* handling and to inline and optimize all thread local accesses from the instrumented program.

A trampoline is a piece of code that triggers the translation of a basic block. The binary translator places these trampolines in a separate code region where they can be reused after the target's translation. This keeps the code cache clean and increases code locality.

The combination of the basic translator and the code cache leads to *greedy trace extraction* and *code linearization*. Traces are formed as a side effect of the first execution and placed in the code cache.

5.2.4 Mapping table

The mapping table is a hash map that maps addresses in the original program to addresses in the code cache. The hash map contains all translated basic blocks.

The original program does not know that it is translated. All targets for indirect control transfers (e.g., function returns, indirect jumps, and indirect calls) must be handled through an online lookup and dispatch. These dispatch mechanisms use the mapping table to get the corresponding translated target in the code cache. The translator uses the mapping table to translate function calls, conditional branches, and direct branches.

The hash function that is used in these lookup functions returns a relative offset into the mapping table. This hash function is kept as simple as possible because (i) perfect or near perfect hashing is not needed, (ii) the overhead for hashing should be low, and (iii) it should be implemented in a few machine code instructions. In the case of a collision the hash function loops through the mapping table until the entry or NULL is found. Tests with SPEC CPU2006 benchmarks and various programs showed that the simple hash function `addr<<3 & (HASHTABLE_SIZE-1)` results in a low number of collisions, where the hashtable size is 2^x bytes. The best size of the hashtable is highly dependent on the amount of code that will be translated. A reasonable value for x is 23, which results in an 8MB hashtable that holds up to 2^{20} individual address mappings.

5.2.5 Basic translation of indirect control flow transfers

Relative control flow transfers and indirect control flow transfers are translated to a sequence of instructions so that the control transfer targets translated code in the code cache.

Relative control flow transfers are replaced with a static sequence of instructions that redirects control flow to the translated counterparts as depicted in Table 5.3. The target can be checked once during the translation of the instruction and no dynamic runtime security check is needed for these static targets.

`jmp target`	`call target`
-	`push eip`
`jmp transl_target`	`jmp transl_target`

Table 5.3: Translation of relative jump and call instructions.

`jmp (r/m`[a]`)`	`call (r/m)`	`ret`
-	`push eip`	-
`push (r/m)`	`push (r/m)`	eip on stack
`jmp indjmp_trm`	`jmp indcall_trm`	`jmp indcall_trm`

Table 5.4: Translation of indirect jump, call, and return instructions.

[a] r/m specifies a register or memory address

Indirect control flow transfers are translated to a lookup in the mapping table so that the control flow can be redirected to the translated target. Table 5.4 shows how indirect control flow transfers are translated. The emitted code prepares the stack, pushes the target pointer if needed, and dispatches the control flow to a thread local indirect call or indirect jump trampoline. The trampolines handle the additional security checks and verify the target.

5.2.6 Optimizations

The basic, simple binary translator uses hardcoded references in the code cache for direct control transfers. The translator must emit an online lookup of the target if the control transfer is indirect (e.g. indirect jump, indirect call, function return, or newly translated block), and the target is not known at translation time. This routine resolves the indirect target, issues a mapping table lookup, translates new targets, and redirects the control flow to the specified target. This naive translation results in high runtime overhead because the mapping table lookup is executed often.

The execution of translated indirect control transfers is the biggest contributor to the overall overhead of binary translation. Different optimization strategies, e.g., function inlining, and indirect control transfer optimizations, are used to reduce the overhead of indirect control transfers.

Function inlining

Function inlining is an effective way of reducing the overhead of indirect control transfers. If a function is inlined then the return instruction can be translated as an addition of 4 to the `%esp` register instead of an indirect jump. Function inlining saves at least 12 instructions compared to a translated return instruction.

```
  pushl eip      # current eip for call trampoline
  pushl target   # target of call instruction
  pushfl
  cmpl $cached_target, 4(%esp)
  jne no_hit
  popfl
  leal 4(%esp), %esp
  jmp $translated_target
no_hit:          # recover and fix trampoline
  movl fixup_locations, secure_stack
  call indcall_fixup_trampoline
```

Listing 5.1: Indirect call instruction using a prediction

Indirect CALL optimization

Indirect calls are used for virtual methods in object-oriented languages as well as for function pointers in C-like languages. The target is not known at translation time and can vary upon successive calls. The translator pushes the callee's address onto the stack and adds a runtime lookup to handle these transfers. The basic binary translator implements two different optimizations:

- The *indirect call fast* optimization compares the first entry in the mapping table with the target. If it matches then control is transferred to the destination. Otherwise the binary translator falls back to the slow path that uses a loop to find the correct destination. This optimization is used if the *indirect call prediction* has a low hit-rate. If the lookup in the mapping table is a direct hit then an indirect call instruction is executed in 13 instructions (this happens in the majority of the cases as the hit rate in the mapping table is above 99.9% for the SPEC benchmarks).

- The *indirect call prediction* caches the last lookup target and destination (see Listing 5.1). If the target is the same then the control can be transferred without a mapping table lookup; otherwise the cache must be updated. This optimization relies on the fact that the target of an indirect call instruction does not change often. The binary translator uses this optimization for all indirect calls at the beginning. If there are more than `MAX_NR_MISPREDICTIONS` (empirically set to 20) mispredictions then the prediction is replaced by a fast indirect call. If the prediction is correct then an indirect jump is executed to the cached target.

Indirect JUMP optimization

Indirect jumps are used in, e.g., dynamically loaded libraries, jump tables, switch statements, or dispatch functions. As with the indirect call instruction, the target can vary and is not known at translation time. The binary translator implements two different optimizations to handle these indirect control transfers:

- The *indirect jump fast* optimization is similar to *indirect call fast*, but without pushing the EIP onto the stack.
- The *indirect jump prediction* follows the same principle as the *indirect call prediction*, without pushing the EIP.

The action handling the indirect jumps selects the most promising optimization to apply and emits corresponding machine code to the code cache.

5.2.7 Signal and syscall handling

User-space binary translation systems need special treatment for signals and system calls. A task or thread can schedule individual signal handler functions and execute system calls. The kernel transfers control from kernel-space back to user-space after a system call or after the delivery of a signal. A user-space binary translator cannot control these control transfers, but must rewrite any calls that install signal handlers or execute system calls so that the binary translator regains control after the context switch.

Our approach catches signal handlers and system calls and wraps them into trampolines that return the control flow to translated code. We handle signals installed by *signal* and *sigaction* and all system calls, covering both interrupts and the *sysenter* instruction. The *sysenter*-handling is specific to the syscall handling of the Linux kernel 2.6 [Gar06].

5.3 Software-based fault isolation layer

SFI extends the generic binary translator described in Section 5.2. An important feature is that the binary translator covers the complete IA32 instruction set. To offer an attractive alternative to full system translation the overhead for the binary translation must be low, both for the translation of new instructions and for the execution of translated code.

libdetox supports (i) the complete IA32 instruction set, (ii) the binary translator is modular and expandable by new handler functions that control the low-level translation of control instructions, and (iii) the trusted computing base is small; libdetox consists of only a couple of thousand lines of code. The binary translator

framework translates all control flow instructions and ensures that the execution stays inside the thread-local code cache.

The combination of a binary translator with the additional security guards implements the code sandbox defined in Section 4.2. Illegal instructions and instructions that redirect control flow to malicious code lead to a immediate abort of the program.

5.3.1 Thread and process handling

All system calls that create threads or new processes are handled in a special way. If the arguments are allowed according to the policy then the system call is instrumented such that the new thread or process is started in a new libdetox instance.

System calls are redirected on the basis of either `int $0x80` or `sysenter` instructions. These instructions are replaced by a check that redirects the system call to the authorization framework. If the system call executes `clone` or `fork` to start new processes or threads then the new thread or process is started under the control of a new libdetox instance. The system call interposition framework rewrites the arguments to ensure that libdetox keeps control of the application.

If an `execve`-like system call is used to replace the current runtime image of the application with an alternate program then the system call interposition framework ensures that the new program is started in a new (and independent) libdetox instance. If the secure loader is available then the new program is executed using the secure loader, otherwise the standard loader is instructed to use the `LD_PRELOAD` directive to start the new program under the control of libdetox (with limited security, see Section 2.4.3).

If the parameters do not match the policies, e.g., if the program is not allowed to start new threads or processes or if the program is not allowed to execute any other program, then the application is terminated.

5.3.2 Additional guards

libdetox reads all symbol and section information when the program or shared libraries are loaded. This information is imported into data structures of libdetox that are used at runtime to implement the additional security guards.

The guards from Section 4.2.1 are executed either when new code is translated or whenever the translated instruction is executed in the application. Non-executable data, executable bit removal, and signal handling are implemented using static implementations during the translation process. The section guard, call guard, and return address verification need both a check during the translation and an additional dynamic check. The check during the translation process is used for fixed or precomputed targets and the dynamic check is patched into the translated code for all dynamic control flow transfers.

```
/* 2 instructions needed to setup secure stack */
  movl %esp, (tld->stack-1)   /* (1) save old SP */
  movl tld->stack-1, %esp     /* (2) change SP */
/* from here on we use the secure stack */
  # push arguments            /* setup translation */
  call translate              /* call function */
  leal 8(%esp), %esp          /* cleanup stack */
  movl %eax, tld->ind_target  /* save target */
  popl %esp                   /* (3) restore SP */
/* back to regular application stack */
  jmpl *(tld->ind_target)     /* return */
```

Listing 5.2: Transfer gate to secure stack for internal functions

Secure context transfers are implemented through changes in the binary translator. The optimizations for indirect control flow transfers are modified so that no pointers to the code cache are left on the stack of the user-program. For example, an indirect jump instruction is translated into code that (i) executes a lookup in the mapping table, (ii) stores the translated target address in a local data structure, and (iii) uses an indirect jump through that data structure to redirect the control flow to the translated target. Using such trampolines guarantees that pointers to the code cache are never left on the application stack and there is no need to overwrite return addresses of the original application which would leak information about the sandbox.

5.3.3 Secure Stack

The implementation of libdetox uses a dedicated stack for all internal functions. No internal data is ever saved to the application stack. The switch to the secure stack happens through a special gate that is used whenever internal functions (e.g., translating new code, updates to the control transfer lookup model, or system call handling) are executed. This call gate stores the application state on the internal stack and switches from the application stack to the internal stack. The second call gate that transfers control back to the application uses the state on the internal stack to switch back to the application stack.

Trampolines that transfer code from one translated block to another do not store any internal data on the stack, thereby reducing the number of context switches. Listing 5.2 shows that switching to a secure stack costs three additional instructions (two instructions in the preamble, one instruction in the epilogue); this overhead is tolerable for functions that, e.g., translate new application code.

```
  pushl %ebx
  /* load RIP from shadow stack */
  movl tld->top_of_shadowstack, %ebx
  movl -4(%ebx), %ebx
  /* compare shadow stack RIP with actual RIP */
  cmpl 4(%esp), %ebx
  jne authorization_error
  /* load translated target from shadow stack */
  movl tld->top_of_shadowstack, %ebx
  movl -8(%ebx), %ebx
  /* store target */
  movl %ebx, (tld->ind_target)
  /* adjust stack */
  subl $0x10, tld->top_of_shadowstack
  popl %ebx
  leal 4(%esp), %esp
  jmpl *(tld->ind_target)
:authorizarion_error
  /* check for stack-based attack */
```

Listing 5.3: Translated return instruction

5.3.4 Shadow stack

The shadow stack is a separate secure stack that is hidden from the application. Translated return instructions use this stack to verify the correct return targets. The current implementation of the shadow stack uses pairs of addresses with original and translated addresses.

Call instructions push the current instruction pointer to the original stack before they transfer control to the new location. The translated call instructions push the current instruction pointer and the translated instruction pointer to the shadow stack as well.

Return instructions ensure that the return instruction pointer on the shadow stack is the same as the return instruction pointer on the application stack. If both pointers match then the top shadow stack frame is removed and control is transferred to the translated target using the pointer on the shadow stack. Listing 5.3 shows a translated return instruction.

5.3.5 Limitations of the current implementation

The current implementation does not support self-modifying code or just in time (JIT) compilers. Code that is generated dynamically cannot be trusted, even if the JIT compiler generates all needed symbol information. A simple approach handles

the code region where all dynamically compiled code is emitted as one large symbol. An attack could exploit the code generator and generate code that breaks the control transfer rules in the JIT code region.

A possible extension raises the JIT compiler into the trusted computing base and implements a (one-way) interface between the JIT compiler and libdetox to exchange information about locations and properties of dynamically compiled methods.

A second limitation of the current approach is that jump-oriented programming attacks [BJFL11] and data-only attacks (application data is over-written using a malicious write to a memory page) are still possible if the sandbox is used in isolation without control flow integrity as discussed in Section 4.3. Jump-oriented attacks and data-only attacks can redirect the control flow to alternate locations in the code but the attacks can never introduce new code or break out of the sandbox. Only translated code is executable and all outgoing edges at the end of a basic block in the code cache are either patched to other translated basic blocks or trigger a fallback into the sandbox to translate previously untranslated code.

5.4 Dynamic control flow integrity

The control flow transfer guards are implemented in the core translator of the libdetox library. Every static control transfer must adhere to the model defined in Section 4.4. The control flow transfer guards use hooks in the individual translator functions that handle the translation of the different control flow transfer instructions. For static control flow transfers the verification of the rules happens during the translation process. For indirect or dynamic control flow transfers the action function that translates the indirect control flow transfer emits additional code that invokes a dynamic check. The dynamic checks use the same model defined in Section 4.4 to verify compliance with the control transfer rules. The dynamic check pushes source location in the original program, dynamic target location, type of control flow transfer (e.g., indirect jump, or indirect call), and the pointer to the translator data onto the stack. This data is then used in the dynamic check to determine if the transfer for the specified type is allowed.

5.4.1 Information gathering

The control transfer lookup model could be constructed through the `LD_AUDIT` hooks from the standard libc dynamic loader, through the `dl_iterate_phdr` function of the dynamic loader that allows a traversal of the loaded dynamic shared objects, or using information from the secure loader. Whenever a new library is loaded or closed the information in the per-application control transfer lookup model must be updated on-demand.

The LD_AUDIT interface

`LD_AUDIT` offers an audit interface for the dynamic linker on Linux platforms. A library can specify individual callback functions that are executed whenever the application loads a new dynamically shared object (e.g., a library) or resolves a symbol or function. Unfortunately, this approach leads to high overhead because a dynamic function is executed whenever a symbol is resolved or an inter-module function call is dispatched. Additionally, this approach does not work for all shared libraries and DSOs. This solution is also prone to the loader black box problem from Section 2.4.3. Therefore we reject this approach.

The dl_iterate_phdr interface

The `dl_iterate_phdr` interface allows a direct on-demand access to the information of the dynamic linker. This interface implements an iterator over all loaded DSOs. A user-definable callback function processes the information about each DSO. This approach is stable and introduces low overhead. The libraries are only resolved once when the virtualization platform starts. Unlike the `LD_AUDIT` approach, there is no additional overhead, for inter-module function calls.

The dynamic linker resolves all shared libraries at runtime. When the dynamic loader finishes it passes control to the libdetox startup routines. Libdetox uses the `dl_iterate_phdr` interface to parse all loaded DSOs. The symbol information of each library is loaded into the control transfer lookup model and additional information (e.g., location and size of PLT and GOT) about each DSO is stored in a general record table. Unfortunately, this solution is also prone to the loader black box problem and hence is also rejected.

The secure loader approach

This approach changes the secure loader directly to export the needed information in the correct data type. The secure loader already starts the sandbox whenever application code is executed. This means that the loader has control over the sandbox and knows where the data structures of the sandbox are located[5].

The loader uses this information to inject an up-to-date information model about all loaded shared objects and all available control flow targets into the sandbox. This model is updated whenever loader code is executed to resolve or update any symbols or when loader code is executed to dynamically load additional shared objects.

[5]The loader knows where the data structures are located because the loader relocated the sandbox and resolved the internal symbols when the library was loaded.

5.4.2 Limitations of the current implementation

The two biggest problems of the current implementation are (i) stripped binaries and libraries and (ii) imprecise symbol definitions. Both problems result from the (static) linking process of the applications.

The static linker can strip (remove) the symbol tables from libraries and binaries. Stripped objects result in a less exact (and less fine-grained) execution model. A solution to this limitation would be to keep the symbol table information in all libraries and applications. The only penalty for this solution is increased on-disk space usage for the (usually small) additional symbol table.

The second problem is due to imprecise symbol definitions in objects. Programmers can pass pointers to functions in other objects (e.g., registering callbacks in libraries). If these function pointers do not point to global symbols then the transfer checks will fail, because the function is called from outside of the module. These functions should be exported as global symbols. Our prototype keeps a whitelist of functions that can be used for such callbacks.

A last limitation is that the `setjmp`, `longjmp` feature is no longer supported if the `longjmp` transfers control flow between different functions or modules. This feature is not used often and if it is crucial for a program then an additional whitelist for target locations of long-jumps could be implemented.

5.5 Policy-based system call authorization

All system calls through interrupts and the `sysenter` instruction are rewritten by the binary translator. The system call interposition framework is implemented on top of the binary translator to wrap all system calls into individual evaluation functions. The system call interposition framework then checks the system call and its arguments against the loaded policy. libdetox loads the policy and parses it into an array of parameter lists. Per system call, a parameter list is generated with combinations of valid parameters. If the user program wants to execute a system call then the list is checked. If a parameter-set matches then the system call is either executed or a fake value is returned. The process is terminated if no parameter-set matches. Process termination ensures that no compromised data escapes the sandbox. Sandboxing ensures that no code-based attacks are possible and policy-based system call authorization ensures that the application is confined to the per-application system call policy. The combination of these two principles makes user-space isolation and encapsulation possible and secure.

The system call interposition framework contains an array of function pointers with handler function for each system call. These handler functions either point to a special function for a specific system call (e.g., the handler for `mmap` or `clone`) or to the general policy handler. See Section 4.5.1 for more details on intercepted system calls with specific handler functions.

```
struct syscall_policy_entry {
  /** argument definitions for all syscall parameters */
  struct syscall_argument *args[6];
  /** what should we do if this policy rule matches? */
  enum syscall_auth_response action;
  ulong_t fake_value;
  /** next argument combination in list */
  struct syscall_policy_entry *next;
};
```

Listing 5.4: Policy entry for a single system call

```
enum syscall_auth_response {
  /** syscall authorization granted, execute syscall */
  SYSCALL_AUTH_GRANTED,
  /** syscall execution denied, return fake value */
  SYSCALL_AUTH_FAKE,
  /** syscall rejected, app is terminated*/
  SYSCALL_AUTH_DENIED
} __attribute__((packed));
```

Listing 5.5: Possible responses to a parameter set

The policy handler keeps a table with policy entries on a per-system call basis. Individual policy entries for a specific system call are enqueued in a linked list. The linked list is traversed until a policy entry matches or until the end of the list is reached. If the end of the list is reached then the default action (e.g., terminate the application with a security exception) is executed.

The implementation uses a table that contains policy entries for each system call. The policy handler checks all entries for the specified system call using the information in Listing 5.4. One such entry corresponds to a single line in the policy file. System calls can have up to 6 arguments that must match and the policy line can specify a single response action if the current entry matches. If the entry does not match then the next pointer can be used to check the next entry.

A policy entry contains information about individual parameters (see Listing 5.5 and Listing 5.6). An individual argument can be either an integer, a pointer to some application-local data structure, a 0-terminated string, or a path which is essentially a string but interpreted by the kernel and resolved to a specific file.

```
enum syscall_argument_type {
  /** an integer value */
  SYSCALL_ARG_INT,
  /** pointer to unspecified data */
  SYSCALL_ARG_POINTER,
  /** a string, \0 terminated */
  SYSCALL_ARG_STRING,
  /** a path (used by file-related system calls) */
  SYSCALL_ARG_PATH,
  /** ignored */
  SYSCALL_ARG_IGNORE
} __attribute__((packed));

struct syscall_argument {
  /** type, e.g., int, pointer, or string data */
  enum syscall_argument_type type;
  union {
    ulong_t int_value;
    void *pointer_value;
    char *string_value;
  } data;
};
```

Listing 5.6: Struct for a single parameter in a policy entry

5.6 Discussion

The basic sandbox protects from all code injection-based attacks (on the stack and on the heap). Regular code sections of the application are mapped read-only and only translated application code in the code cache is executable. Other memory pages of the application are never mapped with executable permissions.

The shadow stack protects the return instruction pointer using a privileged shadow stack in the sandbox domain. This guard protects from all stack-oriented attacks (return to libc attacks and return-oriented programming [Sha07]).

We can use a system call policy to ensure that the application code cannot break out of the sandbox and to protect from jump-oriented attacks and data attacks at a later, more coarse-grained system-call level. An advantage of our extended trusted computing base is that we do not need to consider the system calls needed by the loader in our policy. The policy can be reduced to the functionality actually needed by the application and is not polluted by system calls that are needed for loader functionality.

6

Evaluation

This chapter evaluates the model for the secure execution of untrusted code. The design and feasibility of the model is evaluated in Section 6.1. Different exploit classes are analyzed and we show how an attack can be circumvented by the model. The different modules of the prototype implementation are then evaluated in the later sections. Section 6.2 displays some characteristics for the individual SPEC benchmarks. Section 6.3 contains the evaluation of libdetox, the prototype implementation of the secure execution platform. This section shows that the secure loader is performance competitive to the standard loader and that the sandbox incurs tolerable execution overhead. Section 6.4 evaluates the control flow integrity checks using the information from the Executable and Linkable Format (ELF) headers and symbol tables. Section 6.5 specifies several system call policies and includes details of the different overheads for the system call redirection and authorization framework. A policy for the Apache web server is then evaluated in detail in Section 6.6.

6.1 Security evaluation

It is important to evaluate the effective security properties of both the *theoretical foundations* of the secure execution platform and *libdetox*, the prototype implementation.

The secure loader ensures the safe instantiation of the sandbox and propagates information about the application (symbol locations and executable regions) to the trusted domain. The sandboxing component relies on absolute control over the control flow of the application. Every control flow transfer (both direct and indirect) is checked according to the control flow integrity rules. This ensures that the secure execution platform is always in control of the translated application. The policy-based system call authorization layer ensures system call integrity on a per-application basis.

Section 6.1.1 explains the sandbox principle that ensures that the sandbox stays in control of the sandboxed application. Section 6.1.2 discusses how the secure execution framework (and the libdetox prototype implementation) can guarantee dynamic control flow integrity. The following two sections analyze how the secure execution framework reacts to code-based attacks (Section 6.1.3) and data-based

attacks (Section 6.1.4). Section 6.1.5 lists all security-relevant bugs of Apache 2.2 and discusses how the secure execution platform protects against possible attacks. Section 6.1.6 discusses how malicious applications can exploit the secure execution platform.

6.1.1 Sandbox principle

The sandbox principle ensures that the sandbox always stays in control of the control flow of the application under three constraints. The first constraint is that the sandbox can stop the application and redirect the control flow at one point in time to initialize the sandbox, e.g., before the application starts. The second constraint requires that all control flow transfers are redirected to regions that are under the control of the sandbox, e.g., to translated basic blocks in a code cache or to trampolines that will translate a given untranslated basic block upon execution. The third constraint implies that an application cannot use non-control flow instructions to redirect control flow or that these instructions are properly handled, e.g., system calls and interrupts.

If all three constraints are fulfilled then an application cannot escape out of the sandbox. After the sandbox gets control of the application (1st constraint) all outgoing edges of basic blocks are translated and redirected to code controlled by the sandbox (2nd constraint). In addition all privileged instructions fault into sandbox code (3rd constraint).

6.1.2 Dynamic control flow integrity

The sandbox principle guarantees that the secure execution framework stays in control of the control flow of the application. All control flow transfers are translated and mapped by the sandbox. All static control flow transfers are verified during the translation and mapped to the translated counterparts. All dynamic control flow transfers, i.e., return instructions, indirect jumps and indirect calls, are translated to a runtime lookup that checks and verifies the current target dynamically.

The sandbox protects the application from control flow redirection using overwritten return address on the stack by adding a secure shadow stack in the trusted domain (see Section 5.3.4). All call instructions push the original `eip` to the stack and to the shadow stack. Return instructions read the `eip` from the shadow stack and return to the given translated target. This setup ensures that control flow transfers cannot be redirected using stack-based exploits like, e.g., ROP (see Section 2.2.2).

In addition to the shadow stack dynamic control flow integrity uses symbol and import/export information from the secure loader. Function calls to other libraries may only target exported functions in the library that are imported at the call origin. Function calls inside the same object may only target exported functions if

the symbol table is available. Jump instructions may only target valid code locations inside the current function. If no precise symbol table is available in an object then the model is less precise and local calls and local jump may target any instruction in the current object.

Dynamic control flow integrity limits attacks that rely on control flow redirection to valid exported functions that are reachable in the current object for call instructions and valid instructions in the current function for jump instructions.

6.1.3 Code-based attacks

Code-based attacks include code injection attacks, JOP attacks, and ROP attacks. All code-based attacks need to redirect the control flow of the application by either overwriting a return instruction pointer or a location used for an indirect jump or indirect call (e.g., either a register or a memory location).

Section 6.1.2 guarantees that the control flow of the application cannot be redirected arbitrarily by an attack. Call instructions must target functions and jump instructions only transfer control inside a defined function. Code-based attacks are therefore detected when the exploit has set up the data-structures and tries to execute the initial control flow transfer that starts the malicious code.

We implemented sample programs that are prone to code injection, JOP, and ROP as well as sample exploits that redirect control flow based on a return instruction pointer redirection and function pointer redirection. The exploits execute successfully for unprotected applications. The libdetox prototype implementation catches all attacks before the exploit can do any harm.

6.1.4 Data-based attacks

Data-based attacks are hard to catch without information flow tracking. Our secure execution platform uses a coarse-grained system call-based approach to control all external changes that an application executes.

System call-based security policies stop attacks at the system call level and not at the control flow level. A per-application policy defines a set of system calls and corresponding parameters that are allowed for each application. All other system calls (or parameters) result in an exception. The static policy handles simple system calls; whereas most system calls are simple. In addition, validation functions are executed in the context of the sandbox for complex system calls (e.g., for `mmap` to check that the mapped region does not overlap with the memory used for the sandbox).

A data-based attack can execute any system call that is in the system call policy. The integrity of the system is tied to a least privilege principle and relies on a tight security policy.

We implemented the policy-based system call authorization and added validation functions for system calls like `mmap` and `fork`. We tested the implementation by executing programs that execute different system calls that are part of the policy as well as system calls that violate the policy. The sandbox successfully stops the execution of an application as soon as an illegal system call is executed.

6.1.5 Case study of real attacks

Apache 2.2 is a mature web-server for medium to large scale web sites. Between the first release of version 2.2.0 on December 1st 2005 and the current release 2.2.22 on January 31st 2012, 45 security related bugs are reported[1]. The bugs are fixed in later releases. This study looks at all security-related bugs for Apache 2.2 until the (current) 2.2.22 release. The bugs are classified according to their security impact. Possible impact levels for bugs are[2] *low*, *moderate*, *important*, and *critical*. This study analyzes the security patches and evaluates if the libdetox sandbox can stop an attack.

Table 6.1 shows the different Apache bugs, CVE number, a bug description, the impact of the bug, and if libdetox guarantees the integrity of the system. The following list shows the four libdetox protection categories:

SE: a *security exception* is detected in the code and the thread or process is *gracefully terminated* with a warning message.

SF: an illegal memory access that caused an *segmentation fault* is caught by the sandbox and the thread or process is *gracefully terminated* with a warning message.

PE: a *policy exception* against the per-application policy is detected and the thread or process is *gracefully terminated* with a warning message.

NA!: this bug is *not applicable* and libdetox cannot protect from this bug due to constraints of the secure execution platform.

PI!: libdetox does not protect from this bug due to a limitation of the *prototype implementation*.

Table 6.1 and Table 6.2 use the following abbreviations for bug types: DoS denial of service; EXE arbitrary code execution; IL information leak; lDoS local denial of service; XSS cross-site scripting; CSRF cross-site request forgery; HBUF heap buffer underflow; lPE local privilege escalation; HBOF heap buffer overflow; ACI arbitrary command injection; AIH access to internal hosts; IOF integer overflow.

[1] The Apache homepage lists the security relevant bugs of the 2.2 release at `http://httpd.apache.org/security/vulnerabilities_22.html`.

[2] The Apache homepage `http://httpd.apache.org/security/impact_levels.html` gives a description of the different impact levels.

Fixed	CVE number	Description	Impact	Type	Libdetox
2.2.2	CVE-2005-3352	mod_imap referer XSS	moderate	XSS	NA!
2.2.2	CVE-2005-3357	mod_ssl access control	low	DoS	SF (null)
2.2.3	CVE-2006-3747	mod_rewrite off-by-one	important	EXE	SE, SF, or PE
2.2.6	CVE-2007-1863	mod_cache proxy	moderate	DoS	SF (null)
2.2.6	CVE-2007-1862	mod_cache info. leak	moderate	IL	NA!
2.2.6	CVE-2007-3304	signals to arbitrary pids	moderate	lDoS	PE
2.2.6	CVE-2006-5752	mod_status XSS	moderate	XSS	NA!
2.2.6	CVE-2007-3847	mod_proxy crash	moderate	DoS	SF
2.2.8	CVE-2007-5000	mod_imagemap XSS	moderate	XSS	NA!
2.2.8	CVE-2007-6388	mod_status XSS	moderate	XSS	NA!
2.2.8	CVE-2007-6421	mod_proxy_balancer XSS	low	XSS	NA!
2.2.8	CVE-2007-6422	mod_proxy_balancer DoS	low	DoS	SF (null)
2.2.8	CVE-2008-0005	mod_proxy_ftp UTF-7 XSS	low	XSS	NA!
2.2.9	CVE-2008-2364	mod_proxy_http DoS	moderate	DoS	PI! (mem)
2.2.9	CVE-2007-6420	mod_proxy_balancer CSRF	low	CSRF	NA!
2.2.10	CVE-2008-2939	mod_proxy_ftp XSS	low	XSS	NA!
2.2.10	CVE-2010-2791	mod_proxy_http timeout IL	important	IL	NA!
2.2.12	CVE-2009-0023	APR-util heap underwrite	moderate	HBUF	SE, SF, or PE
2.2.12	CVE-2009-1955	APR-util XML DoS	moderate	DoS	PI! (mem)
2.2.12	CVE-2009-1956	APR-util off-by-one ovfl.	moderate	DoS	SF
2.2.12	CVE-2009-1195	AllowOverride bypass	low	lPE	PE
2.2.12	CVE-2009-1891	mod_defalte DoS	low	DoS	NA! (cpu)
2.2.12	CVE-2009-1191	mod_proxy_ajp info. leak	important	IL	NA!
2.2.12	CVE-2009-1890	mod_proxy rev. proxy DoS	important	DoS	NA! (cpu)
2.2.13	CVE-2009-2412	APR apr_palloc heap ovfl.	low	HBOF	SE, SF, or PE
2.2.14	CVE-2009-2699	Solaris pollset DoS	moderate	DoS	PI! (sig)
2.2.14	CVE-2009-3095	mod_proxy_ftp cmd. inject.	low	ACI	PE
2.2.14	CVE-2009-3094	mod_proxy_ftp DoS	low	DoS	SF (null)
2.2.15	CVE-2010-0408	mod_proxy_ajp DoS	low	DoS	NA! (blocked)
2.2.15	CVE-2010-0434	mode_headers DoS	low	DoS/IL	SF (null)
2.2.15	CVE-2010-0425	mod_isapi unload flaw	important	EXE	PI! (Win)
2.2.16	CVE-2010-1452	mod_cache & mod_dav DoS	low	DoS	SF
2.2.16	CVE-2010-2068	mod_proxy_http timeout	important	IL	NA! (Win)
2.2.17	CVE-2010-1623	apr_brigade_split_line DoS	low	DoS	PI! (mem)
2.2.17	CVE-2009-3560	expat DoS	low	DoS	SF (null)
2.2.17	CVE-2009-3720	expat DoS	low	DoS	SF (null)
2.2.19	CVE-2011-0419	apr_fnmatch flaw	moderate	DoS	PI! (mem)
2.2.20	CVE-2011-3192	range header DoS	important	DoS	NA! (cpu)
2.2.21	CVE-2011-3348	mod_proxy_ajp DoS	moderate	DoS	NA! (blocked)
2.2.22	CVE-2011-3368	mod_proxy rev. proxy	moderate	AIH	PE
2.2.22	CVE-2012-0053	cookie exposure	moderate	IL	NA!
2.2.22	CVE-2011-4317	mod_proxy rev. proxy	moderate	AIH	PE
2.2.22	CVE-2012-0031	scoreboard parent DoS	low	DoS	SF (null)
2.2.22	CVE-2012-0021	mod_log_config crash	low	DoS	SF (null)
2.2.22	CVE-2011-3607	mod_setenvif priv. escal.	low	IOF	SE, SF, or PE

Table 6.1: List of all Apache 2.2 security bugs until version 2.2.22.

Table 6.2 summarizes the information in Table 6.1 and groups all Apache bugs according to libdetox's protection classes. Libdetox protects from 21 of 45 Apache bugs. Out of the 18 bugs that are not applicable to libdetox 5 bugs are denial of service attacks that cause CPU exhaustion or block a thread until a timeout is reached; one bug is only applicable to Windows systems; and 13 bugs are either information leaks, cross-site request forgery, or cross-site scripting. These 13 bugs are out of scope for a combined control flow-based and system call-based approach.

The prototype implementation of libdetox cannot protect from 6 bugs: 4 bugs result in memory exhaustion (which could be checked using additional `mmap` rules), 1 bug enables malicious signals to other local processes on Solaris operating systems,

Libdetox	Vulnerabilities	Total
NA!	CSRF (1), DoS (5)[a], IL (4), IL: Win. only (1), XSS (7)	18
PI!	DoS: mem exhaust (4), DoS: sig (1), EXE: Win. only (1)	6
	Sum of unprotected vulnerabilities	**24**
SF	DoS: null (10), DoS: unbound memory read (2)	12
PE	lDoS (1), lPE (1), ACI (1), AIH (2)	5
SE, SF, or PE	EXE (1), HBUF (1), HBOF (1), IOF (1)	4
	Sum of protected vulnerabilities	**21**

[a]No DoS attack results in an exploitable state. Three bugs use a large amount of CPU to finish bound loops, the other two bugs block a thread until a timeout is reached.

Table 6.2: Summary of the Apache 2.2 bug study.

and 1 bug enables arbitrary code execution on Windows operating systems.

Summarizing the limitations of the secure execution platform shows that 3 bugs are not protected due to different operating systems (Windows or Solaris instead of Linux), 12 bugs result in information leaks that can only be protected with information flow tracking, 5 bugs result in denial of service that is not detectable in a secure execution framework (the bugs either consume large amounts of CPU time or block until a timeout is triggered), and 4 bugs are due to limitations in the prototype implementation. None of these attacks endanger the integrity of the system and an attacker cannot execute arbitrary code.

The fault handler of the secure execution platform catches all memory exceptions (segmentation faults). 12 bugs cause null pointer dereferences and 2 bugs cause unbound memory reads. These bugs are caught using the fault handler which gracefully terminates the application and flags a warning to the administrator. 5 bugs are caught by the per-application policies; the bugs include arbitrary command execution, local privilege escalation, and unauthorized access to hosts on the internal network. 4 bugs can be used to implement arbitrary code execution through ROP, JOP, or code injection. Libdetox catches all control-flow related bugs and terminates the application.

6.1.6 Malicious applications

The execution model for applications inside the sandbox of the secure execution platform assumes *untrusted but non-malicious applications*, i.e., an application may contain bugs but not malicious code. This section discusses potential attacks against the integrity of the sandbox for malicious applications.

A malicious application may execute any code in the sandbox but the system call interposition layer binds the application to the given per-application policy. All protected applications run with user privileges and an attacker can execute only

Benchmark	Instructions	(BBs)	Function calls	(inlined)	Ind. jumps	Ind. calls
400.perlbench	248,753	(54,288)	21,908,972,069	(9.50%)	21,929,721,015	3,902,298,779
401.bzip2	72,261	(11,809)	6,686,411,394	(0.00%)	1,870,766	1,867
403.gcc	2,244,313	(500,030)	11,416,485,263	(2.82%)	5,040,160,816	653,553,951
429.mcf	8,207	(1,467)	6,937,298,559	(0.05%)	1,708,666	574,634
445.gobmk	540,224	(99,735)	17,818,856,119	(1.33%)	117,474,377	185,811,452
456.hmmer	19,667	(3,523)	219,205,278	(26.78%)	163,062,857	1,138,876
458.sjeng	16,604	(3,337)	21,939,941,742	(1.25%)	10,992,904,415	5,070,023,325
462.libquantum	8,209	(1,230)	1,762,122,564	(0.00%)	979	209
464.h264ref	161,107	(22,052)	9,148,416,877	(30.36%)	2,316,733,272	28,445,058,103
471.omnetpp	50,398	(10,281)	17,282,090,091	(19.29%)	3,151,849,446	2,733,835,044
473.astar	27,755	(4,698)	17,389,970,212	(31.63%)	10,809,621	4,996,462,097
483.xalancbmk	158,461	(27,210)	19,825,915,425	(13.87%)	2,426,718,169	9,161,983,117
433.milc	20,237	(2,750)	6,707,912,535	(1.43%)	11,999,314	3,856,839
435.gromacs	41,025	(5190)	3,510,490,537	(75.48%)	27,444,253	3,274,839
436.cactus.	44,365	(7,371)	1,670,570,147	(0.53%)	1,650,753,737	223,565
444.namd	36,545	(3,790)	33,516,882	(20.47%)	14,909,541	1,969,176
447.dealII	103,520	(13,784)	52,965,608,272	(54.00%)	21,163,171,969	540,898,547
450.soplex	79,152	(11,606)	2,482,695,709	(3.54%)	1,722,689,963	27,488,899
453.povray	64,054	(11,368)	18,698,952,109	(8.10%)	433,067,223	7,072,491,358
454.calculix	92,054	(13,964)	5,200,465,040	(25.43%)	502,727,282	11,226,880
470.lbm	6,912	(1099)	5,273,650	(49.98%)	2,627,289	3,724
482.sphinx3	31,403	(5,308)	7,489,332,386	(10.54%)	268,757,951	6,126,858

Table 6.3: Per benchmark number of translated instructions, translated basic blocks, number of function calls, percentage of calls that are inlined, number of indirect jumps, and number of indirect calls.

system calls with local user privileges that are part of the policy for the malicious application.

A user can use the secure execution platform with a tight system call policy to test untrusted applications. According to the sandboxing principle in Section 6.1.1 the malicious application cannot escape out of the sandbox. The control flow inside the application is unchecked but all system calls must conform to the supplied policy.

6.2 SPEC CPU 2006 characteristics

The SPEC CPU 2006 version 1.0.1 benchmarks cover a wide range of applications. These benchmarks have many different code patterns. An efficient binary translator must be aware of the most common code sequences and must optimize for these cases. The performance information was obtained using gcc version 4.1.3 and glibc version 2.6-22. The benchmarks were compiled with `-O2`.

Table 6.3 shows the characteristics of the individual SPEC CPU 2006 benchmarks. The three benchmarks gcc, perlbench, and gobmk have a relatively large code base where many instructions are translated during the benchmark run. Most of the overhead results from the following four different sources:

- **Function pointers:** Calling a function through a function pointer results in an indirect call. Every indirect call leads to the execution of overhead instruc-

tions. The memory location of the function pointer is read and the target is then mapped into the trace cache. To reduce this overhead the last target of the function pointer is saved if the indirect call prediction optimization is activated. If the function pointer changes, then the binary translator must do an additional lookup for the new address.

- **Return instructions:** Every function call incurs overhead due to the indirect control flow transfer in the translated return instruction. The cost of function calls can be decreased through inlining. The dealII benchmark shows nicely that the overhead can be reduced through inlining; the binary translator does not pay the additional cost for a function call in 54% of the calls.

- **Jump tables (switch):** Switch constructs or jump tables are another source of overhead. The C compiler translates switch constructs into indirect jumps. The targets of the memory locations do not change, but the binary translator must add and execute an additional lookup for each of these jumps.

Each of the four worst performing benchmarks (perlbench, sjeng, dealII and povray) include multiple of the above mentioned factors. The most dominant overhead is the combined effect of all translated indirect control flow transfers. If indirect control flow transfers are executed often, and if they cannot be replaced by some optimization (e.g., inlining), or if the cost cannot be reduced through some optimization (e.g., predictions) then they will induce overhead.

6.3 Libdetox performance evaluation

This section evaluates and discusses the implementation prototype of the secure execution platform to demonstrate its practicability. The evaluation shows a performance evaluation for the SPEC CPU benchmarks and discusses limitations of the current secure loader implementation.

The evaluation uses the SPEC CPU 2006 benchmarks version 1.0.1 to evaluate both performance and feasibility of our prototype implementation. The SPEC benchmarks are run on an Intel Core i7 CPU at 3.07GHz in a VirtualBox virtual appliance using a single dedicated core. The operating system is Ubuntu Natty Narwhal 11.04 with gcc version 4.5.2-8ubuntu4 on a x86 ia32 kernel with glibc version 2.13.

The OpenOffice measurements are run on an Intel Core i7 CPU at 3.07GHz on Ubuntu Maverick 10.10 using glibc version 2.13 on a 64-bit kernel with 32-bit support. We use OpenOffice version 3.20 (the exact version is OOO320_m18_native_packed-1_en-US.9502).

6.3.1 SPEC CPU benchmarks

Table 6.4 displays the number of relocations per benchmark run and the number of loaded Dynamic Shared Object (DSO)s for a subset of the SPEC CPU2006 benchmarks. The total number of relocations is low (between 1381 and 1597 relocations) for all the evaluated SPEC CPU 2006 benchmarks.

Table 6.5 shows the overhead of the secure loader compared to the standard loader. The performance of the secure loader is competitive to the standard loader. Comparing the columns of the secure loader to the secure loader with memory protection illustrates that the overhead of the secure loader to protect all writable sections except `.data` and `.bss` is negligible. The memory protection adds overhead for every resolved symbol but the execution time that is spent in the loader is low compared to the overall execution time. The cost for protecting the memory pages that contain the loader data for each shared object is amortized during the runtime of the program.

Table 6.5 evaluates three different configurations and compares them to a native execution. The different configurations are:

LnBT: This configuration uses the secure loader without the security guarantees that the sandbox offers.

Benchmark	Relocations	DSOs	Bin. exec.
400.perlbench	1447	3	3
401.bzip2	1368	2	6
403.gcc	1437	3	9
429.mcf	1381	3	1
445.gobmk	1422	3	5
456.hmmer	1431	3	2
458.sjeng	1386	2	1
462.libquantum	1372	3	1
464.h264ref	1423	3	3
473.astar	3995	5	2
433.milc	1379	3	1
434.zeusmp	1602	5	1
435.gromacs	1597	5	1
436.cactusADM	1613	5	1
444.namd	4001	5	1
450.soplex	4244	5	1
459.GemsFDTD	1644	5	1
470.lbm	1377	3	1
482.sphinx3	1429	3	1

Table 6.4: Per benchmark average number of relocations, loaded DSOs, and number of binaries executed in a benchmark run for a subset of the SPEC benchmarks.

nLBT: This configuration evaluates the overhead of the secure sandbox without the secure loader (using an `LD_PRELOAD`-style injection of the sandbox.

LBT: This configuration evaluates the complete secure execution framework using the secure loader to bootstrap the application. All application code is then executed in the sandbox.

The last column displays the overhead of the secure platform (including secure loader, memory protection from Section 5.1.3 and full sandboxing of all application code). Most programs have low overhead and safe execution is feasible. Running all application code in a sandbox and checking all control flow transfers results in additional overhead between 0.5% and 96% for the SPEC benchmarks compared to the standard loader. The overhead results mostly from binary translation, and only little overhead is induced through the additional security checks. Benchmarks with a very high number of indirect control flow transfers (these transfers incur a runtime check in the sandbox) have higher overhead (e.g., 403.gcc, or 464.h264ref). The average overhead for all evaluated benchmarks is 15.4% which is tolerable for the combination of safe loading and sandboxing. Appendix E shows the evaluation of an earlier version of libdetox that focuses on performance over security.

Benchmark	**Native**	**LnBT**	**Ovhd.**	**nLBT**	**Ovhd.**	**LBT**	**Ovhd.**
400.perlbench	404	400	-1.0%	801	98.3%	792	96.0%
401.bzip2	670	665	-0.7%	709	5.8%	708	5.7%
403.gcc	369	372	0.8%	502	36.0%	500	35.5%
429.mcf	443	437	-1.4%	446	0.7%	433	-2.3%
445.gobmk	503	499	-0.8%	680	35.2%	674	34.0%
456.hmmer	576	574	-0.3%	581	0.9%	581	0.9%
458.sjeng	594	593	-0.2%	971	63.5%	961	61.8%
462.libquantum	606	623	2.8%	618	2.0%	597	-1.5%
464.h264ref	817	815	-0.2%	1160	42.0%	1170	43.2%
473.astar	586	579	-1.2%	660	12.6%	660	12.6%
483.xalancbmk	310	301	-2.9%	561	81.0%	572	84.5%
433.milc	500	505	1.0%	516	3.2%	552	10.4%
434.zeusmp	609	612	0.5%	607	-0.3%	608	-0.2%
435.gromacs	923	922	-0.1%	913	-1.1%	915	-0.9%
436.cactusADM	1370	1380	0.7%	1340	-2.2%	1360	-0.7%
444.namd	556	556	0.0%	559	0.5%	558	0.4%
450.soplex	296	297	0.3%	322	8.8%	318	7.4%
459.GemsFDTD	638	638	0.0%	674	5.6%	658	3.1%
470.lbm	368	361	-1.9%	363	-1.4%	362	-1.6%
482.sphinx3	629	592	-5.9%	596	-5.2%	597	-5.1%
Average	588.4	586.1	-0.4%	679.0	15.4%	678.8	15.4%
Geo.mean	550.0	547.0	-0.5%	639.3	16.2%	638.5	16.1%

Table 6.5: Performance analysis of the libdetox prototype implementation.

6.3.2 OpenOffice

We measured OpenOffice version 3.2.0 startup as a stress test and worst-performance metric, 145 DSOs are loaded, relocated, and executed with very low code reuse. OpenOffice was run on an Intel Core i7 CPU at 3.07GHz on Ubuntu Maverick.

OpenOffice 3.2.0 executes 265,067 relocations during the startup phase and loads 145 individual shared objects. The secure loader imposes an overhead of 44% for OpenOffice and 77% overhead for the additional memory protection. If the full protection sandbox and the secure loader are used in combination then the start-up of OpenOffice is slowed down by 188%.

The overhead for OpenOffice results from additional checks that are carried out whenever a new shared object is loaded and all relocation entries need to be resolved. The OpenOffice startup sequence is evaluated as a worst-case scenario. Code is rarely reused and a huge number of references between objects need to be resolved. The overhead for the secure loader comes from less efficient loading and symbol resolving. The additional overhead between the secure loader and the secure loader plus memory protection comes from the additional `mprotect` system calls used to protect all runtime sections except `.data`, `.bss`, `.tdata`, and `.tbss`.

6.4 Control flow integrity using ELF information

This section describes the static and dynamic overhead for translated and checked control flow transfers in detail. Overhead for each control flow type is analyzed and dissected. Additional information about the generated dynamic control flow graph is discussed as well.

6.4.1 Overhead analysis

The Software-based Fault Isolation (SFI) sandbox imposes low overhead (around 6.4% for a performance optimized version as in Appendix E or around 15.4% for the secure execution framework as in Section 6.3). The overhead for the static origin and destination checks (to verify the control flow transfer rules from Section 4.2.1 during translation) is negligible. The stack is already set up to execute internal functions, and the search in the control transfer lookup model is fast (there are usually not more than 5 to 10 DSOs loaded and there are usually not more than 100 to 200 functions defined per DSO, with a peak of 2,171 entries for the standard libc). The DSOs are chained and must be searched linearly. The individual objects of a DSO are stored in a sorted array structure, this setup enables O(logN) access for each individual object.

Dynamic control flow transfers need a dynamic check every time the translated code is executed. The translator emits a dynamic check alongside the control flow transfer. The additional code per control transfer is depicted in Table 6.6.

```
movl %esp, (tld->stack-1)
movl tld->stack-1, %esp
pushad                          # Save registers
pushl CFTX_CALL_IND             # Handle ind. call
pushl 40(%esp)                  # Pointer to dst.
pushl src                       # Pointer to src.
pushl tld                       # Thread local data
call fbt_check_transfer         # Check transfer
leal 16(%esp), %esp             # Readjust stack
popad                           # Restore registers
popl %esp
jmpl *(tld->ind_target)
```

Table 6.6: Control flow transfer check for an indirect call instruction. The additional code is emitted by the dynamic translator. This code validates a runtime transfer when executed during a dispatch.

The dynamically executed checks add some overhead to the general execution time. But the overhead is barely noticeable even for large interactive applications like OpenOffice.

The following micro benchmarks evaluate individual kinds of control flow transfers and give an overview of general translation and validation overhead. Each micro benchmark tests one form of control flow transfer. The binary translator is started at the beginning, gathers data about the program and runs 1,000,000,000 control flow transfers of each kind. The source code for the micro benchmarks is shown in Listing D.1. We report the average of 5 runs after removing the best and worst result from 7 total runs. Standard deviation is below 2% for all micro benchmarks.

The overall overhead is shown in Table 6.7. All non-control flow instructions and non-privileged instructions are copied verbatim and do not incur additional overhead. Only protected and translated dynamic control flow instructions add overhead. Most applications have few control flow instructions. Experiments show

Control flow type	**Orig.**	**LD**	**Factor**
Indirect Jump	1.11ns	45.3ns	40.9
Indirect Call	2.50ns	37.42ns	15.0
Conditional jump	1.75ns	0.62ns	0.35
Call	1.61ns	1.91ns	1.19
Jump	0.61ns	1.61ns	2.64
Function return	8.54ns	41.18ns	4.82

Table 6.7: Overhead per control flow transfer type for dynamic CFI. Orig. shows the time for an unchecked run; LD accounts for the total time in the libdetox framework. Factor shows the overhead factor between untranslated and sandboxed execution.

that the average overhead for user-space virtualization is between 6% and 9% depending on the additional security guards [PG11]. The differences between static and dynamic control flow transfers are visible in Table 6.7.

Static control flow transfers (call, jump, and conditional jump) where the target is known at translation time incur only minimal overhead or even result in a speedup. This speedup is due to better code layout and trace linearization during the translation process. The target is validated during the translation process, no additional dynamic check is needed during the execution of the code.

Dynamic control flow transfers (indirect jumps, indirect calls, and function returns), on the other hand, result in an online lookup whenever the translated code is executed. Fortunately these dynamic control flow transfers are less frequent than other instructions in applications. There exist several opportunities for optimization of dynamic control flow transfers (e.g., lookup caching, or special lookup tables).

6.4.2 Control transfer graph analysis

Figure 6.1 shows the complete graph with all control flow transfers (direct and indirect) for the small program listed in Listing 6.1. The graph of this simple application visualizes the large number of control flow transfers (even in a small demo program) that must be validated in an online protection framework. Such validation is only manageable with efficient and carefully designed data structures. Libdetox proposes a simple yet effective strategy to peek into the application and to analyze every control flow transfer according to the control flow transfer model.

These visualizations of the control transfer graphs are built as a side effect of the validation and can be recorded using additional instrumentation. Multiple transfers between the same nodes collapse into one edge. The graph contains all symbols from the application including those that are imported from standard libc. Small circles are transfers through PLT tables of shared objects. The two light green nodes (`foo` and `main`) are the functions from the application source of Listing 6.1; the remaining nodes are libC functions used for input and output and functions that are used during the application startup and tear-down.

```
#include <stdio.h>
int foo() {
  printf("We are in foo\n");
  return 0;
}
int main(int argc, char *argv[]) {
  foo();
  printf("Successfully executed foo\n");
  return 0;
}
```

Listing 6.1: Small demo program with two functions

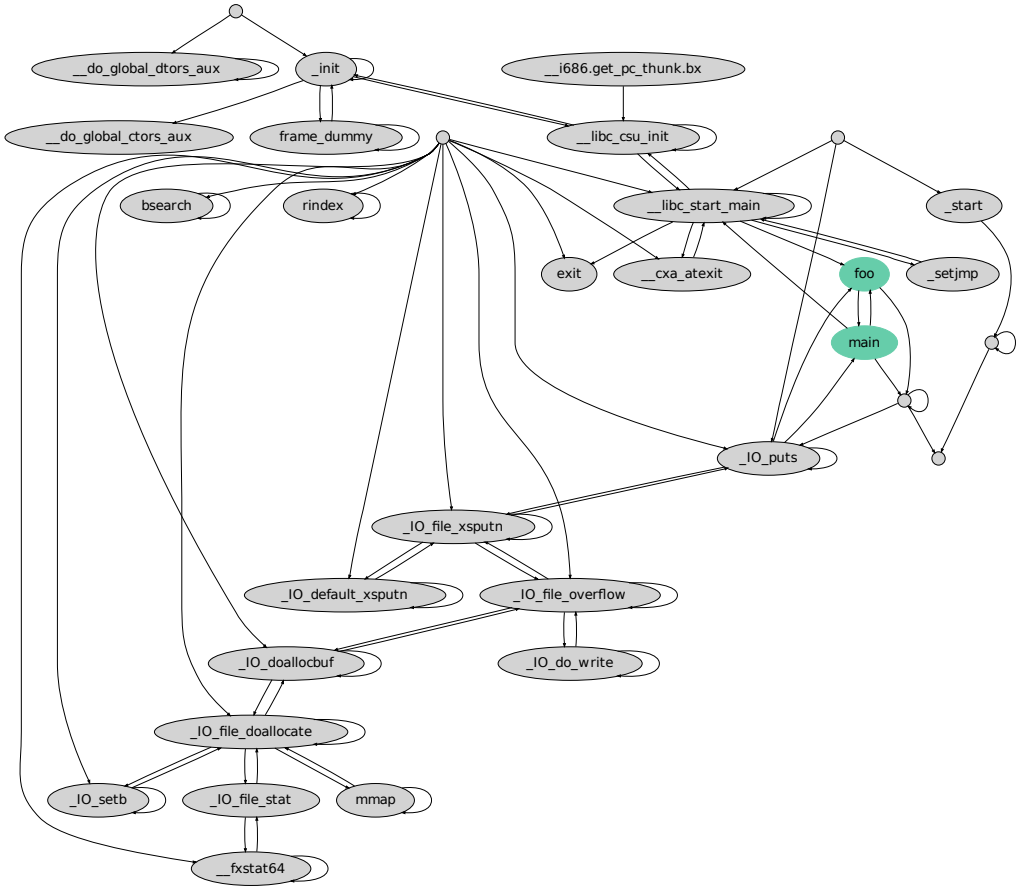

Figure 6.1: Overview of a graph with all control flow transfers of a simple application including all dynamic loading steps that are translated as a side effect. The small circles indicate PLT entries that are dereferenced using indirect jumps.

6.5 System call policies

Listing 6.2 shows a relaxed policy that covers all SPEC CPU2006 benchmarks. This policy is not secure and only used to evaluate the overhead of policy-based user-space SFI. Some rules are relaxed to facilitate the run of the SPEC internal benchmark scripts[3]. The policy is a summary of all individual policies for each SPEC benchmarks so that the overhead for all benchmarks can be evaluated in a single run of the SPEC benchmark script. Differences to real policies include the over-generalization of attributes and the lax handling of the `open`, `unlink`, `mmap2`, `unlink`, and `stat64` system calls. A production policy tightens the policy for a single program and explicitly lists all needed files and directories or restricts these

[3]The SPEC internal benchmark scripts are used internally to setup the environment for the SPEC benchmarks. The execution time of these scripts is not part of the overall benchmark result.

system calls to specific directories. These system calls are used to access many data files in each individual benchmark and for each data size. The long list of explicit configurations was abbreviated through over-approximation to give a cleaner picture. A safe policy for one specific SPEC benchmark does not result in any measurable additional overhead.

Listing 6.2 shows that all SPEC CPU2006 benchmarks need no more than 38 different system calls with a few more individual parameter configurations.

```
/* not listed: abort */
mode:whitelist
/* memory management */
brk(*):allow
mmap2(null,*, PROT_READ | PROT_WRITE,
  MAP_ANONYMOUS | MAP_PRIVATE, -1,*):allow
mremap(*,*,*, MREMAP_MAYMOVE):allow
munmap(*,*):allow
/* allowed prog.s */
execve("/bin/echo",*,*):allow
execve("/opt/cpu2006/bin/echo",*,*):allow
execve("/sbin/echo",*,*):allow
execve("/usr/bin/echo",*,*):allow
execve("/usr/local/bin/echo",*,*):allow
execve("/usr/local/sbin/echo",*,*):allow
execve("/usr/sbin/echo",*,*):allow
/* allowed file I/O */
clone(*,null,0,null):allow
close(*):allow
dup(*):allow
fcntl64(*,*):allow
fstat64(*,*):allow
ftruncate64(*,*):allow
getcwd("*",*):allow
ioctl(*,*):allow
llseek(*,*,*,*, SEEK_SET):allow
llseek(*,*,*,*, SEEK_CUR):allow
lseek(*,*, SEEK_SET):allow
lseek(*,*, SEEK_CUR):allow
lstat64("/opt/cpu2006/benchspec/" \
  "CPU2006/*", *):allow
/* relaxed for spec */
open("*",*):allow
pipe(*):allow
read(*,*,*):allow
stat64("*",*):allow
/* remove foo directories */
rmdir("foo"):allow
/* unlink relaxed for spec */
unlink("*"):allow
write(*,*,*):allow
writev(*,*,*):allow
/* process mgmt */
futex(*, FUTEX_PRIVATE | FUTEX_WAKE,
  0x7FFFFFFF, null,*,*):allow
waitpid(*,*,0):allow
/* signals /*
rt_sigprocmask(SIG_BLOCK,*,*):allow
rt_sigprocmask(SIG_SETMASK,*,null):allow
/* information retrieval*/
getegid32():allow
geteuid32():allow
getgid32():allow
getrusage(RUSAGE_Self,*):allow
gettimeofday(*,null):allow
getuid32():allow
setrlimit(RLIMIT_DATA,*):allow
ugetrlimit(RLIMIT_DATA,*):allow
/* sleep and time */
nanosleep(*,*):allow
time(null):allow
times(*):allow
```

Listing 6.2: Policy for the SPEC CPU2006 benchmarks

The second case study shows a policy for *nmap*, which is a network exploration and security tool that checks and fingerprints running services of servers over the Internet. libdetox virtualizes and encapsulates version 4.53 of nmap into a secure sandbox. The policy in Listing 6.3 shows a set of rules that restricts nmap to a few different system calls, e.g., opening any network connection. The policy allows network access and access to a set of libraries and configuration files. The nmap program uses 23 different system calls; individual parameters are used to `open` 15 different files, use `stat64` on 6 files, and use `access` for two files. The parameters of the `fcntl64` and `ioctl` calls need to be checked in detail in a specific handler function. This policy sandboxes the network scanner, and an attacker cannot escalate privileges if one of the many nmap detection modules contains exploitable code.

```
/* not listed: abort */
mode:whitelist
/* memory management */
brk(*):allow
/* due to shared libraries all mmap
   calls  must be additionally checked
   in a handler function for a set
   exec bit. */
mmap2(*,*,*,*,*,*):allow
munmap(*,*):allow
/* thread futexes */
futex(*,*,*,*,*,*):allow
/* limit I/O */
access("/etc/ld.so.nohwcap",*):allow
access("/usr/share/nmap/" \
  "nmap-services",*):allow
close(*):allow
fcntl64(*, F_GETFL):allow
fcntl64(*, F_GETFD):allow
fcntl64(*, F_SETFL, O_RDWR |
  O_NONBLOCK):allow
fstat64(*,*):allow
ioctl(*, TIOCGPGRP, *):allow
llseek(*,*,*,*,*):allow
newselect(*,*,*,*,*):allow
open("/dev/arandom",*):allow
open("/dev/tty",*):allow
open("/dev/urandom",*):allow
open("/etc/host.conf",*):allow
open("/etc/hosts",*):allow
open("/etc/ld.so.cache",*):allow
open("/etc/localtime",*):allow
open("/etc/nsswitch.conf",*):allow
open("/etc/passwd",*):allow
open("/etc/resolv.conf",*):allow
open("/lib/i686/cmov/" \
  "libnsl.so.1",*):allow
open("/lib/i686/cmov/" \
  "libnss_compat.so.2",*):allow
open("/lib/i686/cmov/" \
  "libnss_files.so.2",*):allow
open("/lib/i686/cmov/" \
  "libnss_nis.so.2",*):allow
open("/usr/share/nmap/" \
  "nmap-services",*):allow
read(*,*,*):allow
stat64("/etc/localtime",*):allow
stat64("/etc/resolv.conf",*):allow
stat64("/home/test/.nmap/" \
  "nmap-services",*):allow
stat64("./nmap-services",*):allow
stat64("/usr/lib/nmap/" \
  "nmap-services",*):allow
stat64("/usr/share/nmap/" \
  "nmap-services",*):allow
write(*,*,*):allow
/* net */
socketcall(PF_NETLINK, SOCK_RAW, 0):allow
socketcall(PF_INET, SOCK_STREAM,
  IPPROTO_TCP):allow
socketcall(PF_FILE, SOCK_STREAM |
  SOCK_CLOEXEC | SOCK_NONBLOCK, 0):allow
/* system information */
geteuid32():allow
gettimeofday(*,*):allow
getuid32():allow
time(*):allow
uname(*):allow
ugetrlimit(*,*):allow
setrlimit(RLIMIT_NOFILE, *):allow
```

Listing 6.3: Policy for the nmap network scanner

Benchmark	**native**	**BT**	**Ovhd.**	**libdetox**	**Ovhd.**
test.html	84.83s	97.47s	14.9%	101.34s	19.5%
	22.48Mb/s	19.57Mb/s		18.82Mb/s	
phpinfo.php	84.40s	98.63s	16.9%	101.34s	20.1%
	3.28Mb/s	2.8Mb/s		2.73Mb/s	
picture.png	249.87s	261.92s	4.83%	266.98s	6.85%
	945.18Mb/s	901.67Mb/s		884.6Mb/s	
Avg. overhead	-		9.29%		12.06%

Table 6.8: The *ab* benchmark is used to compare a native run without isolation to fast binary translation only, and libdetox with user-space fault isolation and policy-based system call authorization.

6.6 Apache case study

The Apache 2.2.11 HTTP server is used to benchmark a daemon that needs both access to local files and is accessible over the network. libdetox encapsulates the Apache processes and threads and only allows few system calls with restrictive parameter configurations. Like the nmap policy in Listing 6.3 Apache is allowed to open only specific files and access files in two directories (`/etc/apache2`, and `/var/www`) and is not allowed to execute other processes. On the other hand, the daemon process is free to open connections over the network. Listing 6.4 lists the full Apache policy.

```
mode:whitelist
access("/etc/ld.so.nohwcap",*):allow
brk(*):allow
chdir("/"):allow
chdir("/var/www"):allow
clone(*,*,*,*):allow
close(*):allow
dup2(*,*):allow
epoll_create(*):allow
epoll_ctl(*,*,*,*):allow
exit_group(*):allow
fcntl64(*,*):allow
fstat64(*,*):allow
futex(*,*,*,*,*,*):allow
getcwd("*",*):allow
getdents(*,*,*):allow
geteuid32():allow
getpgrp():allow
ioctl(*,*):allow
ipc(*,*,*,*,*,*):allow
kill(*,*):allow
llseek(*,*,*,*,*):allow
lstat64("/etc/*",*):allow
lstat64("/var",*):allow
lstat64("/var/www",*):allow
lstat64("/var/www/phpinfo.php",*):allow
mmap2(*,*,*,*,*,*):allow
munmap(*,*):allow
newselect(*,*,*,*,*):allow
open("./*",*):allow
open("/dev/null",*):allow
open("/dev/urandom",*):allow
open("/etc/apache2/*",*):allow
open("/etc/gai.conf",*):allow
open("/etc/group",*):allow
open("/etc/host.conf",*):allow
open("/etc/hosts",*):allow
open("/etc/ld.so.cache",*):allow
open("/etc/mime.types",*):allow
open("/etc/nsswitch.conf",*):allow
open("/etc/passwd",*):allow
open("/etc/php5/*",*):allow
open("/etc/protocols",*):allow
open("/etc/resolv.conf",*):allow
open("/lib/*",*):allow
open("/proc/*",*):allow
open("/usr/lib/*",*):allow
open("/usr/share/file/" \
  "magic.mime",*):allow
open("/usr/share/zoneinfo*",*):allow
open("/var/log/apache2/*",*):allow
open("/var/www/html.html",*):allow
open("/var/www/phpinfo.php",*):allow
pipe(*):allow
poll(*,*,*):allow
read(*,*,*):allow
rt_sigprocmask(*,*,*):allow
sendfile(*,*,*,*):allow
setgid32(*):allow
setgroups32(*,*):allow
setsid():allow
setuid32(*):allow
socketcall(*,*):allow
stat64("/etc/apache2*",*):allow
stat64("/etc/php5*",*):allow
stat64("/etc/resolv.conf",*):allow
stat64("/lib*",*):allow
stat64("/usr/lib/",*):allow
stat64("/usr/share/zoneinfo*",*):allow
stat64("/var/www/",*):allow
stat64("/var/www/html.html",*):allow
stat64("/var/www/phpinfo.php",*):allow
time(*):allow
umask(*):allow
uname(*):allow
unlink("/etc/apache2/*"):allow
waitpid(*,*,*):allow
write(*,*,*):allow
```

Listing 6.4: Policy for the Apache web server

The overhead for the Apache daemon was measured using the *ab* Apache benchmark, which uses 10 concurrent instances to receive each file 1,000,000 times. Table 6.8 shows the overheads using different configurations. The benchmark uses the following files: (i) `test.html`, a static html file with 1.7kB, (ii) `phpinfo.php`, a small php file that issues the phpinfo call, and (iii) `picture.png`, a larger binary file with 242kB.

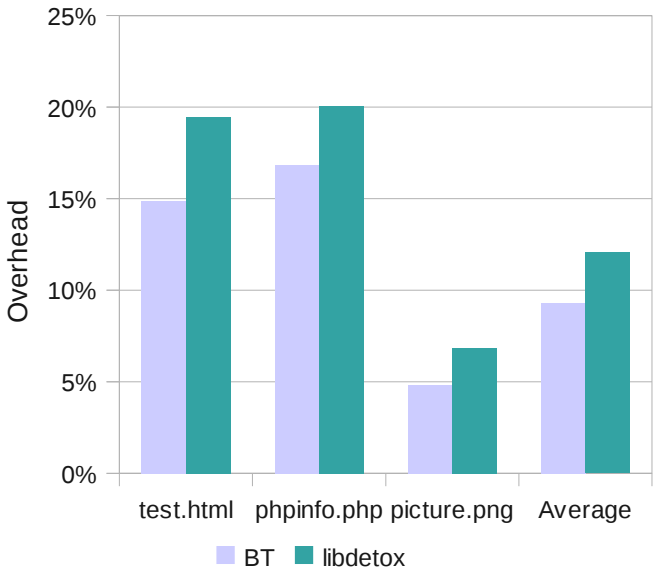

Figure 6.2: Overhead introduced through isolation, sandboxing, and policy-based authorization for the Apache benchmarks.

The overhead to download a small file is 14.9% for binary translation because of the high number of system calls needed to open, read, and send the file. The overhead for libdetox is 19.5%. For larger files, as seen by the numbers for `picture.png`, the overhead of binary translation is 4.83% and 6.85% for libdetox.

An interesting feature is the throughput difference between small and large files. Throughput is increased from 22.48Mb/s to 945.18Mb/s for native runs and even more for libdetox, namely from 19.57Mb/s to 901.67Mb/s, which is more than 46 times faster compared to the small file.

Figure 6.2 shows the overhead introduced through isolation and sandboxing for the different Apache benchmarks. The numbers show that libdetox introduces a moderate overhead of about 20% for small static and small dynamic files. For larger files the overhead drops to below 7%. The average overhead of libdetox for the Apache web-server is 12.06%, which makes user-space fault isolation attractive.

7

Case study: dynamic race detection

Attacks to local security by privilege escalation are a challenging problem because a potential attacker already has access to some privileges on the same machine. An attacker can use these privileges to exploit local bugs in an application to escalate privileges and/or to impersonate a different user.

File-based race conditions are a typical form of Time Of Check To Time Of Use (TOCTTOU) attacks [Bis95, BD96]. The application process leaves a window of opportunity to the attacker where a specific bug is exploitable. A second process may interfere on a shared resource whereas the original process assumes that it is using the resource exclusively. These interferences can be used to replace accessed files between checks or, e.g., to deadlock privileged processes. As multicore systems become more and more popular, we expect an increase in attacks that employ multiple (concurrently) executing processes. Multicore systems enable an attack to run alongside the original program; the attacking process can then use cache effects, or other timing issues to exploit a process.

In a Unix-like system file accesses are particularly prone to race conditions. Potential race conditions arise because the mapping from a filename to a unique inode and device number can change. Filenames are volatile and the corresponding inode and device number can change upon every system call invocation. If a path and filename are passed to a system call then the kernel dynamically resolves the path and filename. The kernel then maps the resolved inode and device number (which corresponds to the file at that moment in time when the system call was executed) to a file descriptor that is then passed back to the application. The mapping from inode and device number to a file descriptor is unique and race-free but the mapping from filename to inode and device number is volatile.

File-based race conditions can occur either in the path to the file (i.e., a directory is a symbolic link that can be changed by the attacker), or in the final file atom (i.e., the last file atom is an attacker-controlled symbolic link). The POSIX community tried to address the problem of file-based races by introducing new system calls that work relative to a specific file descriptor [POS08, cor05]. These new system calls can be used to, e.g., test permissions of files relative to a given base directory.

A programmer can use these system calls to, e.g., authorize a directory and then safely create a file atom in that directory provided that the directory is validated appropriately (using, e.g., user-mode path resolution). If used properly, the new system calls solve the problem of race conditions in the path to the final file atom, but they do not solve the problem of race conditions if the final file atom is a symbolic link.

Much work has been done by the system security community to resolve TOCTTOU races [SW91, MK97, VBKM00, CBWKh01, Che02, CW02, KR02, GSV03, TY03, PLLK04, DH04, SCW+05, UJR05, sLC05, JKDC05, WP05, AJ06, WP06, WSSZ07, THWDS08a, THWDS08b, SGC+09, CHV10]. Recently a practical, portable solution has been found that can deterministically solve the problem without requiring modifications to the kernel or its API: user-mode path resolution [THWDS08b]. This solution, however, suffers from a serious drawback: a programmer must manually identify and modify all pairs of system calls that are vulnerable to TOCTTOU races.

DynaRace (the race detection module of the secure execution platform) builds on and extends user-mode path resolution by removing the burden of manual program modification. DynaRace deterministically performs safe user-mode file path resolution[1] for *unmodified* applications by applying some user-defined action to every atom along the file path in a race-free manner. The key idea is that DynaRace defines user-defined actions such that they guarantee that the metadata of file atoms do not change between invocations of vulnerable system call sequences. DynaRace then dynamically intercepts and replaces these vulnerable sequences with their deterministically-safe alternative. To this end, DynaRace maintains a cache holding the metadata of accessed files and a state machine that quickly identifies periods of vulnerability.

DynaRace is an approach that protects unmodified applications from file-based TOCTTOU race conditions. DynaRace adds a hidden layer between the unmodified application and the operating system that keeps state and metadata in a mapping cache for all accessed files. This mapping cache is used to verify the integrity and consistency of consecutive file accesses according to a state machine. Each accessed file has an associated state. The mapping cache is either updated with the current metadata, or the existing metadata is enforced for the accessed file according to the state of the file. Each file can be in four different states: *new* for new files, *update* for files where the metadata is updated between system calls, *enforce* for files where the cached metadata is enforced between system calls, and *retire* for files that are no longer used in the application.

Individual unsafe system calls like `access` followed by `open` are redirected dynamically to race-free implementations. These race-free implementations are part of the DynaRace framework. They check the state of the file according to the internal mapping cache and the state machine. If the system call executes without a race

[1] See [THWDS08b] for a detailed description of user-mode path resolution.

condition then the mapping cache is updated and the result of the system call is returned to the application.

Our implementation prototype uses user-space binary translation and system call interposition to weave a deterministic race protection framework into the application. An important aspect of our implementation prototype is that neither the application nor any library must be aware of the safe system calls. The system call replacement system works in the background of the process without any coordination. The user-space application can use unsafe system calls and our approach ensures that no file system race conditions are possible.

The contributions of this case study are as follows:

1. DynaRace, a lightweight approach to protect unmodified applications from file-based race conditions.
2. An evaluation and discussion of a DynaRace prototype implementation.

DynaRace keeps state and metadata for each accessed file in an intermediate layer between the application and the operating system. DynaRace redirects all file-based system calls to a stateful inspection framework. This inspection framework uses a state machine and the cached metadata to validate that the metadata of a file has not changed between system calls that test file properties and system calls that use or change files or file metadata.

7.1 Attack model and background information

This section describes the attack model assumed by the DynaRace approach and presents background information on file-based TOCTTOU race conditions. File-based race conditions enable an attacker to escalate privileges, e.g., to access, read, or write privileged files.

7.1.1 Attack model

An attacker has user-access to the system and tries to escalate privileges. The user has no root-privileges, and the attacker has no direct hardware access. The DynaRace protection runs with the same privileges as the protected application. A successful attack breaches the security of the application and grants additional privileges to the attacker. DynaRace terminates the application if an attack is detected. Denial of service attacks are not part of the attack model.

7.1.2 File-based TOCTTOU race conditions

Table 7.1 shows a classic TOCTTOU race condition presented by Mazières et al. in [MK97]. A privileged Set User ID upon execution (SUID) garbage collector script

cleans the `/tmp` directory. An attacker prepares a temporary directory containing an empty file with the same name like the target file. The script reads the directory and relies on the fact that the directory will not change between individual system calls. The attacker removes the temporary directory and replaces it with a link to an existing directory. The script then deletes the file with the same name in the privileged directory.

SUID program	**Attacker**
	`mkdir("/tmp/x")`
	`creat("/tmp/x/passwd")`
`readdir("/tmp")`	
`lstat("/tmp/x")`	
`readdir("/tmp/x")`	
	`rename("/tmp/x", "/tmp/y")`
	`symlink("/etc", "/tmp/x")`
`unlink("/tmp/x/passwd")`	

Table 7.1: An attacker races against the privileged application and deletes a critical system file.

Table 7.2 shows a typical TOCTTOU race condition presented in [BD96]. The `access` system call is used to check permissions of the user in a SUID program. A privileged application relies on the fact that the mapping from a file to an inode and device id remains the same over subsequent system calls. An attacker replaces the file with a link to a privileged file. The privileged application then uses the sensitive file instead of the original file that was authenticated.

SUID program	**Attacker**
`access("file")`	
	`unlink("file")`
	`link("sensitive", "file")`
`fd = open("file")`	
`read(fd, ...)`	

Table 7.2: An attacker races against the SUID program and reads a privileged file.

7.2 The DynaRace approach

DynaRace adds a hidden abstraction layer between the application and the operating system. This abstraction layer keeps track of all file-based system calls and keeps information about accessed files and directories in a metadata cache. The monitoring of all file-based system calls allows DynaRace to keep track of all directories and

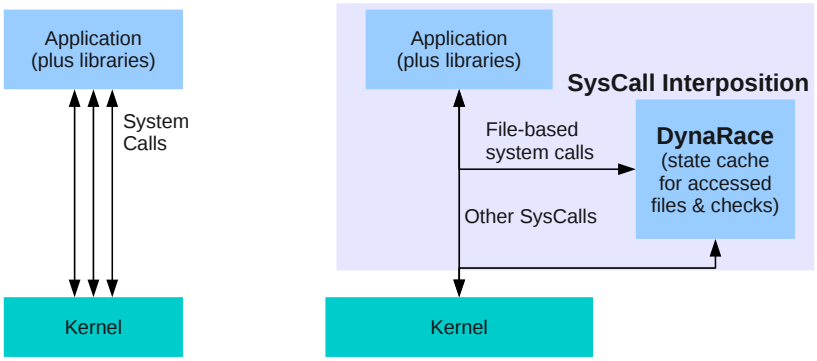

Figure 7.1: Overview of the DynaRace approach.

files that are accessed. Our implementation prototype extends a dynamic system call interposition framework to dynamically monitor all file-related system calls of an application as well as all loaded libraries.

Figure 7.1 shows two system configurations, one configuration without DynaRace and one configuration with DynaRace. Without DynaRace potentially unsafe system calls are executed directly by the kernel without additional checks for race conditions. Each system call is executed in isolation and without knowledge of the results of prior system calls. Applications without special checks (e.g., storing file state internally, or checking for safe accesses using hardness amplification, see Section 7.6.5) for these unsafe system calls are prone to race conditions.

DynaRace intercepts all file-related system calls using system call interposition. DynaRace keeps state for each accessed file and dynamically checks for file-based race conditions. Unsafe system calls like `access` followed by `open` are replaced dynamically with the new and safe race-free sequence of system calls. File-based system calls are no longer executed in isolation but use the state and metadata from prior system calls to validate the current system call. Neither the application nor any library must be aware of the safe system calls. The system call replacement system works in the background of the process without any coordination. The user-space application can use unsafe system calls and DynaRace ensures that no file system race conditions are possible. Due to the replacement strategy of system calls DynaRace can be applied to pre-existing libraries and applications. The DynaRace module works as an independent additional module and can be used in combination with other approaches.

Filenames and directory names can be changed by a concurrent process, e.g., by using symbolic links to redirect a file to a different location. The presented approach uses two security principles: (i) A chain of trust of known directories from the root directory (/) to the directory of the file is constructed by iterating through all the different file atoms. This chain enables the use of the new system calls that are relative to a specific directory. (ii) A thread local transparent mapping of file statistics that keeps track of all accessed, opened, stat'd, and modified files.

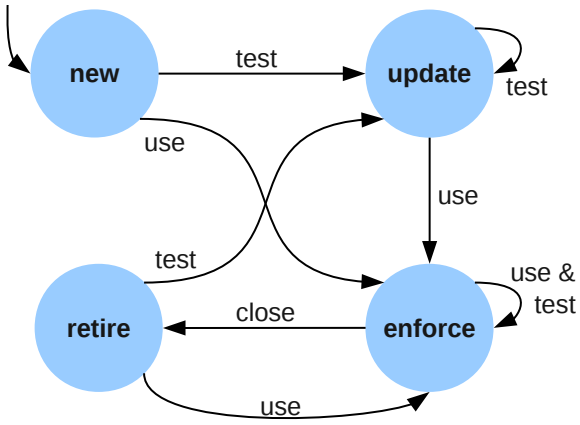

Figure 7.2: State machine with file states and dynamic checks.

7.2.1 File states

The general rewriting policy for unsafe system calls replaces an unsafe system call with a set of dynamic atomic checks. These checks ensure that if a filename is equal to an already accessed filename then the following `open` system call can rely on the subsystem to ensure that the files are also equal.

Equality is defined as *same inode number, device id, and parent directory* for existing files. For files that do not exist equality is defined as *same parent directory and an error code*. File metadata that is stored in the mapping cache must contain the complete metadata to show equality between different files or, if the file does not exist, information of the base directory and the fact that the file does not exist. The metadata of a file includes a pointer to the metadata of the file's directory.

Every file that is accessed has an associated state. Depending on the system call that is executed on the file and the state of the file, different checks are executed.

Figure 7.2 shows the different states and the checks that are executed during a transition. Nodes correspond to the states with associated checks and edges to different groups of system calls that force transitions between states. The three states are *update*, *enforce*, and *retire*. A file can be in any of these three states if the same filename has already been used in the program, or in the start state *new* if the file is used the first time.

The edges of the state machine correspond to different groups of system calls. The transition from one state to another depends on the system call and the current state. The system calls are separated into different sets depending on their function:

Test: the application uses system calls in this set to gather information and to check file metadata (e.g., permissions, or status). The system calls in this group do not change the file or any associated metadata. Examples for *test* system calls are `access` or any of the `stat` system calls.

Use: this group of system calls works with files or changes file-related metadata. The system calls in this group modify the file or the associated metadata. Examples for *use* system calls are `open`, `creat`, or `chmod`.

Close: the group of system calls that closes or deletes files: `close, unlink,` and `unlinkat`. The application uses these system calls to close or delete files. These system calls signal the kernel that the application no longer works with the specified files.

Depending on the prior state of the file and the new state (as determined by the system call) a set of checks is executed. The metadata is updated if the file has not been changed by the application (e.g., in the states new, update, and retire). If the file transitions from the update state to the enforce state (or stays in the enforce state), then DynaRace enforces the correctness of the available metadata. If the file is a new file that has not been used by the application then DynaRace adds the metadata into the metadata cache. If the file has been closed then DynaRace updates the metadata and warns if there is a mismatch in the metadata cache. This "check and warn" case is used for files that have already been used in the application but were closed.

The state machine in Figure 7.2 is updated depending on the type of the system call. The possible states of the state machine are:

Update: all state transitions that end in the update state (*new* $\rightarrow$ *update*, *retire* $\rightarrow$ *update*, and *update* $\rightarrow$ *update*) update the information of a file in the metadata cache or create a new entry for a new file in the metadata cache. This state gathers information about the different files used in the application.

Enforce: whenever a file is used (i.e., either the file itself or the metadata of the file changes) in the application then the state of the file changes to enforce. Enforce ensures that the application can always work with the same file. DynaRace enforces equality if valid information about the file is available in the metadata cache (this holds for the transitions *update* $\rightarrow$ *enforce* and *enforce* $\rightarrow$ *enforce*). If a system call of the use group is used for a new file (*new* $\rightarrow$ *enforce*) then DynaRace adds the information to the cache and changes the state of the file to enforce. If a file has been closed by the application and a system call of the use group is executed (*retire* $\rightarrow$ *enforce*) then DynaRace checks and updates the metadata and emits a warning if there is a mismatch between the cached and the current information.

Retire: a file ends in this state if it has been closed or discarded by the application (*enforce* $\rightarrow$ *retire*). This special state discards and retires files that are no longer used by the application after they have been in the enforce state.

The retire state offers the possibility to retire files that are no longer used by the application. A retired file can be changed by a concurrent process without a security

violation. This feature is important to enable ownerships transfers like log rotation. Process P_1 checks and opens a log file for writing. A second process P_2 rotates the logs (compresses the open log and creates a new, empty log file). Process P_2 signals process P_1 to reopen its log files. Process P_1 closes the log file (thereby retiring the metadata information), checks, and reopens the new file.

7.2.2 File resolution

All file-based system calls are replaced by safe handler functions that dynamically use the mapping cache to verify that identical filenames conform to the same file identified by a unique inode number and device id. File resolution is performed in a manner similar to the `chk_use` algorithm presented by Tsafrir et al. [THWDS08a, THWDS08b]. DynaRace uses the user-defined function in the `chk_use` algorithm to verify and update the mapping cache as the file is resolved atom by atom. The handler functions resolve files in the following way:

1. Split path names into (i) a directory path that contains the directory and (ii) the actual filename.
2. Rewrite relative directory path names to absolute paths according to the current working directory.
3. Identify and check the directory using the directory path.
   (a) If the directory path is already in the mapping cache then the handler function can open the directory and verify that the current directory and the data in the mapping cache are identical.
   (b) Otherwise the handler function builds a chain of trust from the root of the file system to the directory path by opening every single directory on the way and adding the information about the directory to the mapping cache, e.g., for `/var/tmp/file` the directories `/`, `var` relative to `/`, and `tmp` relative to `/var` are checked and added to the mapping cache.

   This function returns a file descriptor that can be used for system calls relative to the verified directory.
4. Next the filename is resolved using the resolved and checked directory. The file is opened and parameters are checked in the handler function (using the new relative system calls `openat` and `fstat64`). If the file is already in the mapping cache then the handler function ensures that the current file and the cached metadata are identical. Otherwise the new file information is added to the mapping cache.

If there is a mismatch between any metadata of either a directory or the file then the application is terminated with a race warning. After the file is resolved the

system call handler is executed. This handler might then add new file information to the mapping cache or change existing information, according to the system call that is executed.

7.3 Implementation

The current prototype implementation of DynaRace relies on several software layers. A binary instrumentation toolkit detects and rewrites all unsafe system calls related to file handling. These system calls are then redirected transparently to the DynaRace module. The application (and all libraries used by the application) still issue the original (unsafe) system calls. Using binary rewriting, DynaRace keeps track of all accessed directories and files.

DynaRace extends the interposition layer of the secure execution platform to check all system calls that modify files. All system calls that execute unsafe file operations are redirected to special handler functions. Each handler function handles one system call. These handler functions either (i) update the mapping cache between filenames and inode and device information for files that have not been used before, or (ii) use available information in the mapping cache to ensure that the filename still maps to the same physical file.

Related work [WP05, sLC05, GSV03] often matches specific predefined pairs of system calls to identify potential races. DynaRace introduces a new mapping cache that keeps track of all accessed files; all file-related system calls are rewritten to use this new mapping cache. This approach detects potential race conditions between any combination of file-related system calls.

The handler functions first resolve the file according to Section 7.2.2 and then check the actual system call or rewrite specific parameters. Handler functions exist for the following system calls:

Test: the following system calls are in the *test* group:

- **stat*:** all `stat` related system calls (e.g., `stat`, `stat64`) are rewritten to ensure that the specified file ends up in the cache. `fstat64` is used as the actual system call.
- **access:** this unsafe system call has no replacement that uses a file descriptor, so the handler function implements this system call using `fstat64`.

Use: the following system calls are in the *use* group:

- **open:** check flags, if `O_CREAT` is used then the handler ensures that `O_EXCL` is set as well, handling potential errors. `openat` is used as the actual system call, relative to the current directory.
- **creat:** reuses the check for the open system call.

chmod: the handler ensures that the current metadata of the modified file is equal to the cached metadata and changes the system call to use `fchmod` with the tested file.

Close: the following system call is in the *close* group:

close: the handler closes the file and reduces the number of open instances of the current file. If the counter reaches 0 then the file enters the *retire* state.

Section 7.2.1 explains the different states in more detail. The current implementation prototype emits a warning and terminates the application if an unimplemented system call is used, but the implementation can easily be extended to include other system calls as well.

7.3.1 Tracking of file states and metadata

A transparent file mapping cache keeps the information of all used files and directories. This file mapping cache enables a secure way to identify files and is used in the handler functions of unsafe system calls. These handler functions rely on the file mapping cache to identify files that have already been used in earlier system calls, e.g., a rewritten `open` system call that is preceded by a (rewritten) `access` system call uses the file mapping cache to ensure that if the filenames passed to the system calls are equal then the files themselves are also equal.

The data structure for file entries contains the following fields:

state: the current state of the file. Possible states for files are either *update*, *enforce*, *retire*, or *new* (see Figure 7.2). Directories are in one of two possible states, either accessible or in an error state.

nropen: the number of open file descriptors for this file that are in use by the application. A file can only transition to the *retire* state if all open instances of a file are closed.

fd: this field contains either 0 if the file is OK, the open file descriptor if the file is currently opened for DynaRace checks, or the error code if the file is invalid.

filename: a string that contains the file atom for files or the full path for directories.

stat: holds the result of the `fstat64` system call when the file was last accessed. This field is used to check equality of files.

dir: a pointer to a file data structure that contains information about the directory of the file. A file is always verified alongside the directory that it is in.

The file mapping cache is constructed lazily. Whenever a new file is used by the application then the file's metadata and state are added to the mapping cache. If a file or directory is reused then the current state of the file system must conform to the data in the mapping cache. The handler functions update the mapping cache for system calls that change metadata after the system call is executed but before control is returned to the translated application.

7.3.2 File resolution

File resolution is split into two steps according to Section 7.2.2. The first step constructs an authenticated chain from the directory of the file to the root of the file system to authenticate the path. The second step uses the base directory to authenticate the remaining file atom (the last component in the path) against the already authenticated base directory using the new file-based race-free system calls. The following sections discuss the implementations to resolve directories and file atoms.

Directory authentication

Path authentication starts with a full path and recursively authenticates single directories moving towards the root directory. The recursive function first checks if the current directory is already in the cache, otherwise a new cache entry is constructed. Newly constructed entries are linked to the parent directory, and the correct parent directory is also verified. Each directory is stored in the cache with the full path to enable a fast lookup.

Directory authentication is implemented using an approach similar to `chk_use` [THWDS08b]. `chk_use` consumes a path one atom at a time, starting with the root directory. The intention of `chk_use` is to specify a check function (e.g., `access`), and a use function (e.g., `open`) which are then executed after another in a race-free way. DynaRace uses the `chk_use` function to authenticate the directory by traversing the path one atom at a time and inserting the metadata information of each sub-path into the cache.

A notable difference to the original `chk_use` implementation is that DynaRace uses the new system calls like `openat` to traverse the directory path. This setup removes the need to change the current working directory of the process in the `chk_use` function and allows DynaRace to support multiple concurrent threads.

The path authentication function checks each directory once. If the check fails then an error is recorded in the list of paths. Errors that occur during the authentication of the chain propagate upwards to the initial directory (and to the caller).

Path authentication takes a full absolute path as an argument and returns either a valid open directory or an error. The opened directory can be used for further authentication of files in that directory.

File authentication

File authentication takes an authenticated directory and authenticates a file atom in that directory. This function keeps track of the state of individual files and validates correctness according to Figure 7.2. New files are initialized and added to the list of accessed files, while existing files are authenticated according to their current state and the target state.

File authentication first searches the list of already accessed files using the authenticated directory and the file name. If there is a cache hit then the file metadata is either updated or verified according to the state of the file. If the directory and the file are not in the cache (i.e., this combination is used for the first time) then a new entry is constructed with the available information.

This function throws two types of warnings. The first type warns if a file was changed by an external process during the runtime of this process. This type of warnings shows potential attacks against the program that were fixed. The second type warns if files are used without validation (e.g., opening files without checking the permissions first). This type shows protocol violations by the application. If a race attack is detected (i.e., the cached information changes in the *enforce* state) then the application is terminated.

7.3.3 Replacing system calls

The system call interposition framework checks all system calls and redirects all file-based system calls to handler functions. These handler functions implement the DynaRace core and keep state for each accessed file and all used directories.

The handler functions use directory authentication from Section 7.3.2 and file authentication from Section 7.3.2 to update the state cache of individual files and directories. These authentication functions abstract the bookkeeping problem and enable clean and simple handler functions.

The current implementation provides handler functions for the `stat`, `access`, `open`, `creat`, `chmod`, and `close` system calls. This section discusses one system call from each state: `access` for the *update* state, `open` for the *enforce* state, and `close` for the *retire* state.

The prototype implements a subset of all file-based system calls. Missing system calls can be added using the available authentication functions and the existing handler functions.

access system call

The `access` system call handler intercepts `access` system calls and rewrites them dynamically to use `fstat64`. The handler function splits the given filename into an absolute pathname (a relative path is resolved using `getcwd`) and a file atom.

The absolute path is authenticated according to Section 7.3.2. All directories on the path are consequently added to the directory cache. Directory authentication returns an open file descriptor for the directory of the used file. This file descriptor is then used to execute a `fstat64` system call. The stat information is needed to construct the state information of the file atom. The file metadata entry is then constructed depending on the return values of directory authentication and the following `stat` system call using the file authentication function.

The `stat64` struct contains all information needed to reimplement the `access` system call. A macro uses the stat information, the user id, and the group id to return the same information as the access system call. The return value of the `access` handler function is either an error value for invalid files or 0. The state update can trigger warning messages that are logged.

open system call

The `open` system call handler intercepts all `open` system calls and rewrites them to use the safe alternatives. The given pathname is split into a file atom and an absolute path similar to the `access` system call handler. The absolute path to the file atom is then authenticated using directory authentication.

The file atom is then opened using the `openat` system call relative to the authenticated base directory. If the application uses the truncate attribute to remove the file's contents then the attribute is removed from the executed `open` system call. The truncating is delayed until after the authentication and the update of the metadata. The handler function then updates file metadata using the file authentication function.

The handler function either returns an error value or the open file handle. If there are authentication errors then the program terminates. If the application opens unchecked files that are in the "new" state then DynaRace emits a warning (in the file authentication function).

close system call

Similar to the `access` and the `open` system calls, the `close` system call is redirected to a handler function. The `close` system call has the advantage that the file is already open and there are no potential race conditions when accessing this open file descriptor.

The handler function searches the cache using the unique inode number and device id. If the file is opened multiple times then the number of open files is reduced. If the last file descriptor of a specific file is closed then the state of the file is updated to the special *retire* state.

New system calls

DynaRace can coexist with applications that already use new system calls that use relative directory file descriptors (`accessat`, `openat`, `fstatat64`, etc.). The new system calls are handled just like regular file-based system calls and redirected to a handler function by the interposition framework. The handler function then updates the file metadata and executes the new system call using the information provided by the application.

7.4 Implementation alternatives

DynaRace uses and extends libdetox, the prototype implementation of the secure execution framework. Three other implementation approaches are possible. The first approach extends the Linux kernel and implements the DynaRace approach, e.g., as a kernel module. The second approach extends the standard libc and implements the DynaRace approach on top of the library. The third approach uses the `ptrace` debugging framework to implement DynaRace in a concurrent process.

7.4.1 Kernel DynaRace implementation

An alternative implementation could extend the kernel with a module that implements the DynaRace approach and keeps a cache of accessed files on a per-application basis. This state cache could then be used whenever the application requests an unsafe system call.

An advantage of this implementation approach is that there is no overhead for binary translation and that the kernel can also keep track of all accessed files for all running applications.

A disadvantage is that a kernel-based implementation needs kernel level access. The kernel-based implementation can contain bugs and lead to serious exploits. Another point is that the Linux kernel already provides file-based system calls relative to open directories. These system calls can be used to implement safe applications. DynaRace is only needed for potentially unsafe applications. New functionality should only be added to the kernel if a user-space implementation is not feasible.

We argue that the risk of potential exploits in kernel space due to additional code is not worth the advantage of the potentially lower overhead. In addition, a user-space implementation is preferable whenever possible.

7.4.2 libc-based DynaRace implementation

A second alternative implementation could extend the standard libc library. This library contains wrappers for most of the file-based system calls. These wrappers are

then used by the application. The implementation would extend the wrappers for all file-based system calls to implement the DynaRace approach. A static cache of accessed files and states would be initialized during the standard libc initialization and used whenever a file is accessed.

An advantage is the lower overhead compared to our prototype implementation because no binary translation is needed to redirect and catch the system calls.

A disadvantage is that the application can still execute native system calls by using an unpatched ("third-party") library. This third-party library breaks the security of the race detection. A second potential problem is that the DynaRace race checks are executed at the same privilege level as the application. Our implementation uses binary translation for the application; the file-based system calls are then intercepted from the sandbox and redirected to the DynaRace implementation in the privileged part of the sandbox in user-space.

We argue that the risk of a third-party library that executes a direct system call is too high compared to the performance overhead for binary translation. A possible implementation should offer "complete" protection for all system calls, and not only the system calls that are executed through the libc.

7.4.3 ptrace-based DynaRace implementation

A third alternative implementation leverages the `ptrace`[2] system call to implement the DynaRace approach. A separate DynaRace process controls the target process and observes all system calls externally. The DynaRace process intercepts and inspects all file-related system calls.

DynaRace replaces the unsafe system call with a piece of code that executes a set of safe system calls and some checks. The DynaRace process would inject that code into the running process and redirect the execution flow to the injected code. This injected code would then replace the original system call and return the result from the kernel to the unmodified application code.

An advantage of this approach is that the application does not need to be translated and virtualized. Depending on the implementation of the DynaRace process this setup could lead to some performance speed-up compared to our prototype implementation.

A disadvantage is that the DynaRace process must either stop the traced process upon every system call, a setup that leads to high context-switching overhead, or DynaRace must inject new code into the application that could lead to unwanted side-effects. The implementation of the DynaRace process and the code-injection technique is critical to keep the overhead low. Original system calls must be permanently redirected to injected code to reduce the task switching overhead between the

[2]The `ptrace` system call is used to remotely control a process. The tracing process can read and write memory locations and registers of the target process, controls signals and signal delivery, and can inspect system calls and parameters.

DynaRace process and the application process. Another disadvantage is that the application process does not profit from the additional protection that the sandbox offers.

We argue that a feasible `ptrace`-based implementation will be too complex due to the need to inject code in the application domain. A virtualization-based implementation adds new code naturally and instruments the executed system calls using simple redirection. Binary rewriting tools provide additional opportunities for error checking that are hard to replicate with external debugging techniques like `ptrace`.

7.5 Evaluation

The DynaRace prototype is implemented on top of the secure execution framework. This section evaluates (i) the raw performance of the DynaRace prototype implementation using several microbenchmarks, (ii) end-to-end performance of the DynaRace prototype implementation using a complex Apache setup, and (iii) evaluates DynaRace using several application scenarios. The discussion of the application scenarios shows how DynaRace protects from file-based race attacks in these scenarios.

7.5.1 Performance evaluation

The set of features and the security guarantees raise the question about the total overhead for libdetox. Optimizations in the dynamic translation process result in an average overhead of 15.4% for the SPEC CPU2006 benchmarks.

The prototype implementation of DynaRace is not yet optimized and there is potential to reduce the number of executed system calls and to use better data-structures for the mapping cache. Table 7.3 shows overheads for specific microbenchmarks depicted in Listing 7.1. Every microbenchmark executes the code sequence 1,000,000 times in a loop. The microbenchmarks are executed on an Intel Core i7 950 CPU at 3.07GHz using a 64-bit version of Ubuntu 10.10. The benchmarks are compiled using gcc version 4.5.2-8ubuntu4.

An interesting result of Table 7.3 is that libdetox can outperform the native execution for raw system call throughput in the open/close test. Two reasons for this behavior are (i) trace linearization inside the code cache (of the binary translator)

	Native	**libdetox**	**DynaRace**
1) long test	4.73ms	4.00ms	20.96ms (4.4x)
2) access test	1.59ms	1.62ms	6.11ms (3.8x)
3) open/close test	1.93ms	1.00ms	6.90ms (3.6x)

Table 7.3: Results of the microbenchmarks compared to native performance.

```
/* 1) long test */
if (access("input", R_OK|W_OK)==0) {
  int fw = creat("test", 660);
  if (fw == -1) perror("Creat");
    int fr = open("input", 0);
    if (fr == -1) perror("Open");
    close(fw);
    close(fr);
} else {
  printf("Unable to open dir");
}

/* 2) access test */
int j = access("input", R_OK|W_OK);

/* 3) open/close test */
int j = open("input");
close(j);
```

Listing 7.1: Microbenchmarks used for the evaluation (C code)

for the translated standard libc function, and (ii) the inlining of all system calls from the `ld-linux.so` library.

The DynaRace prototype implementation uses multiple system calls to verify that already accessed files have not changed between system calls. For new files additional system calls are used to gather information for the mapping cache. This setup leads to an overhead for file-based system calls of around 3-4x (for raw system call performance) to remove potential race conditions. Only the small set of file-based system calls that check or modify file metadata incur overhead. The overhead can be reduced through future optimizations that reuse more information or keep frequently accessed files open. All other (non file-related) system calls do not incur overhead.

The overhead is tolerable as system calls used to check the metadata of files and to open/close files are rare compared to read or write operations or computation. There is no overhead in the access to a file's data (e.g., reading from a file or writing to a file). Any overhead caused by DynaRace is associated with file metadata management. For most programs system calls that modify file metadata are not on the hot path. The time spent for I/O dominates the time spent for metadata management.

7.5.2 Apache web server study

This section presents an end-to-end performance evaluation using the Apache web server. The web server study compares the performance of a system with active DynaRace protection for the Apache server and all scripts to a system without DynaRace. This study shows the completeness and end-to-end performance of our prototype implementation. The study uses Apache 2.2 on 32-bit Ubuntu 10.04 LTS on a Core i7 950 CPU at 3.07 GHz in a VirtualBox virtual appliance using a single core. Apache uses the default Ubuntu configuration (multiple processes multiple threads, full support for the dynamic PHP interpreter and other modules). The study uses the `ab`[3] Apache benchmark in a different virtual machine on a different core on the same CPU to download four different files from a server. Each file is downloaded 100'000 times and Apache is restarted between iterations.

The four different files are `index.html`, a 5kB HTML file; `image.png`, a large 1mB image; `test.php`, a short PHP script that generates 90B output; and `test2.php`, a PHP script that executes `phpinfo()` and generates 48kB of output.

This evaluation does not use the same files as the Apache evaluation in Section 6.6, therefore we evaluate libdetox for this configuration as well. The evaluation uses three different system configurations: (i) native, Apache runs unmodified and unprotected; (ii) libdetox, Apache runs in a protected environment under the control of libdetox; (iii) DynaRace, Apache runs in a protected libdetox environment with a DynaRace module that protects all file accesses as well. All three configurations run inside a virtual machine.

Table 7.4 shows the performance numbers of the Apache benchmark. The three different system configurations are evaluated for each file. Each column shows the number of completed requests per second, the libdetox and DynaRace columns also show the overhead compared to native performance. The relative overhead shows that libdetox and DynaRace exhibit a small performance improvement compared to the native performance for static files. The improvement comes from the binary translation that results in greedy trace extraction for hot code.

[3]The command used for the measurements is: `ab -c 2 -n 100000 http://${VM}/${FILE}`.

	Native	**libdetox**	(ovhd.)	**DynaRace**	(ovhd.)
index.html	1464	1675	-14.5%	1601	-9.4%
image.png	48	51	-6.3%	47	1.6%
test.php	1773	1562	12%	1498	15.5%
test2.php	463	343	26%	320	30.9%

Table 7.4: Results for the Apache benchmark showing the number of requests per second. The benchmark uses four different files to compare native performance, libdetox performance, and DynaRace performance.

For the dynamic files (`test.php` and `test2.php`) libdetox results in 12% and 26% overhead due to the translation of the dynamic PHP interpreter (which results in many indirect control flow transfers). Comparing the libdetox and DynaRace columns shows that DynaRace results in roughly 6% more overhead than libdetox alone. This overhead comes from the additional system calls for the directory and file atom authentication.

The overhead of both libdetox and DynaRace is tolerable, especially when considering that only a prototype implementation is evaluated. The prototype implementation still leaves room for additional performance optimization (e.g., additional caching of open file descriptors, better code optimization, and other optimizations). The current prototype implementation shows that the DynaRace approach is feasible and the overhead is tolerable. Additional optimization is left as a topic for future work.

7.5.3 Protection from file-based race conditions

DynaRace relies on the handler functions for the different system calls and the state of each file to protect and to remove potential race conditions. The file state is updated using transitions from one state to another.

This section shows three important usage scenarios that are common in applications. The usage scenarios are (i) checked file access where access permissions for a file are checked before it is used, (ii) temporary file creation where a file is created in an unsafe directory, and (iii) log rotation where files are replaced by an external process. Each usage scenario is dissected into the individual system calls and what (implicit) assumptions are used between the system calls. The described usage scenarios are prone to race conditions in their original form. DynaRace adds additional state to each file and enforces the assumptions between the system calls. We implemented demo exploits for all three scenarios with sufficient wait time between system calls so that an attacker can exploit the race condition. DynaRace protects all of our sample programs effectively.

State transitions between system calls are shown in the following way:

$$oldstate \xrightarrow[targetstate]{systemcall} newstate$$

The state associated with a file transitions from *oldstate* to *newstate* if a *systemcall* system call is executed that induces the *targetstate* state. Both *newstate* and *targetstate* are needed for files that are in the *enforce* state. If a system call would induce the *update* state (e.g., the `access` system call) and the file is currently in the *enforce* state then the file remains in the *enforce* state.

Checked file access

The checked file access pattern consists of four steps to work with a file. The first step checks access permissions of the existing file using the `access` (or `stat`) system call. The second step opens the same file using the `open` system call, returning a file descriptor. The third step uses the file (e.g., reading from, or writing to the file descriptor using the `read` and `write` system calls; `read` and `write` system calls do not change the state of the file). The fourth step closes the file again using the `close` system call, completing the usage pattern.

A potential attack races to exchange the checked file after the first validation step and before the second opening step. Using mazes of connected directories [BJSW05] an attacker can win this race every time if he or she can inject a symbolic link into any part of the file name (see Section 7.6.4).

With DynaRace the following four state transition sequences are possible:

The file has not been accessed/used before by the application:

$$new \xrightarrow[update]{access} update \xrightarrow[enforce]{open} enforce \xrightarrow[retire]{close} retire \tag{7.1}$$

The file has been used before but was closed by the application:

$$retire \xrightarrow[update]{access} update \xrightarrow[enforce]{open} enforce \xrightarrow[retire]{close} retire \tag{7.2}$$

The file has been accessed by the application:

$$update \xrightarrow[update]{access} update \xrightarrow[enforce]{open} enforce \xrightarrow[retire]{close} retire \tag{7.3}$$

The file is still opened by the application:

$$enforce \xrightarrow[update]{access} enforce \xrightarrow[enforce]{open} enforce \xrightarrow[retire]{close} enforce \tag{7.4}$$

The above state transition sequences leave no opportunity for a potential attacker to change files between a check system call and a "use" system call. An attacker could swap files in the *update* or *retire* state but not if the current state or next state is *enforced*.

A potential attacker tries to change a file after the `access` system call and before the `open` system call. Using the DynaRace states an attacker can change the file only before the `access` system call in Equation 7.1; in Equation 7.2 and Equation 7.3 a warning is printed, and in Equation 7.4 the application is terminated with a race condition error message. DynaRace guarantees that in all cases no unchecked file can be injected between the `access` and the `open` system call. A potential attacker can change the files before the permissions are checked but never between the permission check and the system call that uses the checked file.

As soon as a file transitions into the *enforce* state, the last checked or accessed information is fixed and verified. This setup results in a guarantee to the program that the last checked information of a file is enforced when that information is used later.

Temporary file creation

Safe temporary file creation is an important and hard problem. An access test using an `access` or `stat` system call first checks that the file does not exist. A following `creat` system call (or an `open` system call with the create exclusive flag) then creates the file. An attacker can try to race between the existence check and the file creation to add a link to an already existing file that will then be overwritten.

$$new \xrightarrow[update]{access} update \xrightarrow[enforce]{open} enforce \tag{7.5}$$

Equation 7.5 shows the state transitions when a new file is created. The state information contains a valid directory and the error code for the file after the `access` or `stat` system call. The directory where the file will be placed in must exist but the temporary file must not exist.

Log rotation

Log rotation is a technique that is used by many daemons to renew/rotate their log files. A daemon opens a log file and writes information to that file. A rotation daemon moves these log files to a different place and signals the original daemon to reopen the log file. The daemon then closes the old file and reopens the log file (which no longer points to the same inode and device id).

DynaRace's *retire* state enables log rotation. When a file is closed it can be reopened even if the metadata of the file has changed. The *retire* state is similar to the *new* state where we do not know anything about the file. The information in the cache is updated and enforced only if there exists a valid set in the cache.

$$enforce \xrightarrow[retire]{close} retire \xrightarrow[update]{access} update \xrightarrow[enforce]{open} enforce \tag{7.6}$$

Equation 7.6 shows a sequence of system calls that is used during the rotation of a log file. The file is closed by the daemon, checked, and reopened. The `access` system call following the `close` system call prints a warning message that the file has changed. The daemon continues in the *update* state and enforces the information from the check when the file is opened.

This scheme leaves a window of opportunity for the attacker where he or she can change the log file after it was closed but before it is accessed. The access check in the daemon catches these attacks and the state information is later enforced.

```
...
lfd = open(tmp, O_CREAT|O_EXCL|O_WRONLY,
           0644);
...
if (lfd < 0) {
  unlink(tmp);
}
...
write(lfd, pid_str, 11);
/* unchecked relaxation */
chmod(tmp, 0444);
...
```

Listing 7.2: TOCTTOU vulnerability in X.org (os/utils.c, C code)

7.5.4 X.org file permission change vulnerability

Version 1.4 to 1.11.2 of the X.Org X11 X server had a severe TOCTTOU race condition [vla11]. The X server creates a temporary lock file and relaxes the permissions of the lock file using unsafe system calls. Any local attacker with permission to run the X server can exploit this vulnerability to set the read permission for any file or directory on the system.

Listing 7.2 shows the code containing the race condition. The variable `tmp` contains a fixed string of `/tmp/.tXn-lock` where X is the n-th X display running on that computer. An attacker executes the SUID X binary as P_1. The attacker stops P_1 after the `write` system call and before the `chmod` system call. In a second execution P_2 of the X binary the `open` system call fails and the temporary file created by P_1 is removed using the `unlink` system call. The attacker kills P_2 and links `/tmp/.tXn-lock` to an arbitrary file (e.g., `/etc/shadow`) and continues P_1. P_1 will then set the arbitrary file to world-readable.

DynaRace protects from this TOCTTOU race condition. As soon as the file identified by the `tmp` variable is in the *enforce* state it can no longer be modified by a concurrent process. Equation 7.7 shows the states for the temporary file for P_1. Before the `chmod` system call P_1 is stopped and P_2 depicted in Equation 7.8 is executed. P_1 then continues and DynaRace throws an error because the metadata of the enforced temporary file has changed.

$$P_1 : new \xrightarrow[enforce]{open} enforce \xrightarrow[enforce]{chmod} FAIL \tag{7.7}$$

$$P_2 : new \xrightarrow[enforce]{open} enforce \xrightarrow[unlink]{retire} retire \tag{7.8}$$

7.6 Related work to file-based race detection

DynaRace is a dynamic TOCTTOU race detection mechanism that is implemented as a libdetox extension. File-based race detection can be built on different techniques. Static race detection analyzes the program source code or the binary of the application before execution; dynamic race detection uses code that is added during the compilation of the program to detect races at runtime; dynamic race prevention uses added code to prevent possible races; and novel race-free APIs change the available file system API to remove or reduce potential races.

7.6.1 Static race detection

Static source code analysis [VBKM00, Che02, CW02, SCW+05] can be used to detect potential file-based race conditions. A scanner reads the source code of the application and observes calls to library functions or system calls. If a sequence of calls opens a race condition then the static analysis tool emits a warning.

Static analysis tools often have a high number of false positives because they rely on pattern matching and they cannot use runtime data to verify potential races. Due to the high error rate these approaches are not feasible for online services that must meet some response time constraints.

The DynaRace approach adds state to each accessed file. Due to the classification of system calls into groups and the additional state, DynaRace can determine for each system call if a potential race condition occurs.

7.6.2 Dynamic race detection

A dynamic approach can observe all system calls as they happen. These system calls can either be logged and analyzed postmortem [KR02] or an online analysis can evaluate the pair of system calls according to a given policy [sLC05]. Aggarwal and Jalote [AJ06] use static binary translation to add dynamic guards that are executed at runtime.

IntroVirt [JKDC05] uses OS virtualization and a virtual machine monitor to run predicates outside of the OS scope. These predicates check the health of system software and also check for file race conditions if the programs have the right predicates. Wei and Pu [WP05] enumerate likely TOCTTOU pairs that are used for common tasks. Goyal [GSV03] uses the `strace` framework to collect system call traces and detect TOCTTOU attacks.

The technique of enumerating pairs of system calls misses the opportunity to detect race conditions dynamically between any combination of system calls. This approach is similar to blacklisting several pairs of system calls without looking at all possible combinations. Only an approach like DynaRace that takes care of all file-based system calls can protect from all race conditions.

7.6.3 Dynamic race prevention

RaceGuard [CBWKh01] defends against several classes of TOCTTOU attacks using an in-kernel cache but is limited for file swap attacks where the file itself is changed from one execution to the other. Tsyrklevich and Yee [TY03] implement pseudo-transactions in a kernel module to detect specific combinations of check/use system calls targeting the same file. Multiple similar approaches followed [PLLK04, UJR05, WP06, SGC+09].

Several novel race-free APIs have been proposed to replace the POSIX-like system calls for file handling. Some of these novel APIs use transactions to run a set of related system calls atomically [SW91, WSSZ07]. The OS ensures that no other application may interfere with these system calls.

Dean and Hu prove that no deterministic solution to prevent races exists (with the set of available system calls in 2004) and introduce a probabilistic way to prevent races [DH04]. They assume that an attack is unable to win all races. Therefore they propose hardness amplification. Hardness amplification executes the check multiple times in a loop (a so called K-loop) and checks if all results are as expected.

Transactions of system calls need kernel modification. In addition to the new code in the kernel the applications must be rewritten to include transactional code. DynaRace supports a gradual approach that adds protection from file-based TOCTTOU races. Applications and libraries can use a mix of old file-based system calls and new file-based system calls. Using the file-state and the new system calls DynaRace deterministically protects every file-based system call.

7.6.4 Maze-based race attacks

Borisov et al. break the probabilistic approach by enabling an attacker to win all races [BJSW05]. If all races are won by the attacker then the check proposed by Dean and Hu [DH04] is unable to detect the race condition. An attacker constructs so called mazes. A *maze* is a chain of temporary directories that contains a link to the next chain as the leaf. The attacked program needs time to go through the maze (e.g; once for `access` and once for `open`) and in that time the attacker swaps the final link.

7.6.5 Hardness amplification-based race protection

Mazières and Kaashoek [MK97] propose an alternate approach that uses inodes (low level file objects) instead of filenames. If the binding between inode and filename is immutable then no file swap race conditions are possible when the target file changes. Tsafrir et al. [THWDS08a, THWDS08b] implement a similar approach in user-space. Their approaches render the maze attack from Borisov et al. [BJSW05] impossible. User-space checks guard applications from TOCTTOU attacks. A file

is resolved step by step. The algorithm checks for each part of the filename if the current part is a file or a symlink. Files are opened using an approach similar to Dean and Hu [DH04] using a K-loop probabilistic check. Symlinks are opened using a recursive check using the same algorithm.

Chari et al. extend the work of Tsafrir er al. to implement a `safe-open` mechanism [CHV10] that ensures that all elements of a path are safe to open by the current user id. Safe file elements can be modified only by the same user or root.

DynaRace introduces a hidden mapping and state cache and rewrites unsafe system calls. This technique gives similar guarantees as Mazières and Kaashoek without the need to change the implementation of the application or the libraries. DynaRace resolves a file step by step similar to Tsafrir et al. using safe system calls.

7.7 Limitations and weaknesses

This section discusses limitations and weaknesses of the DynaRace approach and highlights insights obtained through the DynaRace prototype implementation. The idea of DynaRace is to associate state to each file. Files are identified using a mapping cache that maps filenames to unique inodes and device numbers.

Using the state machine in Figure 7.2 enables specific checks for file-based system calls, whereas the state transitions are decided based on the group that the current system call is in and the state of the file. All usage cases of a file inside an application are covered using the state diagram that never throws an exception if the file has not been changed between different system calls.

The *retire* state enables multiple concurrent processes to modify files in a safe way. Closing a file in one application retires the information in the mapping cache and the application no longer assumes that the state of the file remains the same. Ownership transfers, e.g., log rotation, are only possible using the *retire* state.

7.7.1 Retirement schemes

The current implementation keeps all files in the *enforce* state after they have been used with a system call of the *use* group. A possible extension of the state machine in Figure 7.2 would be a retirement scheme for system calls that change the file but do not return an open file descriptor (e.g., `chmod`, `chown`, or more general system calls in the *use* group without `creat` and `open`). The application does not signal the kernel that it no longer expects to use the associated file (e.g., through the `close` system call for opened files). DynaRace therefore cannot check if a file is no longer used for these system calls.

The retirement scheme for these system calls could be implemented through a timer that starts when the system call is executed. The timer would signal when these files can be retired and the state of the file would move from *enforce* back to

update, thereby enabling concurrent modification without terminating the program. On the other hand a timer would open a window of opportunity for an attacker to delay the application long enough. We leave this problem to future work.

7.7.2 Broken (legal) usage scenarios

DynaRace protects from changes to file-metadata from concurrent processes if the file state is enforced (i.e., if the application assumes that information from prior system calls is still valid). Possible broken usage scenarios are concurrent file/directory modification and polling.

If an application works with a file in the *enforce* state and a concurrent process changes the parent directory (e.g., by moving the directory, or by renaming the directory) then the directory verification will fail and an error is thrown. Two concurrent processes also cannot work together on a single file if it is in the enforce state in both processes.

Polling (i.e., checking metadata of the file) from concurrent processes works as long as the file is in the *update* state. If both processes want to work with the file (e.g., reading, writing, changing metadata), then DynaRace will terminate the second application due to a metadata mismatch.

DynaRace limits the number of modifiers of each file to a single process. If multiple processes try to modify a file at the same time then all processes except the first are terminated due to metadata mismatches.

7.8 Summary

This case study presents DynaRace, a novel approach to detect and prevent file system races in unmodified applications. DynaRace adds state to each file and keeps metadata for all accessed files in a mapping cache.

File system races are detected if the information in the mapping cache does not match the information of the current file. Our prototype implementation emits a warning and terminates the application if races are detected.

DynaRace ensures that there are no file-based races possible and dynamically rewrites unsafe system calls into race-free versions. A benefit of DynaRace is that it protects all file-based system calls. DynaRace allows partial migration of individual libraries to the new set of file-related system calls. The new system calls are no panacea: even with new system calls a programmer still has to worry about the state of individual files. DynaRace removes this burden from the programmer and offers a state-based dynamic approach for all file-related system calls.

Currently programmers have to live with potential races when they use standard system calls. DynaRace enables programmers to use both the unsafe and the new system calls without the need to add explicit safety checks in the application.

8

Future directions

This chapter discusses possible extensions for the secure execution platform. The discussions in this chapter are of purely qualitative nature and should be seen as suggestions: extensions presented here are not implemented and are therefore not empirically evaluated.

The first extension discussed in Section 8.1 enables a more fine-grained control over control flow transfers. The second proposed extension in Section 8.2 facilitates the generation of system call policies by analyzing system call parameters. The third extension discussed in Section 8.3 analyzes all network-based I/O from and to the application and checks for specific patterns. Section 8.4 discusses a dynamic patching mechanism that protects running applications from potential future exploits.

8.1 Compiler-based CFG generation

An execution model with all valid targets for a single location can be generated in multiple ways. The original Control Flow Integrity (CFI) [ABEL05] approach uses static analysis to enumerate all possible targets, while our approach in Section 4.4 uses a dynamic approach that takes advantage of already available information in the Executable and Linkable Format (ELF) headers of shared objects. Some compiler extensions [ACR+08, CCM+09, MCZ+11] described in Section 3.6 protect the execution of the application by additional runtime checks.

A more fine-grained alternative would be the automatic generation of a control flow transfer model by the compiler during the compilation of a library or an application. The compiler and to some extent the linker have full information about all possible control flow transfers that an application can execute (including domain specific knowledge about the exact program behavior). The compiler could compile this exact information to a detailed map that is emitted alongside the regular compiler output (e.g., object files or assembly code). This generated map could then be integrated into the shared object (or application) as an additional section in the ELF header[1].

[1]The GNU specific debugging information or GCC specific line number tables are added in a similar way.

The secure execution framework would then be able to load this exact information during the startup phase of the application. This would lead to a more detailed model for control flow transfers and would provide the same precision as CFI with the advantages of a dynamic protection system.

8.2 A compiler-driven approach to system call policies

During the compilation of a program detailed information is available about possible parameters for function calls. A system call can be seen as a function call with extended privileges and a specific calling convention.

The compiler can determine which parameters of a system call are static and which ones are dynamic (e.g., depending on user-input or depending on some other program state). A compiler is therefore in the unique position to generate an exact system call policy with fine-grained parameters for a compiled program.

The proposed extension would extend a compiler and add additional checks for system calls. Using the collected information about the different system call parameters combined with constant propagation and other compiler optimizations leads to exact knowledge about the individual system call parameters (e.g., ranges for parameters, or exact static values). Based on the parameter information a set of rules would then be generated minimizing the number of rules and the verbosity of the parameters.

An optional manual pass by, e.g., the administrator or the application developer could use the domain specific knowledge about the context of the application that cannot be determined automatically to further limit the parameters that are left in an open state.

This approach would make it easier to generate fine-grained and exact policies with little or no human interaction required. The generated policies can then be used as a drop-in replacement for the hand-coded policies in the secure execution platform.

8.3 I/O purification extension

The I/O purification extension is an orthogonal extension to the DynaRace approach presented in Chapter 7 where a set of system calls is redirected to additional checker functions in user-space to remove a specific threat.

Applications use network I/O to do useful work, to read input-data and to return a result to the requesting process (or user). Most exploits use an I/O channel to transfer malicious data into the application which then triggers some exploit. On the other hand, Internet protocols define possible legal messages.

The I/O purification extension implements a transparent input parser in front of the application code processing the input request. This input parser validates the request before it is passed to the application. Such an input validation has the advantage that the protocol can be defined in a compact way and the only result the parser must generate is either accept or deny. If a request is accepted then it is forwarded to the application, otherwise the request is logged and dropped.

This approach is similar to network-based intrusion detection systems that analyze the data flow between hosts. The local approach has the advantage that a potential attack can be circumvented as well and the application is protected. A network-based intrusion detection system can only detect the attack but cannot protect the application from the attack without disrupting the network connection.

8.4 Dynamic patching

Dynamic patching enables a patch to be applied on-the-fly to a running application. The secure execution framework protects running applications from any code-based attacks. Attacks are detected and the application is terminated. An attacker still has the opportunity to start a denial of service attack against the application: the attacker is unable to gain any privileges on a system but an exploit can still be used to bring a service down. The secure execution framework cannot catch these attacks dynamically. An approach similar to Section 8.3 would help to block known bad input data before they are passed to the application, but the exploit opportunity still remains in the running application.

Dynamic online patching extends the secure execution platform by a service that dynamically patches the application. Small security patches (that do not change the data-layout of the application) can be applied dynamically to a running application to remove the opportunity for a denial of service attack. Simple security patches that do not change any data structures but only add or replace individual instructions can be replaced easily without the need to restart the application. The online patching mechanism is related to Aspect-Oriented Programming (AOP) but the dynamic execution platform enables a patching mechanism that does not rely on a specific compiler and a special AOP runtime. Translated code in the code cache can be dynamically replaced using a coordinated flush of the code cache in all running threads and a list of patched addresses that are replaced during the (new) translation. Larger patches that replace individual functions can be patched by adding additional data structures in the secure loader. The functions can swap the complete code but the data structures must remain the same due to already existing data structures that are in use by the application.

The combination of the secure loader and the dynamic execution platform enables small instruction patches and function patches to replace exploitable parts of the application at runtime.

9

Concluding remarks

This thesis presents a novel secure model for the execution of untrusted application code. Today large programs run as daemons or desktop applications. Each program has full access to all the capabilities of the user that runs the program. These programs can be attacked through different possible intrusion vectors to gain control of a running process. The secure execution model uses a small framework to protect the running program using a secure loader and a runtime sandbox. Programs that execute under the control of the secure execution model are protected from all code-oriented and data-oriented attacks, and the privileges of the program are limited to a well-defined policy.

The list of Common Vulnerabilities and Exposures (CVE) shows that many exploits are code-oriented. Larger software projects are still written in low-level languages mainly out of speed reasons and to keep up compatibility with older projects. Even high level languages like Java usually run in a virtual machine that is written in a low level language. In addition high level languages often provide access to native methods. A small trusted computing base that executes large programs in a safe environment removes the threat of code-oriented exploits and still retains speed and compatibility to legacy software.

9.1 Summary and contributions

As noted in the thesis statement in Section 1.4 this thesis shows that it is possible to build a secure dynamic execution platform that protects from code-oriented and data-oriented exploits. The platform combines dynamic control flow integrity, a secure loader mechanism, and a sandbox to run untrusted code. The platform is expandable with modules that guarantee additional security properties.

The contributions are as follows:

1. Chapter 4 gives a detailed explanation of the secure execution platform combining a secure loader, a dynamic sandbox for the execution of untrusted code, and a system call policy guard that checks all system calls executed in the untrusted code. The secure loader is used to form a chain of trust during the bootstrapping of the application and to provide information for the dynamic

control flow checks that restrict the control flow transfers of the translated code in the sandbox.

2. Chapter 5 presents the prototype implementation of the secure execution platform (which is released as open source). The prototype implementation is then evaluated using different benchmarks in Chapter 6. The evaluation shows that the performance overhead for the complete execution platform is low (around 15% of overhead on average for the SPEC CPU benchmarks if the complete feature set of libdetox is used). The security discussion also shows that code-oriented exploits are no longer possible and will be detected by the model.

3. Chapter 7 demonstrates the expandability of the secure execution platform by adding additional guards that detect file-based race conditions.

9.2 Expandability

The secure execution platform is designed to add additional security guards and modules. The combination of secure loader and sandbox protects from code-oriented exploits and the system call policy protects from privilege escalation using data-based exploits. Data-based exploits can still use the available system calls. New modules can be added in both the sandbox and the system call authorization layer.

Information from the loader component is already used to remove the additional indirection for Procedure Linkage Table (PLT) calls during the translation of code in the sandbox. The sandbox can be extended with additional checks for different instructions to, e.g., search for specific code sequences, and to rewrite unsafe instructions.

The system call interposition layer, on the other hand, can be extended to include additional control logic. A specific set of system calls can be redirected to a privileged module that checks these system calls for malicious parameters, potential race conditions, or bad input data. Chapter 7 demonstrates the expandability by adding checks for file-based race conditions.

A

x86 ISA

The x86 Instruction Set Architecture (ISA) is defined by Intel [Cor12a, Cor12b, Cor12c] and describes the instruction set of all x86 compatible processors. Intel uses a Complex Instruction Set Computer (CISC) design for the x86 ISA to enable multiple low-level operations (e.g., a sequence of a memory load, an arithmetic operation, and a memory store) in one instruction. The x86 ISA supports 16-bit, 32-bit, and 64-bit addressing modes and operand modes. Figure A.1 shows the encoding of 32-bit x86 instructions.

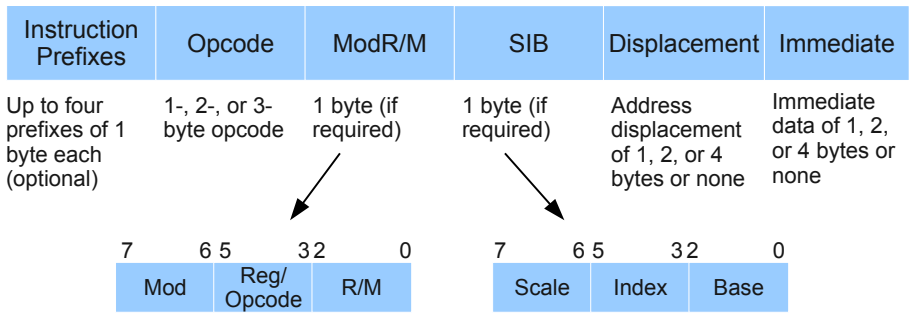

Figure A.1: The x86 instruction format according to the Intel manual [Cor12a].

Instructions are parsed from left to right. The x86 architecture uses a variable instruction length encoding of 1 to 3 bytes for the opcode to specify a large set of instructions. 2 and 3 byte opcodes use escape bytes as the first or as the first 2 bytes, enabling an easier lookup in multiple lookup tables. Prefixes can be used to override the address size, to override the operand size, to repeat certain instructions, to override segments, to lock memory transfers, and to give branch hints.

The ModR/M byte and SIB byte encode information about registers or memory addresses. Depending on the instruction and the encoding of the ModR/M byte and the SIB byte a displacement and immediate value is encoded as well. This results in an instruction length from 1 to 17 bytes for 32-bit mode.

The x86 ISA uses multiple privilege rings. Ring 0 is the most privileged ring and runs the kernel. Ring 3 is the least privileged mode and usually runs the application code. Privileged instructions raise an exception if executed in an unprivileged ring.

B

ELF format and the Linux loader

Modern Unix operating systems and the Linux kernel use the Executable and Linkable Format [SCO96, Dre10] (Executable and Linkable Format (ELF)) to specify the on-disk layout of applications, libraries, and compiled objects. A file that uses the ELF format to define its internal layout is called a Dynamic Shared Object (DSO). The ELF format defines two views for each DSO. The first view is the program header that contains information about segments; the program header controls how the segments must be mapped from disk into the process image. The second view is the section header table; this table contains detailed section definitions. Figure B.1 shows the data layout and the two different views.

The Linux kernel reads the ELF information and the program header to map parts of the binary file into memory. The kernel then passes control to the dynamic loader that is specified by the `interpreter` parameter in the program header. The

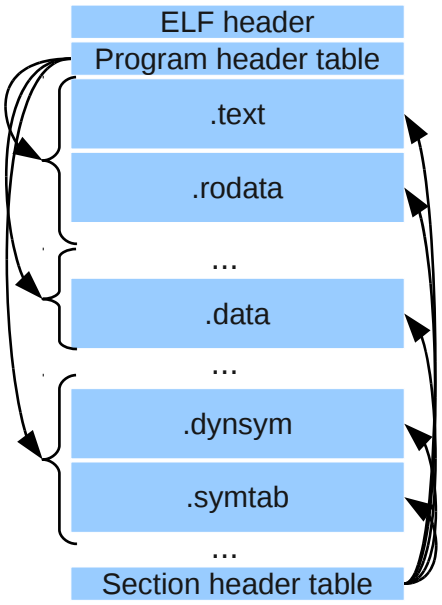

Figure B.1: Overview of an ELF DSO. The program header table follows the ELF header. The section header table is at the end of the file.

```
dlerror:     calloc@plt  /* call in, e.g., libdl.so.2 */
             ...
calloc@plt:  jmp *0x1c(%ebx)  /* jmp through GOT slot */
             ...
calloc:      push %ebp  /* target of call in libc.so.6 */
             ...
```

Listing B.1: Example of PLT-based position independent code

dynamic loader then reads the section header table, handles all relocation entries in the current DSO, loads all auxiliary shared libraries, and resolves all imported symbols. If there is no interpreter set then the kernel starts the static binary directly.

Applications load libraries to dynamic (non-constant) addresses in their process image. These DSOs must therefore be compiled as Position Independent Code (PIC). DSOs that use shared libraries contain two additional tables, the GOT and the PLT. These tables make it possible to share most physical memory of the libraries between processes as relocation information is not stored in the code itself but in the GOT. The GOT contains pointers to symbols referenced in the current

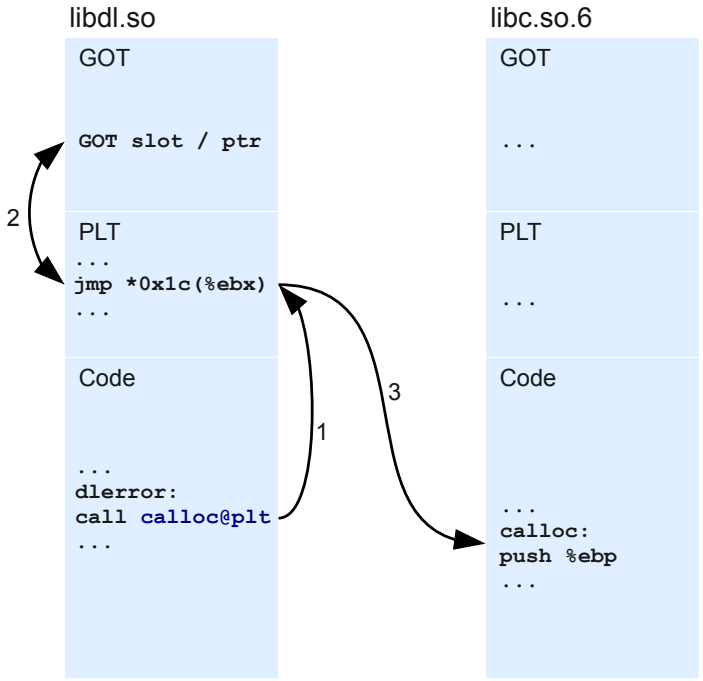

Figure B.2: Example of PLT-based position independent code. The function in the code section transfers control to the PLT slot (1), the code in the PLT slot executes a lookup in the GOT section (2), and transfers code to the other shared object (3).

DSO. The dynamic linker calculates the pointers in the GOT during relocation. Using a GOT enables code in the current DSO to access symbols in other DSOs without knowing the absolute address of the code. The base-address of the other DSO changes every time the application is started and the dynamic linker fixes the offset in the GOT. The GOT entries for shared variables must be relocated when the DSO is loaded. The PLT, on the other hand, uses lazy on-demand initialization. A function is resolved the first time it is referenced. PLT is used to transfer control to symbols in other DSOs, where entries in the PLT correspond to an indirect jump through an GOT slot. See Figure B.2 and Listing B.1 for an example.

Both the dynamic loader and the linker use the section header table. Depending on the linker flags the section header table of a linked executable or shared library contains information about all functions and symbols, or it contains only information about exported symbols.

The `dynsym` section contains the location information of all exported symbols and is available in all dynamically linked libraries and applications. This section is used by the dynamic loader to resolve references.

The `symtab` section is not needed by the dynamic loader and can be stripped from the final library or application. This section contains detailed information at the highest possible level of granularity and includes details of all available symbols, even static and weak variables or functions, hidden symbols, and individual object files.

The granularity of the `dynsym` and `symtab` information is no longer at the section level but on the level of individual compiled objects (usually individual source files) and functions. If available this information maybe used, e.g., by the debugger.

C

System call interface

A program running in user-space uses system calls to request services from the operating system. The operating system runs at a higher privilege level than the user-space processes. System calls implement the interface between applications (individual processes and threads) and the operating-system (the kernel). The separation between the operating system and processes is enforced using different hardware modes. These modes are called rings for Intel x86 hardware. Typical Unix-like kernels provide functionality for process control, file management, device management, information maintenance, and communication.

System calls are selected by a specific system call number. The mapping between system call and system call number can change between different hardware platforms and Instruction Set Architecture (ISA)s. Linux supports two system call interfaces using either software interrupts or `sysenter` instructions. Figure C.1 shows a sample invocation of the `getpid` system call.

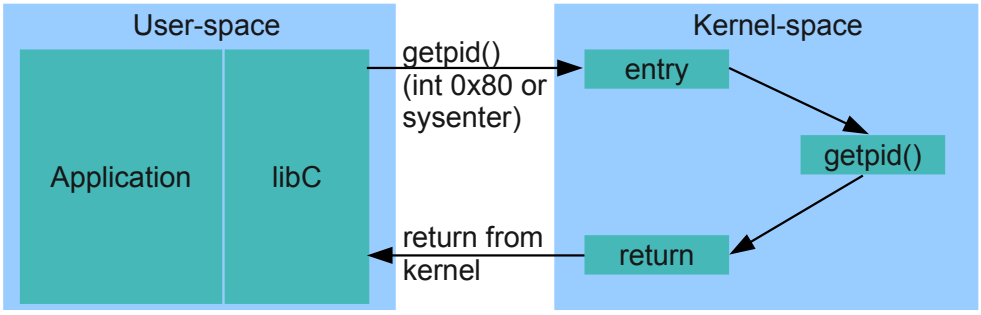

Figure C.1: Example of a system call invocation.

C.1 Argument passing

On x86 the `%eax` register holds the system call number. Up to 5 arguments to the system call are passed in registers (`%ebx`, `%ecx`, `%edx`, `%esi`, and `%edi`). System calls that exceed 5 arguments push all arguments on the user-stack (e.g., the `mmap` system call), or the arguments are passed using a pointer to a `struct`.

C.2 Software interrupts

Until Linux 2.5 the preferred way to execute system calls was to use software interrupts. Application programs use the `int 0x80` instruction to request services from the operating system. The system call number and arguments are passed according to Section C.1.

Execution of software interrupts results in a trap and a branch to the kernel interrupt service routine (ISR). Upon kernel entry all user state is saved and the appropriate system call is executed.

C.3 `sysenter` instruction

Newer processors feature the `sysenter` instruction that enables a fast switch to ring 0. This instruction does not store any context information. The application in user-space must ensure that all needed registers and the current instruction pointer are saved before the system call is executed.

Linux implements a virtual dynamic shared object (VDSO). This shared library is exported from the kernel and imported into the application by the Linux loader. The VDSO page defines `__kernel_vsyscall` that stores callee-safe registers and executes the actual `sysenter` instruction. The system call number and arguments are passed in registers according to Section C.1.

This faster way of executing system calls was introduced in Linux 2.5 because executing software interrupts on the processors available at that time was some orders of magnitude slower than executing `sysenter` instructions.

D

CFI micro benchmarks

Listing D.1 shows the micro benchmarks that evaluate dynamic control flow integrity.

```
#include <stdio.h>
#include <sys/time.h>
#include "libfastbt.h"
#define NUM_ITER 1000000000

static void jmpindirect(int nriter) {
  int i, loc;
  for (i = 0; i < nriter; ++i) {
    asm("movl $.out, %%eax\n"
        "movl %%eax, %0\n"
        "jmp *%0\n"
        ".out: nop\n" : : "m"(loc)
        : "eax", "memory");
}}
static void callindirect(int nriter) {
  int i, loc;
  for (i = 0; i < nriter; ++i) {
    asm("movl $.ciout, %%eax\n"
        "movl %%eax, %0\n"
        "call *%0\n"
        ".ciout: leal 4(%%esp), %%esp\n"
        : : "m"(loc) : "eax", "memory");
}}
static void call(int nriter) {
  int i;
  for (i = 0; i < nriter; ++i) {
    asm("call .cout\n"
        ".cout: leal 4(%%esp), %%esp\n"
        : : : "memory");
}}
static void ret(int nriter) {
  int i;
  for (i = 0; i < nriter; ++i) {
    asm("pushl $.rout\n"
        "ret\n"
        ".rout: nop\n" : : : "memory");
}}
static void jcc(int nriter) {
  int i, ret = 0;
  for (i = 0; i < nriter; ++i) {
    if ((i%2) == 0) ret+=3; else ret-=2;
}}
static void jump(int nriter) {
  int i;
  for (i = 0; i < nriter; ++i) {
    asm("jmp .jout\n"
        "nop\n.jout: nop");
}}
static void measure(int (*fctptr)(int),
                    char *name)
{
  struct timeval start, stop;
  double total;
  gettimeofday(&start, 0);
  fctptr(NUM_ITER);
  gettimeofday(&stop, 0);
  total = stop.tv_sec - start.tv_sec;
  if (stop.tv_usec - start.tv_usec < 0)
    total++;
  total *= 1000*1000;
  total += stop.tv_usec - start.tv_usec;
  total /= (NUM_ITER/1000);
  printf("%s %10f ns\n", name, total);
  struct thread_local_data *tld = \
    fbt_init(NULL);
  fbt_start_transaction(tld,
    fbt_commit_transaction);
  gettimeofday(&start, 0);
  fctptr(NUM_ITER);
  gettimeofday(&stop, 0);
  fbt_commit_transaction();
  fbt_exit(tld);
  total = stop.tv_sec - start.tv_sec;
  if (stop.tv_usec - start.tv_usec < 0)
    total++;
  total *= 1000*1000;
  total += stop.tv_usec - start.tv_usec;
  total /= (NUM_ITER/1000);
  printf("%s %10f ns\n", name, total);
}

int main() {
  measure(&jmpindirect, "jmpindirect");
  measure(&callindirect, "callindirect");
  measure(&jcc, "jcc");
  measure(&call, "call");
  measure(&jump, "jump");
  measure(&ret, "ret");
  return 0;
}
```

Listing D.1: Micro benchmark to evaluate dynamic CFI overhead

E

Libdetox evaluation with additional optimizations

This chapter shows the evaluation of the first libdetox version without the separate secure loader [PG10, PG11]. This version uses two additional optimizations for indirect jump instructions (compared to Section 5.2.6) and a different benchmark machine:

- The *indirect jump prediction* caches the last two jump targets (and their translated counterparts) instead of just one as in the current implementation.
- The *indirect jump jumptable* is an optimization that uses the fact that most indirect jumps that use a jump table are translated in a similar way (e.g. `jmp table_base(,%xxx, 4)`) where `table_base` is the address of the jump table and `%xxx` is the register containing the offset into the table. This pattern is recognized, and the jump table is replaced by a shadow jump table which removes the mapping table lookup.

These two optimizations have been removed in the current reimplementation for simplicity reasons. The benchmarks are run under Ubuntu 9.04 on an E6850 Intel Core2Duo CPU running at 3.00GHz, 2GB RAM, and GCC version 4.3.3. Averages are calculated by comparing overall execution time for all programs of untranslated runs against translated runs. The SPEC CPU2006 benchmarks are presented as a way to compare performance with other systems.

Table E.1 and E.2 display overheads for all SPEC CPU2006 benchmarks compared to an untranslated run. This evaluation uses the `LD_PRELOAD` feature to hijack control flow at startup. The different configurations are:

BT: A configuration without additional security features, showing the overhead of the isolation and binary modification toolkit.

libdetox: This configuration shows libdetox's overhead using the default guards.

libdetox+mprot: The last configuration shows full encapsulation including protection of internal data structures using explicit memory protection through `mprotect`.

Benchmark	**BT**	**libdetox**	**libdetox +mprot**
400.perlbench	55.97%	59.88%	74.69%
401.bzip2	3.89%	5.39%	5.54%
403.gcc	20.86%	22.68%	55.56%
429.mcf	-0.49%	0.49%	0.25%
445.gobmk	18.17%	14.57%	16.69%
456.hmmer	4.64%	4.75%	5.72%
458.sjeng	24.62%	27.65%	31.22%
462.libquantum	0.98%	0.98%	0.98%
464.h264ref	6.17%	9.20%	9.20%
471.omnetpp	13.91%	14.11%	15.12%
473.astar	3.66%	3.83%	4.33%
483.xalancbmk	23.72%	27.22%	31.27%
410.bwaves	2.12%	2.68%	3.91%
416.gamess	-3.50%	-4.20%	-0.70%
433.milc	0.97%	2.18%	3.26%
434.zeusmp	-0.13%	-0.25%	0.13%
435.gromacs	0.00%	0.00%	0.00%
436.cactusADM	0.00%	-0.66%	0.00%
437.leslie3d	0.00%	0.00%	0.86%
444.namd	0.65%	0.65%	0.65%
447.dealII	44.20%	41.12%	43.66%
450.soplex	7.25%	5.02%	7.25%
453.povray	22.10%	25.14%	26.52%
454.calculix	-1.68%	-0.56%	-1.12%
459.GemsFDTD	1.79%	1.79%	2.68%
465.tonto	9.19%	10.27%	12.43%
470.lbm	0.00%	0.00%	-0.11%
482.sphinx3	2.36%	2.25%	1.89%
Average	**6.00%**	**6.39%**	**8.21%**

Table E.1: Overhead for different configurations executing the SPEC CPU2006 benchmarks (relative to an untranslated run). +mprot: libdetox with full memory protection.

Table E.1 uses the standard SPEC CPU2006 benchmarks and shows the overhead for long running programs. The average slowdown for binary translation (and no other transformation) for the full SPEC CPU2006 benchmarks is 6.0%. The libdetox security extensions increase the overhead a little to 6.4%. The full protection mechanism results in an overhead of 8.2%. The overhead for binary translation and basic libdetox protection for most programs is between -3.5% and 4.0%; some benchmarks like 400.perlbench, 433.gcc, 453.sjeng, 483.xalancbr, 447.dealII, and 453.povray re-

Benchmark	**BT**	**libdetox**	**libdetox +mprot**
400.perlbench	29.49%	29.74%	1158.97%
401.bzip2	2.28%	2.78%	9.11%
403.gcc	41.38%	41.38%	588.97%
429.mcf	0.80%	-0.40%	8.37%
445.gobmk	18.10%	14.29%	47.14%
456.hmmer	5.80%	6.68%	12.48%
458.sjeng	23.91%	24.11%	39.13%
462.libquantum	32.69%	10.58%	118.27%
464.h264ref	6.09%	10.87%	14.78%
471.omnetpp	65.58%	50.49%	195.45%
473.astar	3.67%	6.42%	7.34%
483.xalancbmk	55.81%	68.22%	2597.67%
410.bwaves	1.72%	2.59%	6.90%
416.gamess	8.52%	4.17%	426.96%
433.milc	2.38%	6.74%	15.03%
434.zeusmp	-0.49%	0.00%	3.88%
435.gromacs	1.71%	0.34%	25.26%
436.cactusADM	0.25%	0.25%	24.56%
437.leslie3d	0.35%	0.70%	2.11%
444.namd	1.20%	1.20%	4.82%
447.dealII	36.40%	35.20%	42.80%
450.soplex	101.75%	13.41%	2316.91%
453.povray	23.12%	16.12%	185.29%
454.calculix	58.60%	11.69%	1566.67%
459.GemsFDTD	7.21%	7.46%	39.30%
465.tonto	30.56%	27.08%	182.64%
470.lbm	0.93%	0.47%	4.36%
482.sphinx3	12.44%	12.44%	54.22%
Average	**9.60%**	**9.93%**	**48.40%**

Table E.2: Overhead for different configurations executing the SPEC CPU2006 benchmarks with the test data set (relative to an untranslated run). mprot tBT: libdetox with full memory protection.

sult in a higher overhead of 23% to 60% due to many indirect control flow transfers that cannot be optimized. The speedup of some programs is achieved by a better code layout through the translation process. libdetox adds static overhead per translated block and per system call. The SPEC CPU20006 benchmarks have a low number of system calls and high code reuse, which is typical for server applications. Therefore the libdetox extensions add no measurable overhead to these programs.

References

[ABEL05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS'05: Proc. 12th Conf. Computer and Communications Security*, pages 340–353, 2005.

[ACR+08] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *SP'08: Proc. 2008 IEEE Symposium on Security and Privacy*, pages 263–277, 2008.

[AJ06] A. Aggarwal and P. Jalote. Monitoring the security health of software systems. In *ISSRE'06: 17th Int'l Symp. Software Reliability Engineering*, pages 146 –158, nov. 2006.

[AKS99] Albert Alexandrov, Paul Kmiec, and Klaus Schauser. Consh: Confined execution environment for internet computations. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.488`, 1999.

[Ale96] Aleph1. Smashing the stack for fun and profit. *Phrack*, 7(49):`http://phrack.com/issues.html?issue=49&id=14`, November 1996.

[AR00] Anurag Acharya and Mandar Raje. MAPbox: using parameterized behavior classes to confine untrusted applications. In *SSYM'00: Proc. 9th Conf. USENIX Security Symp.*, pages 1–17, 2000.

[BA05] Derek Bruening and Saman Amarasinghe. Maintaining consistency and bounding capacity of software code caches. In *CGO '05*, pages 74–85, 2005.

[Bau06] Mick Bauer. Paranoid penguin: an introduction to Novell AppArmor. *Linux J.*, 2006(148):13, 2006.

[BBSD05] Sandeep Bhatkar, Eep Bhatkar, R. Sekar, and Daniel C. Duvarney. Efficient techniques for comprehensive protection from memory error exploits. In *SSYM'05: Proc. 14th USENIX Security Symp.*, pages 255–270, 2005.

[BD96] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Journal for Computing Systems*, pages 131–152, 1996.

[BDA01] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *ACM Workshop Feedback-directed Dyn. Opt. (FDDO-4)*, 2001.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00*, pages 1–12, 2000.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03*, pages 164–177, 2003.

[BDS03] Eep Bhatkar, Daniel C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *SSYM'03: Proc. 12th USENIX Security Symp.*, pages 105–120, 2003.

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proc. USENIX ATC*, pages 41–41, 2005.

[BGA03] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03*, pages 265–275, 2003.

[Bis95] Matt Bishop. Checking for race conditions in file accesses. Technical report, University of California at Davis, 1995.

[BJFL11] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS'11: Proc. 6th ACM Symp. on Information, Computer and Communications Security*, pages 30–40, 2011.

[BJSW05] Nikita Borisov, Rob Johnson, Naveen Sastry, and David Wagner. Fixing races for fun and profit: how to abuse atime. In *SSYM'05: 14th USENIX Security Symp.*, pages 303–314, 2005.

[BKGB06] Derek Bruening, Vladimir Kiriansky, Timothy Garnett, and Sanjeev Banerji. Thread-shared software code caches. In *CGO '06*, pages 28–38, 2006.

[Bro11] Tim Brown. Breaking the links: Exploiting the linker. `http://www.nth-dimension.org.uk/pub/BTL.pdf`, 2011.

[BST00] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proc. USENIX ATC*, pages 251–262, 2000.

[Bug04] Edouard Bugnion. Dynamic binary translator with a system and method for updating and maintaining coherency of a translation cache. US Patent 6704925, March 2004.

[CBB+01] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. Formatguard: automatic protection from printf format string vulnerabilities. In *SSYM'01: Proc. 10th USENIX Security Symp.*, 2001.

[CBJW03] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointguardTM: protecting pointers from buffer overflow vulnerabilities. In *SSYM'03: Proc. 12th USENIX Security Symp.*, 2003.

[CBKH+00] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Parsimonious server security. In *Proc. 14th USENIX Conf. System Administration*, pages 355–368, 2000.

[CBWKh01] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-hartman. RaceGuard: Kernel protection from temporary file race vulnerabilities. In *SSYM'01: Proc. 10th USENIX Security Symp.*, page 12, 2001.

[CCM+09] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *SOSP'09*, pages 45–58, 2009.

[CDD+10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In Angelos Keromytis and Vitaly Shmatikov, editors, *CCS'10: Proceedings of CCS 2010*, pages 559–572. ACM Press, 2010.

[CGC08] J. Chow, T. Garfinkel, and P.M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. USENIX ATC*, pages 1–14, 2008.

[Che02] Brian V. Chess. Improving computer security using extended static checking. In *S&P'02: IEEE Symp. on Security and Privacy*, 2002.

[CHH+98] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. Fx!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, 1998.

[CHV10] Suresh Chari, Shai Halevi, and Wietse Venema. Where do you want to go today? escalating privileges by pathname manipulation. In *NDSS*, 2010.

[cor05] corbet. New system calls. `https://lwn.net/Articles/164887/`, 2005.

[Cor12a] Intel Corp. Intel 64 and IA-32 Intel Architecture Software Developer's Manual Combined Volumes 2A and 2B: Instruction Set Reference, A-Z, 2012.

[Cor12b] Intel Corp. Intel 64 and IA-32 Intel Architecture Software Developer's Manual Combined Volumes 3A and 3B: System Programming Guide, Parts 1 and 2, 2012.

[Cor12c] Intel Corp. Intel 64 and IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture, 2012.

[CPM+98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM'98: Proc. 7th USENIX Security Symp.*, 1998.

[CW02] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *CCS'02: Proc. 9th ACM Conf. Computer and Communications Security*, pages 235–244, 2002.

[CXM+11] Ping Chen, Xiao Xing, Bing Mao, Li Xie, Xiaobin Shen, and Xinchun Yin. Automatic construction of jump-oriented programming shellcode (on the x86). In *ASIACCS'11: Proc. 6th ACM Symp. on Information, Computer and Communications Security*, pages 20–29. ACM, 2011.

[Dan10] Vincent Danen. CVE-2011-1658: ld.so ORIGIN expansion combined with RPATH. `https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2011-1658`, 2010.

[DBR02] Scott W. Devine, Edouard Bugnion, and Mendel Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. US Patent 6397242, 2002.

[DH04] Drew Dean and Alan J. Hu. Fixing races for fun and profit: how to use access(2). In *SSYM'04: Proc. 13th USENIX Security Symp.*, SSYM'04, pages 14–14, 2004.

[Dre10] Ulrich Drepper. How to write shared libraries. `http://www.akkadia.org/drepper/dsohowto.pdf`, Dec. 2010.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL '84*, pages 297–302, 1984.

[EAV+06] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *OSDI'06*, pages 75–88, 2006.

[Erl07] Ú. Erlingsson. Low-level software security: Attacks and defenses. *FOSAD'07: Foundations of security analysis and design*, pages 92–134, 2007.

[Eva09] Chris Evans. Linux kernel "seccomp" facility minor vulnerability. CESA-2009-004, 2009.

[FC08] Bryan Ford and Russ Cox. Vx32: lightweight user-level sandboxing on the x86. In *Proc. USENIX ATC*, pages 293–306, 2008.

[FS01] Michael Frantzen and Michael Shuey. StackGhost: Hardware facilitated stack protection. In *SSYM'01: Proc 10th USENIX Security Symp.*, 2001.

[FS08] Christof Fetzer and Martin Suesskraut. Switchblade: enforcing dynamic personalized system call models. In *EuroSys'08: Proc. 3rd Europ. Conf. Computer Systems*, pages 273–286, 2008.

[Gar03] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS'03: Proc. Network and Distributed Systems Security Symp.*, pages 163–176, 2003.

[Gar06] Manu Garg. Sysenter based system call mechanism in linux 2.6. `http://manugarg.googlepages.com/systemcallinlinux2_6.html`, 2006.

[Gil51] S. Gill. The diagnosis of mistakes in programmes on the EDSAC. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 206(1087):538–554, 1951.

[GPR04] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS'04: Proc. Network and Distributed Systems Security Symp.*, 2004.

[GR03] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS'03: Proc. Network and Distributed Systems Security Symp.*, 2003.

[GSV03] Bharat Goyal, Sriranjani Sitaraman, and S. Venkatesan. A unified approach to detect binding based race condition attacks. In *CANS'03: Intl. Workshop on Cryptology & Network Security*, 2003.

[GWTB96] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *SSYM'06: Proc. 6th USENIX Security Symp.*, 1996.

[Haa10] Paul Haas. Advanced format string attacks. `https://www.defcon.org/images/defcon-18/dc-18-presentations/Haas/DEFCON-18-Haas-Adv-Format-String-Attacks.pdf`, DEFCON 18 2010.

[HFC+06] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *EuroSys'06: Proc. 1st Europ. Conf. Comp. Sys.*, pages 29–41, 2006.

[HK01] Etoh Hiroaki and Yoda Kunikazu. ProPolice: Improved stack-smashing attack detection. *IPSJ SIG Notes*, pages 181–188, 2001.

[HKZ+06] Jason Hiser, Naveen Kumar, Min Zhao, Shukang Zhou, Bruce R. Childers, Jack W. Davidson, and Mary Lou Soffa. Techniques and tools for dynamic optimization. In *IPDPS*, 2006.

[HS06] Kim Hazelwood and Michael D. Smith. Managing bounded code caches in dynamic binary optimization systems. *TACO '06*, 3(3):263–294, 2006.

[HWH+07] Jason D. Hiser, Daniel Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *CGO '07*, pages 61–73, 2007.

[JKDC05] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP'05: Proc. 20th ACM Symposium on Operating Systems Principles*, pages 91–104, 2005.

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *SSYM'02: Proc. 11th USENIX Security Symp.*, pages 191–206, 2002.

[KF01] Thomas Kistler and Michael Franz. Continuous program optimization: Design and evaluation. *IEEE Trans. Comput.*, 50(6):549–566, 2001.

[KR02] Calvin Ko and Timothy Redmond. Noninterference and intrusion detection. In *S&P'02: Proc. 2002 IEEE Symposium on Security and Privacy*, pages 177–187, 2002.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI'05*, pages 190–200, 2005.

[LSVS09] Zhenkai Liang, Weiqing Sun, V. N. Venkatakrishnan, and R. Sekar. Alcatraz: An isolated environment for experimenting with untrusted software. *ACM Trans. Inf. Syst. Secur.*, 12(3):1–37, 2009.

[May87] C. May. Mimic: a fast System/370 simulator. In *SIGPLAN '87: Papers of the Symposium on Interpreters and interpretive techniques*, pages 1–13, 1987.

[McC04] Steve McConnell. *Code Complete, Second Edition*, chapter 19. General Control Issues, page 30. Microsoft Press, Redmond, WA, USA, 2004.

[MCZ+11] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *SOSP'11*, pages 115–128, 2011.

[MK97] David Mazières and M. Frans Kaashoek. Secure applications need flexible operating systems. In *HotOS'07: Workshop on Hot Topics in Operating Systems*, pages 56–61, 1997.

[MM06] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *SSYM'06: Proc. 15th USENIX Security Symp.*, pages 209–224, 2006.

[Ner07] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 11(58):`http://phrack.com/issues.html?issue=67&id=8`, November 2007.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI'07*, pages 89–100, 2007.

[Orm10a] Tavis Ormandy. CVE-2010-3847: GNU C library dynamic linker $ORIGIN expansion vulnerability. `http://www.exploit-db.com/exploits/15274/`, 2010.

[Orm10b] Tavis Ormandy. CVE-2010-3856: GNU C library dynamic linker LD_AUDIT arbitrary DSO load vulnerability. `http://www.exploit-db.com/exploits/15304/`, 2010.

[OWA] OWASP. Definition of format string attacks. `https://www.owasp.org/index.php/Format_string_attack`.

[Pay09] Mathias Payer. secuBT: Hacking the hackers with user-space virtualization. In *Proceedings of the 26c3*, pages 157–163, 2009.

[Pay10] Mathias Payer. I control your code - attack vectors through the eyes of software-based fault isolation. In *Proceedings of the 27c3*, 2010.

[PB04] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2:20–27, 2004.

[PCC+96] Daryl Pregibon, Herman Chernoff, Bill Curtis, Siddhartha R. Dalal, Gloria J. Davis, Richard A. DeMillo, Stephen G. Eick, Bev Littlewood, and Chitoor V. Ramamoorthy. *National Research Council (U.S.). Panel on Statistical Methods in Software Engineering: Statistical Software Engineering.* National Academy Press, 1996.

[PcC03] Manish Prasad and Tzi cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proc. 12th USENIX ATC*, pages 211–224, 2003.

[PG09] Mathias Payer and Thomas R. Gross. Requirements for fast binary translation. In *2nd Workshop on Architectural and Microarchitectural Support for Binary Translation*, 2009.

[PG10] Mathias Payer and Thomas R. Gross. Generating low-overhead dynamic binary translators. In *Proc. 3rd Annual Haifa Experimental Systems Conf.*, SYSTOR '10, pages 22:1–22:14. ACM, 2010.

[PG11] Mathias Payer and Thomas R. Gross. Fine-grained user-space security through virtualization. In *VEE'11: Proc. 7th Int'l Conf. Virtual Execution Environments*, pages 157–168, 2011.

[PG12] Mathias Payer and Thomas R. Gross. Protecting applications against tocttou races by user-space caching of file metadata. In *VEE'12: Proc. 8th Int'l Conf. Virtual Execution Environments*, 2012.

[PHG12] Mathias Payer, Tobias Hartmann, and Thomas R. Gross. Safe loading - a foundation for secure execution of untrusted programs. In *S&P'12: Proc. Int'l Symp. on Security and Privacy*, 2012.

[Pla10] Captain Planet. A eulogy for format strings. *Phrack*, 14(67):`http://phrack.com/issues.html?issue=67&id=8`, 2010.

[PLLK04] Jongwoon Park, Gunhee Lee, Sangha Lee, and Dong-Kyoo Kim. RPS: An extension of reference monitor to prevent race-attacks. In *PCM'04: 5th Pacific Rim Conf. on Multimedia*, pages 556–563, 2004.

[POS08] POSIX. openat syscall. `http://linux.die.net/man/2/openat`, 2008.

[Pro03] Niels Provos. Improving host security with system call policies. In *SSYM'03: Proc. 12th USENIX Security Symp.*, 2003.

[PT03] PaX-Team. PaX ASLR (Address Space Layout Randomization). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[Ros10] Dan Rosenberg. CVE-2010-0830: Integer overflow in ld.so. `http://drosenbe.blogspot.com/2010/05/integer-overflow-in-ldso-cve-2010-0830.html`, 2010.

[Sch00] Bruce Schneier. Software complexity and security. `https://www.schneier.com/crypto-gram-0003.html#8`, 2000.

[SCO96] SCO. System V Application Binary Interface, Intel386 Architecture Processor Supplement. `http://www.sco.com/developers/devspecs/abi386-4.pdf`, 1996.

[SCW+05] Benjamin Schwarz, Hao Chen, David Wagner, Jeremy Lin, Wei Tu, Geoff Morrison, and Jacob West. Model checking an entire Linux distribution for security violations. In *Proc 21st Computer Security Applications Conference*, pages 13–22, 2005.

[SD01] Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. Technical report, University of Virginia, 2001.

[SD02] Kevin Scott and Jack Davidson. Safe virtual execution using software dynamic translation. *ACSAC'02: Annual Comp. Security Applications Conf.*, page 209, 2002.

[SGC+09] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling transactional file access via lightweight kernel extensions. In *FAST'09: Proc. 7th Conf. on File and storage technologies*, pages 29–42, 2009.

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS'07: Proc. 14th Conf. on Computer and Communications Security*, pages 552–561, 2007.

[sLC05] Kyung suk Lhee and Steve J. Chapin. Detection of file-based race conditions. *Int'l Journal Information Security*, 4(1-2):105–119, 2005.

[SPP+04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS'04: Proc. 11th Conf. Computer and Communications Security*, pages 298–307, 2004.

[SSB07] Swaroop Sridhar, Jonathan S. Shapiro, and Prashanth P. Bungale. HDTrans: a low-overhead dynamic translator. *SIGARCH Comput. Archit. News*, 35(1):135–140, 2007.

[SSNB06] Swaroop Sridhar, Jonathan S. Shapiro, Eric Northup, and Prashanth P. Bungale. HDTrans: an open source, low-level dynamic instrumentation system. In *VEE'06: Proc. 2nd Virtual Execution Environments*, pages 175–185, 2006.

[SW91] Frank Schmuck and Jim Wylie. Experience with transactions in quicksilver. In *SOSP'09: Proc. 13th ACM Symposium on Operating Systems Principles*, pages 239–253, 1991.

[THWDS08a] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably solving file TOCTTOU races with hardness amplification. In *FAST'08: Proc. 6th USENIX Conf. on File and Storage Technologies*, pages 13:1–13:18, 2008.

[THWDS08b] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably preventing file race attacks with user-mode path resolution. Technical Report RC24572, IBM T. J. Watson Research Center, June 2008.

[TY03] Eugene Tsyrklevich and Bennet Yee. Dynamic detection and prevention of race conditions in file accesses. In *SSYM'03: Proc. 12th USENIX Security Symp.*, pages 243–255, 2003.

[UJR05] Prem Uppuluri, Uday Joshi, and Arnab Ray. Preventing race condition attacks on file-systems. In *SAC'05: Proc. ACM Symposium on Applied computing*, SAC '05, pages 346–353, 2005.

[VBKM00] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: a static vulnerability scanner for C and C++ code. In *ACSAC'00: Ann. Comput. Security Applications Conf.*, 2000.

[vdVM04] Arjan van de Ven and Ingo Molnar. Exec shield. `https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`, 2004.

[vla11] vladz. Xorg file permission change vulnerability (CVE-2011-4029). `http://vladz.devzero.fr/Xorg-CVE-2011-4029.txt`, 2011.

[WALK10] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *SSYM'10: 19th USENIX Security Symp.*, pages 29–46, 2010.

[WCS+02] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *SSYM'02: Proc. 11th USENIX Security Symp.*, 2002.

[WKCG00] Ronnie Chaiken Wen-Ke Chen, Sorin Lerner and David M. Gillies. Mojo: A dynamic optimization system. In *ACM Workshop Feedback-directed Dyn. Opt. (FDDO-3)*, 2000.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP'93*, pages 203–216, 1993.

[WP05] Jinpeng Wei and Calton Pu. TOCTTOU vulnerabilities in UNIX-style file systems: an anatomical study. In *FAST'05: Proc. 4th Conf. USENIX Conf. File and Storage Technologies*, pages 12–12, 2005.

[WP06] Jinpeng Wei and Calton Pu. A methodical defense against TOCTTOU attacks: the EDGI approach. In *ISSSE'06: IEEE Int'l Symp. on Secure Software Engineering*, 2006.

[WSSZ07] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending ACID semantics to the file system. *Trans. Storage*, 3, June 2007.

[YSD+09] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symp. on Security and Privacy*, pages 79–93, 2009.

Acronyms

ASLR Address Space Layout Randomization. 2, 42–44

CFG Control Flow Graph. 31, 32, 43, 44, 48

CFI Control Flow Integrity. 25, 31–36, 44, 45, 48, 49, 86, 119, 120

DEP Data Execution Prevention. 2

DSO Dynamic Shared Object. 46, 47, 55, 56, 58, 71, 83, 85, 126–128

ELF Executable and Linkable Format. 11, 24, 36, 38, 42, 44, 48, 54–56, 75, 119, 126

GOT Global Offset Table. 18, 22, 38, 56, 71, 127, 128

ISA Instruction Set Architecture. 11, 12, 23, 24, 30, 34, 59, 125, 129

PLT Procedure Linkage Table. 22, 38–40, 56, 58, 71, 87, 88, 124, 127, 128

SFI Software-based Fault Isolation. 2, 5–8, 24, 25, 30, 33, 35, 36, 38, 45, 49, 51, 53, 54, 58, 66, 85, 88

SUID Set User ID upon execution. 3, 22, 39, 95, 96, 114

TOCTTOU Time Of Check To Time Of Use. 29, 30, 34, 93–96, 114–116

Curriculum Vitae

Mathias Josef Payer

April 29, 1981	Born in Grabs, SG, Switzerland
1988 – 1993	Primary School in Schaan, Liechtenstein
1993 – 2001	High school (Gymnasium) in Vaduz, Liechtenstein
2001 – 2006	Master of Science ETH in Computer Science, ETH Zurich, Switzerland (major: system software, minor: robotics)
2005	Teaching assistant during graduate studies for Compiler Design I, ETH Zurich, Switzerland
2006 – 2012	Doctoral Studies at ETH Zurich, Switzerland
2010	Research-oriented internship in the anti-phishing team, Google Inc., Mountain View, CA, USA