# Securing Application Software in Modern Adversarial Settings

Dissertation zur Erlangung des Grades eines Doktor-Ingenieurs der Fakultät für Elektrotechnik und Informationstechnik an der Ruhr-Universität Bochum

vorgelegt von

**Felix Schuster**

aus Hattingen

im Juli 2015

| | |
|---|---|
| Berichter: | Prof. Dr. Thorsten Holz |
| | Prof. Dr. Ahmad-Reza Sadeghi |
| Tag der mündlichen Prüfung: | 15. September 2015 |

## Abstract

Two big trends that can be observed in the way that software is developed and used are the commoditization of software components and computing infrastructure:

It has long become the norm that software products are not entirely developed by a single trusted entity. Instead, even large software vendors routinely rely on software components from different and numerous external sources. These sources include typically public open source projects or external contractors.

Changes of similar nature can be observed in the ways that software is used and deployed; without doubt has *cloud computing* been one of the biggest trends in IT in recent years. The central concept of cloud computing is that the users' software no longer runs on their own systems but rather on demand, in a cost-effective manner, in the data centers of a cloud provider. This arrangement is also referred to as "infrastructure as a service".

Due to these developments new attack surfaces for application software arise: Classically, the security of application software is considered in adversarial settings where a trusted application is running in a largely trusted environment. The attacker is only foreseen to partly control inputs and outputs of the application. A typical concrete scenario is here for example a web browser in which the attacker triggers a buffer overflow vulnerability using a malicious web site.

This one-dimensional attacker model appears not always appropriate in the context of software components from external sources and cloud computing as it disregards important risks: a software component could contain a "backdoor" or a malicious cloud administrator could access the code and data of a cloud application at runtime.

Hence, this dissertation explores the topic of application software security in three modern adversarial settings: *(i)* the *classic* setting, *(ii)* the *backdoor* setting in which the attacker may additionally add a backdoor to a component of a software, and *(iii)* the *cloud* setting in which the attacker largely controls hardware and software and may at different places read or manipulate a cloud applications code and data.

In this dissertation, we begin with an evaluation of existing defensive approaches (e. g., *control-flow integrity*) in the classic setting. Thereto, we present various advanced *code-reuse* attacks. Our attacks break with commonly held assumptions on the nature of code-reuse attacks and as such bypass many existing academic and commercial defenses. Among others, our results here show that purely intuitive arguments or limited empirical evidence are no sufficient criteria for the security of a defense.

We discuss the challenges of the *backdoor* and the *cloud* adversarial settings and propose and evaluate novel ways to tackle them. We present a dynamic analysis approach for the detection and dismantling of backdoors in binary server applications across different processor architectures. Among others, we demonstrate how our approach can disarm real-world backdoors (e. g., in a malicious version of OpenSSH) in a fully automated manner.

Furthermore, we describe the first end-to-end secure system for the execution of distributed (MapReduce) applications in the untrusted cloud. The security of this system founds on two novel cryptographic protocols—for which we provide proof—and Intel's SGX technology as hardware-rooted *trusted computing base* (TCB).

## Zusammenfassung

Die Art und Weise in der Software entwickelt und genutzt wird hat sich in der näheren Vergangenheit gewandelt. So werden Softwareanwendungen heutzutage nur noch selten komplett von einer einzelnen vertrauenswürdigen Partei entwickelt. Selbst große Softwarehersteller bauen mittlerweile in ihren Produkten häufig auf Softwarekomponenten aus unterschiedlichen externen Quellen.

Veränderungen ähnlicher Art gibt es auch bei der Nutzung von Software: Cloud-Computing ist einer der großen IT-Trends der letzten Jahre. Das zentrale Konzept von Cloud-Computing ist, dass Software nicht mehr lokal auf den Systemen eines Anwenders läuft, sondern *on-demand* und kosteneffektiv in den Rechenzentren eines Cloud-Computing-Anbieters. Man spricht hier auch von „Infrastructure as a Service".

Durch diese Veränderungen ergeben sich neue Angriffsflächen für Software: Klassische Angreifermodelle in der Softwaresicherheit betrachten den Programmcode und die Ausführungsumgebung einer Anwendung typischerweise als vertrauenswürdig. Dem Angreifer wird nur die Möglichkeit zugestanden die Ein- und Ausgaben einer Anwendung in Teilen zu kontrollieren. Ein klassisches Szenario ist hier z.B. ein Web-Browser in dem vom Angreifer mit Hilfe einer bösartigen Webseite ein Pufferüberlauf (engl. *buffer overflow*) erzeugt wird. Dieses eindimensionale Angreifermodell erscheint im Kontext von Softwarekomponenten aus unterschiedlichen externen Quellen und von Cloud-Computing nicht immer weitgreifend genug – es lässt wichtige Fragestellungen außer Acht: *Was ist wenn eine Komponente einer Anwendung eine Hintertür enthält? Was ist wenn ein Cloud-Administrator mit Hardwarezugriff den Programmcode oder die Daten einer Cloud-Anwendung zur Laufzeit manipuliert oder liest?*

In dieser Dissertation wird daher das Thema Softwaresicherheit in drei Angreifermodellen bearbeitet: *(A)* Das klassische Modell, *(B)* das Modell „Externe Softwarekomponenten" in dem der Angreifer zusätzlich einmalig Hintertüren in bestimmte Komponenten einer Anwendung einbauen kann und *(C)* das Modell „Cloud-Computing" in dem der Angreifer weite Teile von Soft- und Hardware kontrolliert und an vielen Stellen Programmcode und Daten einer Cloud-Anwendung lesen und manipulieren kann.

Es werden zunächst existierenden Defensivansätzen (z.B. *Control-Flow Integrity*) im Modell A analysiert und bewertet. Dazu werden mehrere neuartige Code-Reuse-Angriffe präsentiert, die mit weit verbreiteten Annahmen brechen und so viele existierende akademische und kommerzielle Defensivmaßnahmen umgehen. Unter anderem zeigen die Ergebnisse hier, dass informelle Argumente oder rein empirische Belege kein hinreichendes Kriterium für die Sicherheit von Defensivmaßnahmen sind. Im Anschluss werden Herausforderungen der erweiterten Modelle B und C diskutiert und entsprechende neuartige Defensivmaßnahmen vorgeschlagen und evaluiert. Konkret wird für Modell B ein System zum Auffinden von bestimmten Arten von Hintertüren in Binärsoftware mittels dynamischer Analysen beschrieben. Für Modell C wird ein Ende-zu-Ende sicheres System zur Ausführung von verteilten Anwendungen in der Cloud beschrieben. Die Sicherheit dieses Systems beruht auf zwei kryptographischen Protokollen und verwendet Intels SGX-Technologie als *Trusted Computing Base* (TCB).

## Acknowledgements

First, I would like to express my sincere gratitude to my advisor Thorsten Holz who initially convinced me to pursue a PhD. During the three and a half years I spent in his Systems Security group at the Ruhr-Universität Bochum, Thorsten provided me with a great research environment and lots of freedom, but also invaluable guidance and advice. During this period I also had the privilege to do two remarkably educating internships in the Systems and Networking group at Microsoft Research in Cambridge. I kindly thank Manuel Costa for making this possible, for his support and advice, and for our fruitful collaborative work on the VC3 system.

Besides these two gentlemen, I have also been very fortunate to work and publish together with Ahmad-Reza Sadeghi, Andreas Maaß, Cédric Fournet, Christian Rossow, Christopher Liebchen, Christos Gkantsidis, Jannik Pewny, Lucas Davi, Lukas Bernhard, Marcus Peinado, Martin Steegmanns, Moritz Contag, Per Larsen, Stephen Crane, Stijn Volckaert, and Thomas Tendyck. Without their work and help (and also sometimes patience) this thesis would not have been possible—thank you!

My appreciation also goes to Andreas Fobian, Behrad Garmany, Carsten Willems, Fabian Yamaguchi, Johannes Dahse, Johannes Hoffmann, Konrad Rieck, Ralf Hund, Robert Gawlik, and Tilman Frosch for enlightening discussions and the good times spent together in Bochum, at conferences, or elsewhere. Finally, I wholeheartedly thank my family for their great support and encouragement at all times—especially my parents Ingrid and Norbert Schuster and Josefin Annemüller, the mother of our son Lennart.

# Contents

Chapter 1

# Introduction

The (in)security of software has for decades been a pressing problem and a hot topic for industry and academia alike. As such, a myriad of ways have been presented for securing software against threats in almost all conceivable scenarios. Classically, the security of software is considered in adversarial settings where an application is running in a largely trusted environment with an attacker attempting to compromise the software by misusing its external interfaces. As this is a rather abstract observation, consider for example the following two attack scenarios, which in this or a similar form are probably taking place hundreds of times a day across the Internet:

*Attack on a client application.* A user is lured, maybe by a "phishing" e-mail, to point his or her web browser to a malicious web site. By making the web browser parse carefully misshaped HTML code, the web site triggers a critical bug in the web browser, which in the consequence installs a malware, e. g., a "bot" or a "virus", on the user's computer.

*Attack on a server application.* This time the other way round, a malicious client connects to an Internet-facing server application, e. g., the web server of an online shop. By abusing the public interface of the server, the client provokes a critical bug and instead of a web site, the server delivers its internal data base—including customers' credit card numbers—to the remote attacker.

The root cause for all this mischief is easy to spot: to blame are the bugs, the programming errors in handling external input that make application software vulnerable to these kinds of attacks. Infamous bug classes that all could have enabled the two described attacks are for example: *buffer overflows* [10], *cross-site scripting* vulnerabilities [117], or *SQL injections* vulnerabilities [94]. At least in theory, many forms of *exploitable bugs* can be eradicated through careful protocol and data format design and systematic sanitization and parsing of input [39, 173]. Furthermore, a range of approaches exists that reliably mitigate the dangerous effects of such bugs in one way or another. For example, existing C/C++ code, which is among others notoriously prone to buffer overflows, can automatically be augmented with runtime checks such that unsafe memory accesses (like said buffer overflows) cannot happen [110, 143].

All that said, one question follows naturally: *Why hasn't the problem of insecure application software and insecure computing in general been solved yet?* The answer to this question is twofold and may be obvious to some:

**A** deploying provably strong measures to mitigate exploitable bugs is usually costly[1] in one way or another; and

**B** even software that is free of exploitable bugs can still very well be vulnerable in various ways.

The consequence of aspect **A** is that provably strong protections are oftentimes not applied in practice. Hence, the design of reliable yet cost-effective defenses which mitigate exploitable bugs and corresponding attack attempts is an ongoing struggle in academia and industry. In Chapter 2 we contribute to this struggle by assessing the practical strength of a range of recently proposed (and partly also commercially deployed) cost-effective defenses that aim to prevent the exploitation of memory access bugs (e. g., buffer overflows) in applications written in "unsafe" programming languages like C and C++.

The second aspect (**B**) implies that apart from input handling bugs—depending on the adversarial setting—there are also *other threats to the security of application software.*

**Other Threats to the Security of Application Software** It has long become the norm that software products are not entirely developed *top to bottom* by a single trusted entity, e. g., by a company's own in-house development team. Instead, even large software vendors routinely rely on software components from external sources such as public open source projects or external contractors. For example, at the time of this writing, Apple Inc. lists[2] more than 100 open source projects that are used in its proprietary Mac OS X operating system. Among other advantages, the employment of third-party software is cost-effective and convenient. Indeed, it should in the uttermost cases improve the overall security of a software if it is built on proven standard components like for example the ubiquitous OpenSSL. However, from a security perspective, there are also certain risks as Thompson famously stated in 1984 [210]:

"You can't trust code that you did not totally create yourself. [...] No amount of source-level verification or scrutiny will protect you from using untrusted code."

In line with this, one can formulate that it does not help much if a software is safe from conventional security-critical bugs if it contains purposely installed *backdoors.* For example, even if a server application does not suffer from buffer overflows, SQL injection vulnerabilities, or the like, there could still be a simple backdoor, e. g., a hidden key word, that allows a remote attacker to connect and steal the internal data base nonetheless. For example, an infamous backdoor incident occurred in 2010 when unknown attackers secretly broke into the source code repository of the ProFTPD server software to modify the functionality of the unsuspicious HELP command. In the consequence, until the backdoor was found and removed, ProFTPD would give a remote "root shell" (i. e., full remote system access) to anyone who asked for it.

[1] The cost for applying a defensive measure may encompass various things; among others are usually decreased runtime performance and the need to recompile or modify an application.

[2] See `https://www.apple.com/opensource/` (accessed 01/10/2015)

Evidently, in the case of backdoors, a revised adversarial setting is at hand in which the attacker cannot only interfere with an application's external interfaces but may also be able to add her own malicious code at a certain point in time. This adversarial setting is explored in Chapter 3 in which we describe novel ways for the identification and dismantling of backdoors in server applications.

A development complementary to the commoditization of software components is the ever more widespread adoption of *infrastructure as a service* (IaaS), also known under the catchier term "cloud computing". Without doubt is cloud computing one of the biggest trends in commercial computing of the recent past. Well-known computing companies from Amazon to Google to Microsoft are offering a broad range of cloud computing services in a market that by one estimate [204] had a volume in excess of $15 billion in 2014. The central concept of cloud computing is that software no longer runs on the local systems of users; instead, corporations and private consumers alike entrust *cloud providers* with the processing of their code and data. Typically, cloud providers make the capacities of their data centers available on demand to users, who seek to profit from increased flexibility and cost-effectiveness [21].

Of course data can safely be stored encrypted in the "cloud". However, as soon as an application software is to be run in the cloud, providers today always need access to their customers' data and code in plain form. This essentially requires customers to fully trust their cloud providers with the execution of their application software. Transitively, this translates to trust in the cloud provider's hardware and software infrastructure as well as its employees. For example a single malicious administrator within the ranks of a cloud provider can today effectively manipulate and steal all a cloud application's code and data at runtime from memory, e. g., by misusing privileged software or by physically applying a probe to a memory bus. Additionally, even if a cloud provider is considered absolutely trustworthy, the risk of external attackers exploiting conventional vulnerabilities in the cloud provider's software stack is also always given.

Accordingly, the adversarial setting for cloud applications can by any means only be described as challenging. Effectively, code and data of a cloud application have to be considered to be permanently under attacker control. While all hope may appear to be lost in this setting on first glance, we present in Chapter 4 a practical and provably end-to-end secure system for the execution of distributed applications in the cloud with reliance on Intel's SGX [133] technology as *trusted computing base* (TCB).

## 1.1 Modern Adversarial Settings

So far, we have outlined three different adversarial settings that are typical for modern software deployments. In summary, these adversarial settings are:

- Classic setting: The attacker controls certain inputs to an application. For example, the attacker may attempt to exploit a critical bug in the victim's web browser using a malicious web site.
- Backdoor setting: The attacker is able to once modify the code of a software component and add a *backdoor*. For example, the attacker may be a malicious

developer or may be an external intruder that temporarily obtains access to a source code repository.

- Cloud setting: The attacker controls the entire execution environment of an application and, hence, has unlimited control over the application's code and data. This is the typical cloud computing attacker model.

At its core, this dissertation concerns with the question of how to secure application software in these settings. While we focus thereby on applications written in C and C++, our results at least partly also extend to other programming languages or application software in general. For convenience, we use the short forms Classic, Backdoor, and Cloud in the following to refer to these settings.

## 1.2 Outline and Contributions

Each of the three defined adversarial settings is treated in a separate chapter. We now give an outline for each chapter, list its contributions, and enumerate the peer-reviewed publications it is based on. The full list of the publications that the author contributed to during the course of the work on this dissertation is given on page 143.

**Challenging and Improving Existing Defenses against Code-Reuse Attacks.** We begin with the exploration of the Classic setting in Chapter 2. In that chapter, we thoroughly introduce the issue of memory access errors/bugs that have for decades plagued applications written in unsafe programming languages and are today especially a concern in the context of C and C++. We explain the fundamentals of *control-flow hijacking* and common forms of *code-reuse attacks* and systematically introduce corresponding defenses that have been proposed or actually deployed in the recent past. Following this introduction, we formulate our observation that many defenses against code-reuse attacks offer only intuitions or limited empirical evidence as arguments for their effectiveness. Subsequently, follow the main scientific contributions of Chapter 2:

- we present novel variants of the ROP attack and a completely new form of code-reuse attack dubbed COOP;
- with these attacks we demonstrate that many different defenses that were believed to be "good enough" are in fact not; and
- we sketch designs for better future defenses.

These contributions have already been published in similar forms at different academic venues: the ROP attack variants were published jointly with Tendyck, Pewny, Maaß, Steegmanns, Contag, and Holz in the paper *Evaluating the Effectiveness of Current Anti-ROP Defenses* [180] at the $17^{th}$ *International Symposium on Research in Attacks, Intrusions and Defenses* and, in more detail, also in an accompanying technical report [181]. The COOP attack technique was first published jointly with Tendyck, Liebchen, Davi, Sadeghi, and Holz in a paper titled *Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications* [179] at the $36^{th}$ *IEEE*

*Symposium on Security and Privacy*. We further generalized the techniques from this paper in a yet unpublished joint work with Crane, Volckaert, Liebchen, Larsen, Davi, Sadeghi, Holz, De Sutter, and Franz titled *It's a TRAP: Table Randomization and Protection against Function Reuse Attacks* [59].

**Towards the Mitigation of Backdoors in Software.** After the discussion of the CLASSIC setting in Chapter 2, we proceed with the extended BACKDOOR setting in Chapter 3. The chapter starts with an intuitive definition of the term "backdoor" and discusses different attack scenarios to support and to exemplify this definition. Further, we highlight the unique challenges that exist in the BACKDOOR setting. Subsequently, we present our novel approach for the detection and dismantling of certain types of backdoors in binary server applications. Specifically, the types of backdoors our approach concerns with are *flawed authentication routines* and *hidden commands*. We explain how, by repeatedly invoking a server application according to a formal protocol description and observing the resulting control flow, these types of backdoors can be identified or disabled using different heuristics. Finally, the implementation of our approach in the form of tool called WEASEL is described and an extended evaluation of WEASEL is conducted on real-world and artificial backdoors. In particular, the following scientific contributions are made:

- we introduce an automated way to identify critical parts in server applications that are typically prone to backdoors;
- we implemented our techniques in a tool called WEASEL for x86-32, x86-64 and MIPS32 systems running different versions of Linux; and
- we show how several real-world backdoors for ProFTPD and OpenSSH can successfully be detected or disabled using WEASEL and demonstrate the applicability of our approach to different platforms, including COTS embedded devices.

These results and contributions were published together with Thorsten Holz in a paper titled *Towards Reducing the Attack Surface of Software Backdoors* [177] at the $20^{th}$ *ACM Conference on Computer and Communications Security*.

**Trustworthy Data Analytics in the Cloud using SGX.** In Chapter 4, finally the third of our adversarial settings is considered: the CLOUD setting. The chapter starts with an introduction to the unique and yet unsolved security challenges for application software in the cloud. We identify that cloud users typically should have a strong interest in integrity and confidentiality guarantees for their code and data while both are being processed on remote untrusted systems. It is outlined how existing approaches fall short to achieve this in a practical and end-to-end manner. Subsequently, Intel's upcoming SGX instruction set extension for x86 is introduced and we detail the design of our *VC3* (short for *Verifiable Confidential Cloud Computing*) system on top of it. In particular, two novel lightweight cryptographic protocols are presented which enable *VC3* to run distributed MapReduce[3] applications programmed in C++ securely within stock Hadoop[4] environments. We discuss the properties and shortcomings of *VC3* and its protocols and

[3]MapReduce is a widely used programming paradigm for distributed applications.

[4]Apache Hadoop is probably one of the most widely used frameworks for the distributed execution of MapReduce applications.

provide formal proofs for their security. In the remainder of the chapter, future applications of *VC3* are discussed and the results of an extensive and realistic performance evaluation are given. In summary, Chapter 4 makes the following scientific contributions:

- We describe *VC3*, the first system that executes distributed applications in the cloud at close to native speed while guaranteeing *confidentiality* and *integrity* of code and data. *VC3* also allows for the verification of a distributed job's results.
- *VC3* relies on two novel lightweight cryptographic protocols and we provide proof for their correctness and security.
- We report on a practical implementation of *VC3* that is based on Intel's upcoming SGX architecture. *VC3* is the first distributed cloud application using SGX.

These results were published at the $36^{th}$ *IEEE Symposium on Security and Privacy* in a joint paper with Costa, Fournet, Gkantsidis, Peinado, Mainar-Ruiz, and Russinovich titled *VC3: Trustworthy Data Analytics in the Cloud using SGX* [176]. The proofs were published by the same group of authors in a corresponding technical report [175].

Chapter 2

# Challenging and Improving Existing Defenses against Code-Reuse Attacks

In this chapter, we analyze the status quo of application software security in the classic adversarial setting (Classic) where an attacker attempts to compromise an application by triggering a bug in it. Specifically, we present and discuss new practical forms of so-called code-reuse attacks. Our attacks break with common assumptions and reveal inherent weaknesses of existing defenses against code-reuse attacks. We also discuss new approaches for the prevention of our attacks.

We commence with an introduction to the Classic setting in the context of unsafe programming languages (Section 2.1). Subsequently, background on control-flow hijacking and code-reuse attacks and corresponding defenses is given and related work is introduced (Section 2.2). We motivate our research (Section 2.3) before we proceed with the detailed discussions of our attacks (Section 2.4 and Section 2.5) and conclude (Section 2.6).

## 2.1 Adversarial Setting

For decades, attackers have been exploiting erroneous memory accesses (abbreviated *memory errors*) to hijack the control flow of software written in unsafe programming languages like C or C++ [205]. Such attacks against software are usually exercised in some form of a client-server scenario where either the client or the server may be the malicious entity under the attacker's control. In the following, we refer to this as the Classic setting.

For example, a file server compiled from trusted but not necessarily bug-free C++ code may run on a trusted system. A likely duty for this file server would be to process requests from remote and untrusted clients. Hence, by sending a specially crafted network packet to the file server, an attacker may be able to trigger and exploit a memory error in the server. Another example for an adversarial setting like this is a privileged native application (the server) running locally on a trusted Linux system. An unprivileged user may attempt to elevate her privilege by passing malicious inputs over the command line (the client) that "smash" the stack [10] of the privileged application. Probably one of today's most relevant settings in practice is the case of a user's trusted web browser (e. g., Microsoft Internet

Explorer or Google Chrome) that processes possibly malicious content from a multitude of untrusted web servers. Very similar and likewise practically relevant is the case of a user's trusted document viewer that is prone to memory errors (e. g., Adobe Reader or Microsoft Word) and that processes documents from attacker-controlled sources, e. g., an e-mail attachment.

## 2.2 Background

What exactly constitutes a memory error and what measures are taken to exploit and to mitigate such errors in the Classic setting is developed in this section. In particular, we describe how the much-discussed class of code-reuse attacks emerged from control-flow hijacking attacks. We give an overview of current offensive and defensive approaches and introduce work related to ours alongside.

### 2.2.1 From Memory Errors to Control-Flow Hijacking

Abstractly, a memory error is at hand when a pointer is dereferenced—for reading or writing—while it is not pointing to its designated item in memory. Whereby one speaks of a *spatial* memory error when a pointer is dereferenced after it has gone out-of-bounds. In turn, one speaks of a *temporal* memory error when a pointer is dereferenced *(i)* before the corresponding item in memory has been initialized or *(ii)* after the item has been disposed. In this context, the term "dangling pointer" is also often used to describe the state of a pointer before/after the lifetime of its referenced item in memory has started/ended.

An example for a spatial memory corruption error is the infamous buffer overflow. Buffer overflow vulnerabilities can still be found in considerable quantities in software and are notoriously exploited by attackers to hijack control flow, e. g., by overwriting a *code pointer* that is later used as target in an indirect control-flow transfer. Code pointers reference executable memory and are used by software to dynamically dispatch control flow at runtime. Generally, *function pointers* and *return addresses* are the most common types of code pointers.

Classically, attackers used to overwrite a return address in order to hijack a program's control flow [10]. Return addresses were easy and obvious targets, because they are stored in functions' stack frames on virtually all relevant modern processor architectures. As such, every stack-based buffer overflow is guaranteed to hit a return address at a certain offset. (And every return address is guaranteed to be the target of an indirect control-flow transfer when the corresponding function *returns*.)

However, due to the wide adoption of the stack canary mechanism [57] in C/C++ compilers, return addresses often cannot be hijacked directly anymore through buffer overflows. Hence, attackers today often also exploit other forms of spatial memory corruptions, e. g., heap-based buffer overflows or indexing bugs [205] where the index into an array can be controlled by the attacker and thus precise out-of-bounds reads or writes relative to the array's head can be performed.

Besides, the exploitation of temporal memory errors has also become commonplace. Attackers here in particular abuse so called *use-after-free* conditions where data is erroneously read from a memory item that has previously been deleted. Use-after-free errors

are prevalent even in modern industrial-grade C/C++ software. For example, recently Qu and Lu discovered dozens of possibly exploitable use-after-free vulnerabilities in Internet Explorer [165] using a relatively simple form of blackbox fuzz testing [89]. In order to exploit a use-after-free condition, an attacker needs to inject her own data to the memory location that the respective dangling pointer is pointing to. Depending on the context, there may be different ways to achieve this, e. g., using variations of the so-called *heap spraying* [168] or *heap feng shui* [196] techniques.

Generally, to hijack a program's control flow, attackers today often aim at (spatially or temporarily) corrupting C++ objects. To understand why, some background on C++ is necessary, which is provided next. This background is in particular important for the later discussions in Section 2.5.

### 2.2.2 Control-Flow Hijacking by Corrupting C++ Objects

In C++ and other object-oriented programming languages, programmers define custom types called *classes*. Abstractly, a class is composed of a set of member data fields and member functions [201]. A concrete instantiation of a class at runtime is called *object*.

*Inheritance* and *polymorphism* are integral concepts of the object-oriented programming paradigm: new classes can be derived from one or multiple existing ones, *inheriting* at least all visible data fields and functions from their *base classes*. Hence, in the general case, an object can be accessed as instance of its actual class or as instance of any of its (immediate and mediate) base classes. In C++, it is possible to define a member function as *virtual*. The implementation of an inherited virtual function may be overridden in a derived class. Invoking a virtual function on an object always invokes the specific implementation of the object's class even if the object was accessed as instance of one of its base classes. This is referred to as *polymorphism*.

Figure 2.1 gives a simple example of C++ inheritance involving the class `Base` and the thereof derived classes `A` and `B`. `Base` defines the (purely) virtual function `func()` which is implemented differently by `A` and `B`. The function `print()`, depicted in the top right of Figure 2.1, invokes either `A::func()` or `B::func()` depending on the dynamic type of its argument `Base* obj`.

**The Virtual Function Pointer Table** C++ compilers implement virtual function calls (vcalls) with the help of virtual function pointer tables (vtables). A vtable is an array of pointers to all, possibly inherited, virtual functions of a class. Hence, a static pointer exists to each virtual function in a C++ program. (This aspect becomes important in Section 2.5.) In the following we also use the term *address-taken* to refer to the circumstance that a static pointer exists to a function. For brevity, we do not consider the case of *multiple inheritance* here.

Every object of a class with at least one virtual function contains a pointer to the corresponding vtable at its very beginning (offset +0). This pointer is called *vptr*. For instance on Windows x86-64, a vcall is typically translated by a compiler to an instruction sequence similar to the following:

```
mov     rdx, qword ptr [rcx]
call    qword ptr [rdx+8]
```

```
class Base {
protected:
  size_t x;
public:
  virtual size_t func() = 0;
  virtual ~Base();
};

class A : public Base {
public:
  size_t func() { return x; }
  ~A() { /* ... */ }
};

class B : public Base {
public:
  size_t func() { return 2 * x; }
  ~B() { /* ... */ }
};
```

```
void print(Base* obj) {
  std::cout << obj->func();
}

int main(int argc, char** argv)
{
  A* objA = new A;
  print(objA);
}
```

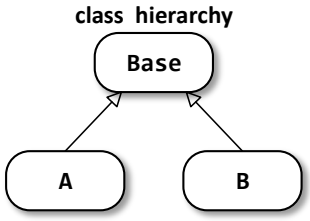


Figure 2.1: Simple C++ inheritance and polymorphism example; the highlighted virtual function invocation in `print()` dynamically dispatches to `A::func()` or `B::func()` at runtime.

Here, `rcx` is the object's `this` pointer—also referred to as *this-ptr* in the remainder of this work. First, the object's vptr is temporarily loaded from offset +0 from the this-ptr to `rdx`. Next, in the given example, the second entry in the object's vtable is called by dereferencing `rdx+8`. Compilers generally fix the index into a vtable at a vcall site. Accordingly, this particular vcall site always invokes the second entry of the vtable referenced by the given object's vptr.

#### 2.2.2.1 Vtable Hijacking

Vtables are composed of code pointers and C++ programs typically contain many of them. Accordingly, vtables may on first glance appear as premier targets for corruption in control-flow hijacking attacks. However, as vtables are static data structures, they are almost always stored in fixed read-only memory (e. g., in the `.rdata` section), which makes direct corruptions infeasible. This is naturally different for vptrs, which are used to dynamically reference vtables. As such, it has become common practice for attackers to corrupt a vptr such that the next vcall on the corresponding object leads to a code location of their choice. This kind of attack is also referred to as *vtable hijacking* [235].

For multiple examples of vtable hijacking attacks, consider Figure 2.2. It depicts the order (①, ②, and ③) in which data and code pointers are dereferenced during the dispatching of the virtual function invocation `obj->func()` from Figure 2.1. In case an attacker is able to corrupt the stack or the heap, she can modify the data pointers ① (the object pointer on the stack) or ② (the object's vptr on the heap) in different ways in order to hijack the control flow as indicated by the dashed arrows in Figure 2.2, for example:

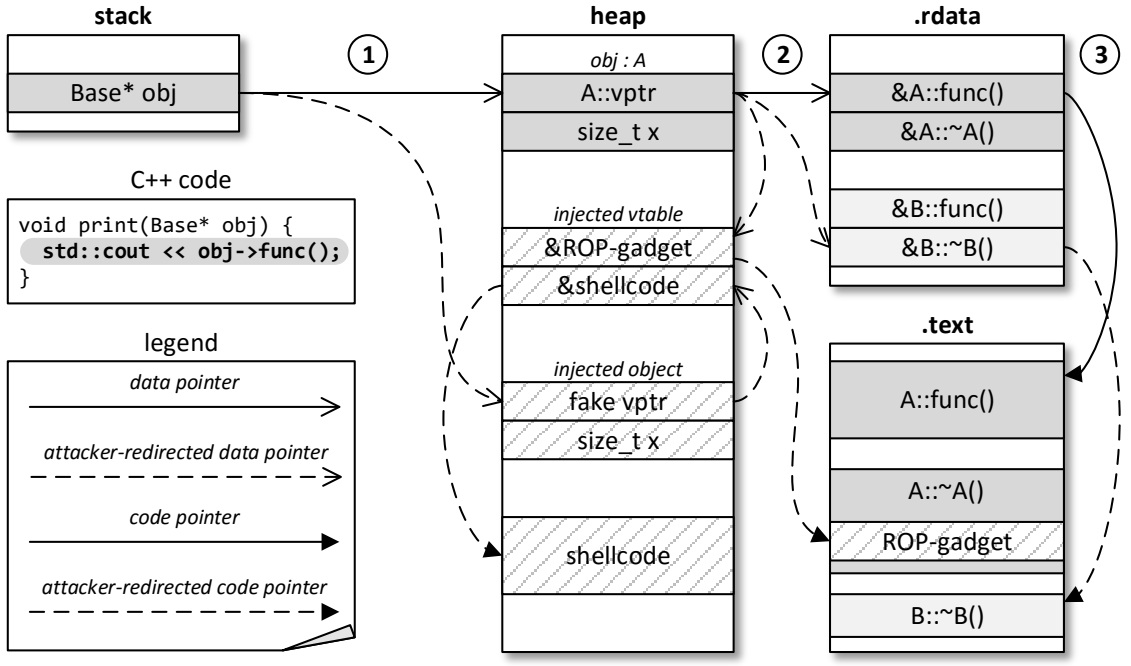


Figure 2.2: Exemplary sequence of pointer dereferences in a C++ virtual function invocation; dashed arrows indicate ways to perform vtable hijacking attacks via corruptions of the stack or the heap.

- The object's vptr can be redirected to a different existing vtable—here the vtable of class `B`—such that a completely different virtual function is invoked than intended; in this case `B::~B()` instead of `A::func()`. This form of vtable hijacking is also known as *vtable reuse* attack [235].
- The object's vptr can be redirected to a vtable injected by the attacker to the heap. Consequently, an attacker-controlled code pointer is invoked. This kind of vtable hijacking attack is also referred to as *vtable injection* [235]. In a vtable injection attack, the attacker typically directs the hijacked control flow to her injected *shellcode* or to the first gadget of a *ROP chain* (both explained next in Section 2.2.3).
- The object pointer on the stack can be redirected to an object injected by the attacker. Subsequently, a *vtable injection* or *vtable reuse* attack can be conducted as above.

We note that many of today's real world control-flow hijacking attacks against C++ applications (e. g., [51, 111, 217]) employ forms of vtable hijacking.

### 2.2.3 From Control-Flow Hijacking to Code-Reuse Attacks

Up to this point, we described ways for an attacker to hijack control flow. In the past, attackers typically immediately redirected hijacked control flow to their own malicious

code—also often referred to as *shellcode*—which they injected to the stack or the heap (*code injection attack*). This changed with the integration of the *execute disable bit* feature [107]—also known as NX bit—in the x86 and other processor architectures. With this hardware feature, it became possible to mark writable memory pages as non-executable. In the consequence, the concept of *data execution prevention* (DEP)—also known as *write XOR execute* (W⊕X)—has been widely implemented in operating systems, e. g., in Windows and Linux. In the presence of DEP, memory pages that are meant to hold data are per default marked as non-executable. As such, typically only an application's `.text` section resides in executable memory, while the stack, the heap, and sections such as `.data` remain non-executable. A common exception are so called just-in-time (JIT) compilers, which dynamically emit and execute machine code. For instance, modern browsers make heavy use of JIT compilers.

Thanks to the prevalence of DEP, the immediate injection and execution of shellcode is in many attack scenarios infeasible today. Accordingly, the initial control-flow hijacking is today typically followed by a more sophisticated *code-reuse attack*. Code-reuse attacks emerged as a direct response to DEP and are immensely popular among attackers today. The basic idea is to induce malicious program behavior by misusing existing code chunks in the target program's address space. While different code-reuse techniques have been shown to offer Turing-complete semantics[1], code reuse is in practice often only used to pave the way for a traditional code injection attack by making attacker-controlled memory executable. For example on Windows, the attacker's goal typically is to invoke the Windows API (WinAPI) function `VirtualProtect()`, which allows to change access permissions for memory pages.

### 2.2.4 Code-Reuse Attack Techniques

As a basis for the following discussions, we now briefly introduce the most common forms of malicious code reuse.

#### 2.2.4.1 Return-into-libc

The first documented code-reuse attack technique was *return-into-libc* (RILC) [71]. Classic RILC is geared towards x86-32 and the cdecl [136] calling convention (and it also neglects stack canaries): a stack-based buffer overflow is used to overwrite a return address such that the vulnerable function "returns" to the entry of an attacker-chosen function, e. g., a sensitive library function like `system()` in the eponymous libc. Along with the fake return address, typically also fake arguments are written to the stack, such that in effect an attacker-chosen function is invoked on attacker-chosen arguments. An extended form of RILC [144] allows to chain multiple attacker-chosen function invocations: multiple fake return addresses, alongside fake arguments, are written to the stack such that one attacker-chosen function returns to the next. Depending on calling conventions and other circumstances, it may be necessary to adjust the stack pointer (`esp` on x86-32) between the invocations of two functions in such a RILC attack. For this, fake return addresses

[1] In essence, *Turing-complete* in this context means that an attacker can induce arbitrary malicious computations.

are injected that do not reference whole functions but rather existing short instruction sequences that manipulate the stack pointer and end in a return instruction (`retn` on x86-32). An example for such an instruction sequence is the following, which effectively removes 16 bytes from the stack before it returns:

```
add     esp, 10h
retn
```

RILC has been shown to be Turing-complete when multiple chosen functions from the standard libc can be invoked [212]. It was later showed that the same can also be achieved for other common libraries [191].

#### 2.2.4.2 Return-oriented Programming

Probably the presently most widely used code-reuse attack technique is *return-oriented programming* (ROP) [188]. ROP can be considered a generalization of the older RILC approach. In fact, the former directly emerged from the latter [119, 144]: in a ROP attack, as in RILC, fake return addresses are written to the stack. Like in RILC, these reference whole functions or short instruction sequences ending in a return. However, these instruction sequences—often called *gadgets*—play a much bigger role in ROP. Not only are gadgets used to adjust the stack pointer as in RILC but also to load, to store, or to modify other registers.

In ROP, the attacker "programs" (hence the name of the technique) the desired malicious semantics by chaining gadgets. In such a *ROP chain*, one gadget returns to the next and each gadget performs a specific primitive operation, but may also have certain unwanted side effects that need to be compensated. Typically, gadgets work relative to the stack pointer, e. g., a common type of gadget pops a value from the stack into a register. Hence, the attacker usually interleaves return addresses with data on the hijacked stack. Typically, at least on x86, suitable gadgets for ROP attacks exist in sufficient quantities in most non-trivial programs [75,102]. However, ROP attacks have been demonstrated on a wide range of platforms including more exotic ones like SPARC [169] and Z80 [49]. ROP has been shown to be Turing-complete given a certain set of common gadget types [169] and there are also compilers [104,182,194] that automatically convert a given shellcode into a target program-specific ROP chain.

**Example ROP Chain** Figure 2.3 visualizes the execution of an x86-64 ROP chain which implements the simple summation of the integers 5 and 6: the chain is composed of the gadgets ❶, ❷, and ❸ (in this order). Accordingly, the hijacked stack contains a corresponding fake return address for each. The hijacked control flow reaches gadget ❶ first. The gadget pops the attacker-chosen value 1 from the stack into the register `rax`. The `retn` instruction in gadget ❶ pops the next fake return address from the stack and branches to gadget ❷. This gadget pops the value 2 into `rdx`. As an unwanted side effect, the gadget also pops a value into `r8`. To compensate this, the hijacked stack contains a *dummy value*. Subsequently, gadget ❷ returns to gadget ❸, whose purpose is to add `rax` to `rdx`. As side effects, the gadget writes to the 32-bit register `eax` and to the memory pointed to by `rcx`. Accordingly, in order to prevent the target program from crashing, the

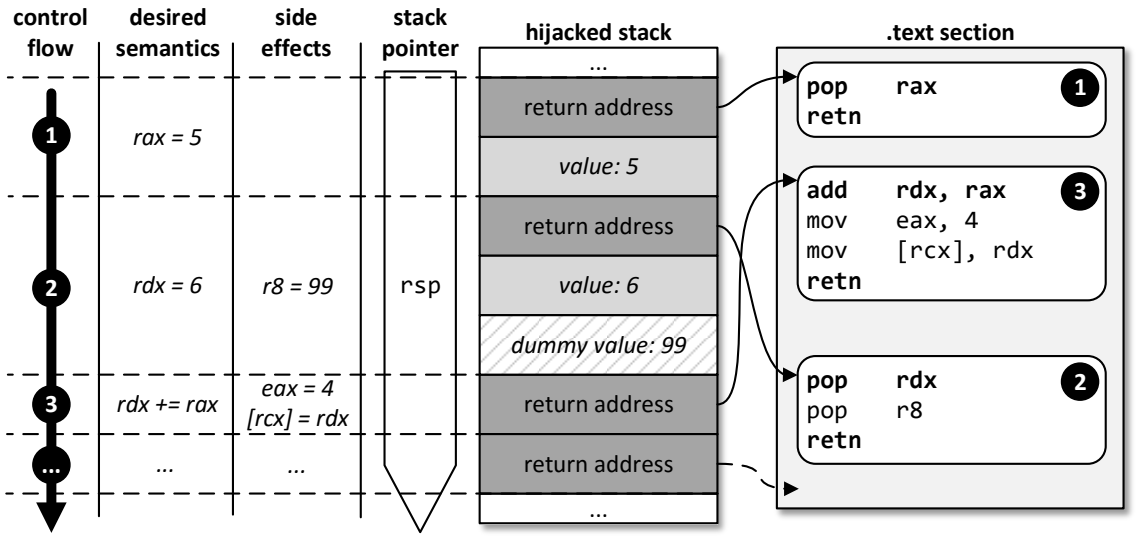


Figure 2.3: Visualization of the execution of a simple x86-64 ROP chain composed of three gadgets; the hijacked stack is composed of return addresses and data injected by the attacker.

attacker would need to make sure that `rcx` points to writable memory. After the execution of gadget ❸, the control flow reaches the next gadget in the chain and `rdx` holds the final result $5 + 6 = 11$.

**Unaligned Gadgets** On x86, gadgets can be *aligned* as well as *unaligned* with the original instruction stream produced by the target program's compiler. This is because on x86, other than on RISC architectures, instructions may start at any offset into a code page. Hence, any byte in executable memory of the value `C3`, which corresponds the `retn` instruction, may be the ending of one or multiple useful gadgets on x86. In the consequence, code on x86 tends to exhibit a higher frequency of useful gadgets than code on RISC architectures. For example, the x86 instruction `setz bl` is encoded through the bytes `0F 94 C3`, whereby the last two bytes `94 C3` taken for themselves in turn encode the following useful two-instruction gadget:

```
xchg    eax, esp
retn
```

**Pivoting of the Stack Pointer** On x86-32, this particular gadget can be used to swap the values of the register `eax` and the stack pointer `esp`. (On x86-64, the gadget only swaps the lower 32-bit of the stack pointer.) In cases where the attacker cannot overwrite the stack directly, it is common practice to initially direct the hijacked control flow to a gadget like this in order pivot the stack pointer to a fake stack in attacker-controlled memory, e. g., on the heap. Of course, for this to work for the given gadget, `eax` would need to point to the attacker-controlled memory.

**Characteristics** Besides unaligned gadgets (x86-only) and the hijacking of the stack, another revealing characteristic of a classic ROP attack is that instruction sequences of gadgets are often not preceded by call instructions. As such, during the course of a ROP attack, typically the execution of (many) returns can be observed that do not lead to *call-preceded* code locations (but to the beginning of gadgets). Control-flow transfers of this kind can, with certain exceptions, never be observed during the execution of benign code, because a regular return always leads directly behind the call instruction that invoked the corresponding function.

#### 2.2.4.3 Other Techniques

A closely related technique to ROP is *jump-oriented programming* (JOP). In JOP, gadgets end in indirect jumps and calls rather than return instructions [35, 48]. In basic JOP, return instructions are emulated by using a combination of a *pop-jmp* pair. However, in contrast to ROP, attacker-injected code pointers to gadgets do not necessarily need to reside on the stack in JOP. In particular, an "update-load-branch" sequence with general purpose registers can be used to load the next-in-line code pointer from an arbitrary memory location [48]. In general, JOP can offer the same expressiveness as ROP to an attacker: practical Turing-complete gadget sets for JOP have been presented for ARM and x86-32 [35, 48]. The term *call-oriented programming* (COP) is sometimes used to refer to ROP-derived techniques that employ indirect calls instead of returns (as in ROP) or indirect jumps (as in JOP). [47, 90].

Finally, *sigreturn oriented programming* (SROP) [37], is a distinct code-reuse attack approach that misuses UNIX signals. SROP is Turing-complete and in contrast to ROP does not chain short chunks of instructions sequences. In SROP, the UNIX system call *sigreturn* is repeatedly invoked on an attacker supplied *signal frames* lying on the stack. Accordingly, as prerequisites, the attacker needs to control the stack and needs to be able to divert the control flow such that *sigreturn* is invoked. SROP was not specifically designed to circumvent modern protection techniques, but rather as an easy-to-use and portable alternative to ROP and for implementing stealthy backdoors.

### 2.2.5 Defenses against Code-Reuse Attacks

Code-reuse attacks are very powerful and preventing them in a reliable yet cost-effective manner is an ongoing struggle in academia and industry. In this section, we give an initial overview of a range of defensive approaches that have been proposed (and partly also practically deployed) to tackle code reuse or, more generically, control-flow hijacking. Specifically, we consider four different classes of defenses in the following:

- Memory safety
- Control-flow integrity
- Heuristics-based runtime detection
- Hiding, shuffling, or rewriting of code or data

While this classification is not necessarily exhaustive and certainly not strict, we think that it decently captures the landscape of contemporary defenses.

#### 2.2.5.1 Memory Safety

Conceptually, code-reuse attacks can generically be prevented if the initial control-flow hijacking that necessarily precedes them is thwarted [205]. It follows that code-reuse attacks cannot be mounted if critical *spatial* memory errors like buffer overflows and *temporal* memory errors like use-after-free conditions are prevented in the first place (and high-level logical errors in programs are neglected).

A range of techniques has been proposed that provide forms of spatial memory safety [8, 9, 142], forms of temporal memory safety [7, 143], or both [52, 122, 187]. Unfortunately, for precise and comprehensive guarantees, these techniques typically require access or even changes to a program's source code, may be incompatible with existing binary libraries, and may incur considerable overhead. This hampers their broader deployment [205].

For example, SoftBound is a "[...] compile time transformation for enforcing complete spatial safety of C" [142] and the complimentary CETS is a "[...] compile-time transformation for detecting all violations of temporal safety in C programs" [143]. On the baseline, SoftBound and CETS maintain separately stored metadata for each pointer in a program at runtime. Every time a pointer is to be dereferenced, its metadata is consulted to check if the dereferencing is safe in spatial or temporal terms respectively. In conjunction, SoftBound and CETS guarantee full memory safety for C programs. On the downside, the overhead for combined SoftBound and CETS is on average above 100% for popular benchmarks [143]. Moreover, they naturally require to-be-protected software to be recompiled and offer only limited compatibility with existing binary software.

#### 2.2.5.2 Control-Flow Integrity

An orthogonal concept to memory safety is *control-flow integrity* (CFI) [1]. CFI does not hinder control-flow hijacking as such but rather aims at containing its effects by preventing "unexpected" control-flow transfers as they can typically be observed in code-reuse attacks. More formally, CFI enforces a program's control-flow to adhere to a certain control-flow graph (CFG). The enforcement of CFI is a two-step process:

1. determination of a program's approximate CFG $X'$.
2. instrumentation of a subset of the program's indirect branches (i. e., call, jump, and return instructions) with runtime checks that enforce the control flow to be compliant with $X'$.

CFI checks are often implemented by assigning IDs to all possible indirect branch locations in a program. At runtime, a check before each indirect branch validates if the target ID is in compliance with $X'$ for the given indirect branch.

The approximate CFG $X'$ can be determined statically or dynamically; on source code or on binary code. In any case, $X'$ should be a supergraph of the intrinsic CFG $X$ encoded in the original source code of a program. Given a precise CFI policy, if $X'$ is equal to $X$, it is impossible for an attacker to divert control flow in any way that is not conform

to the semantics of a program's source code [2]. However, in practice, this can be hard to achieve as it requires precise *points-to analysis* for all pointers influencing a program's intrinsic CFG $X$.

Similar to comprehensive memory safety techniques, there are practical obstacles like overhead or required access to source code that hinder the broader deployment of precise CFI. As such, more practical but only partially precise CFI solutions have been designed that for example focus on securing C++ virtual function invocations [110, 211] or on enforcing the integrity of returns using a *shadow call stack* [1, 66, 82].

**Imprecise Control-Flow Integrity** Moreover exists a range of imprecise CFI solutions [1, 64, 85, 145, 156, 163, 206, 216, 235, 236, 238] that often only require access to an program's binary code to protect it. It is characteristic for these imprecise solutions to differentiate between only a few branch location IDs. In one of the simplest cases [1], a CFI policy with only two IDs is enforced: one ID is assigned to all address-taken code locations and the other ID is assigned to all call-preceded code locations. At runtime it is then enforced that the former can only be reached by indirect calls/jumps and the latter only by returns. This way, among others, rogue returns that are characteristic for ROP and RILC are prevented.

A metric used for rating the effectiveness of imprecise CFI solutions is the *average indirect target reduction* (AIR). AIR "[...] quantifies the fraction of possible indirect targets eliminated by a CFI technique" [238]. For example, an AIR score of 99.13% was measured when the described simple two-ID CFI policy was applied to the SPEC CPU2006 benchmark [238]; meaning in essence that on average, for every existing indirect branch in the program, an attacker can only choose from .87% of the available branch destinations in the unprotected version of the program. However, advanced ROP-based attacks [65, 90] have been demonstrated that still bypass certain imprecise CFI solutions [1, 236, 238].

#### 2.2.5.3 Heuristics-based Runtime Detection

Related to and partly also inspired by CFI, a group of defenses exists [54, 83, 152, 228, 239] that aim to prevent ROP and other code-reuse techniques through runtime detection heuristics: on certain triggers, e. g., on the execution of a sensitive library function, these defenses examine the recent (and in some cases also the expected future) control flow of the protected program for abnormal branches. The control flow is either obtained/approximated using hardware debugging features of contemporary processors or through forms of software emulation. Commonly employed detection heuristics include:

- returns to not call-preceded code locations (indicator for RILC and ROP)
- a high frequency of indirect branches (indicator for code reuse in general)

The former detection heuristic is comparable to an imprecise CFI policy where all call-preceded code locations are assigned the same ID and can be reached from all return instructions.

Naturally, heuristics-based defenses can only offer intuitive arguments or empirical evidence for their security. However, they are usually simple to deploy, widely applicable, and incur relatively little overhead; characteristically they require none or only minimal

rewriting of the binary code of a to-be-protected application. As such, it comes as no surprise that the heuristics-based ROPGuard approach [83] has been widely deployed as part of Microsoft's Enhanced Mitigation Experience Toolkit (EMET) [134] for Windows. Hence, ROPGuard is possibly one of the most prevalent code-reuse defenses in practice. In turn, it also comes as no surprise that probably most heuristics-based defenses, including in particular ROPGuard/EMET, have been shown to be vulnerable to variants of ROP [47, 65, 70, 91, 161] in different scenarios.

#### 2.2.5.4 Hiding, Shuffling, or Rewriting of Code or Data

Lastly, a category of defenses exists that do not at all attempt to hinder the attacker from manipulating a program's control flow. Instead, the idea is to impede malicious code reuse by hiding [31, 209], shuffling [221], or rewriting [151] a program's code or data, often in a pseudo-random manner. There are also various hybrid approaches [22, 58, 59].

Undoubtedly, the most prominent example here is the *address space layout randomization* (ASLR) [31, 209] technique that is almost ubiquitous on modern operating systems. ASLR ensures that the stack, the heap, and executable modules of a program are mapped at secret, pseudo-randomly chosen memory locations. This way, among others, the whereabouts of useful code chunks are concealed from an attacker. Unfortunately, it is sufficient for an attacker to disclose a single code pointer to locate the executable code segment of a program protected by ASLR.

To tackle this problem, finer-grained randomization approaches have been proposed. For example, the STIR system [221] pseudo-randomly shuffles a program's basic blocks on each start-up. However, even finer-grained code randomization defenses can still be circumvented when the attacker is able to (repeatedly) disclose memory contents by exploiting an *information leak* vulnerability in the protected program [194]. Information leaks can often be crafted from conventional memory corruption errors [205].

## 2.3 Research Motivation and Contributions

Our observation is that while provably secure defenses against code-reuse attacks exist (e. g., precise CFI or full memory safety), they are typically expensive or difficult to deploy. This is why the design of "good enough" defenses with acceptable costs and moderate requirements is still a busy field of research. Unfortunately, arguments for the security of such defenses are often informal or based on limited empirical evidence. Typical evidence here includes, for instance, that an imprecise CFI solution has a seemingly high AIR score (see Section 2.2.5.2) or that experiments show that a defense can prevent a certain subset of the 850 attacks from the RIPE testbed [224] or other existing attacks. The central thesis of this chapter is that evidence if this kind is merely a *necessary* but not a *sufficient* criterion for the effectiveness of a defense. In other words: we aim to show that from the effectiveness against certain existing attacks, effectiveness against (slightly) adapted or new attack forms does not necessarily follow. To this end, we describe several advanced code-reuse attack techniques in this chapter. These techniques break with common assumptions that many defenses across all four introduced categories are built on. Consequently, they expose fundamental deficiencies in existing academic as well as commercial defensive approaches

that were assumed to be "good enough". We believe that these results are important contributions, which will be helpful for the design and implementation of future, more secure defenses against code-reuse attacks.

**Variants of Return-oriented Programming** We begin in Section 2.4 with the evaluation of the effectiveness of three heuristics-based defenses (kBouncer [152], ROPecker [54], and ROPGuard [83]) which attempt to detect ROP-based attacks at runtime. We observe that all three defenses rely on certain assumptions regarding ROP that do not necessarily hold in practice. To proof this, we present practical variants of ROP that break with these assumptions and consequently bypass the defenses in a generic manner.

We developed these ROP variants concurrently to three other closely related scientific works, which were all published at the *Usenix Security Symposium 2014*: Carlini and Wagner [47] demonstrated practical ROP-based bypasses for kBouncer and ROPecker. Their approaches are closely related to ours. In particular, they also describe and demonstrate a technique similar to our "LBR flushing" technique that is presented in Section 2.4.1.2 to bypass kBouncer. Göktaş et al. [91] also demonstrated bypasses for the same two defenses. In their approach "heuristics breaker" gadgets are periodically mixed into ROP chains. This is similar to the technique we use to bypass ROPecker. Finally, Davi et al. [65] presented a Turing-complete gadget set contained in the standard Windows system library kernel32.dll. This gadget set can be used to create arbitrary ROP chains that go unnoticed by kBouncer, ROPecker, and ROPGuard and also a certain imprecise CFI solution for binary code [238]. Still, unique to our work is that

- we do not only present attacks against the kBouncer concept on x86-32 but also on x86-64,
- we show that all required gadgets for our attack against kBouncer on x86-32 can already be found in a minimal "hello world" C/C++ application, and
- we conducted our own experimental false-positive analysis for kBouncer and ROPecker using independently developed emulators.

**Counterfeit Object-oriented Programming** Next in this chapter, in Section 2.5, we present *counterfeit object-oriented programming* (COOP). COOP is a novel code-reuse attack technique against C++ applications. In some ways, COOP can be seen as a generalization of the loop-based 64-bit ROP attack against kBouncer that is described in Section 2.4.1.4. However, COOP is a new form of code-reuse attack and not a variant of ROP. COOP employs forms of *vtable reuse* (see Section 2.2.2.1) and is related to RILC inasmuch as that only whole functions are reused.

With COOP we demonstrate the limitations of defenses from all four categories in the context of C++. We show that it is essential for code-reuse defenses to consider C++ semantics like the class hierarchy carefully and precisely. As recovering these semantics without access to source code can be challenging or sometimes even impossible, our results here demand for a rethinking in the assessment of binary-only defenses and make a point for the deployment of precise source code-based defenses where possible.

In particular, our observation is that COOP circumvents virtually all CFI solutions that are not aware of C++ semantics. Moreover, we also observe that various defenses from other categories that do not consider these semantics *precisely* to be prone to COOP. In fact, we show that even several recently proposed defenses against control-flow hijacking or code-reuse attacks that specifically target C++ programs (CPS [122], T-VIP [85], vfGuard [163], and VTint [235]) offer at most partial protection against COOP, and we can successfully bypass all of them in realistic attack scenarios. We also discuss how COOP can reliably be prevented by precise C++-aware CFI, defenses that provide (spatial and temporal) integrity for C++ objects, or defenses that prevent certain common memory disclosures, e. g., through the *shuffling* of vtables.

We demonstrate the practical relevance of COOP by implementing working low-overhead exploits for real-world vulnerabilities in Microsoft Internet Explorer 10 (32-bit and 64-bit) on Windows and Chromium 41 on Linux x86-64. To launch our attacks against modern applications, we inspected and identified easy-to-use gadgets in a set of well-known Windows system libraries—among them the standard Microsoft Visual C/C++ runtime that is dynamically linked to many applications—using basic symbolic execution techniques. We also show that COOP is *Turing-complete* under realistic conditions.

## 2.4 Challenging Heuristics-based Defenses with Advanced ROP

We begin the discussions in this section with the analysis of kBouncer, a defensive mechanism that aims at detecting and preventing ROP-based attacks against user mode applications on the Windows operating system. kBouncer leverages the *last branch recording* (LBR) feature incorporated in current AMD and Intel x86-64 processors [4, 107] to check for suspicious control flows. kBouncer received broad attention not only from the research community when its first version [150] was announced as the $200,000 winner of the Microsoft BlueHat Prize [36]. We show, theoretically and experimentally, that kBouncer's latest version [152] can be circumvented in virtually all realistic 32-bit and 64-bit attack scenarios with little extra effort. More specifically, we demonstrate how three recent ROP-based exploits—e. g., for Microsoft Internet Explorer on Windows 8—can be modified to bypass kBouncer. Furthermore, we show that even the `.text` section of a minimal 32-bit C/C++ application compiled with Microsoft's Visual Studio contains all necessary gadgets required to bypass kBouncer. We also discuss why the kBouncer concept is prone to high rates of false-positive detections in practice. Subsequently, in Section 2.4.2, we demonstrate how successful attacks against kBouncer in practice often also circumvent ROPGuard. The ROPGuard defensive approach placed second at the BlueHat Prize and has since been incorporated into Microsoft's EMET. Section 2.4.3 closes the in-depth discussion of heuristics-based defenses with the analysis of ROPecker [54]. ROPecker was presented in 2014 and also leverages the LBR feature to protect applications on Linux from ROP-based attacks. We show that ROPecker suffers from conceptual weaknesses similar to kBouncer. In its published form, ROPecker can be circumvented in a generic way by an adversary. We empirically verify our attack and demonstrate a successful low-overhead bypass for a recent vulnerability of the popular web server software Nginx. Further, we analyze ROPecker's susceptibility to false-positive detections.

**Last Branch Recording** Both kBouncer and ROPecker rely on the LBR feature to examine an application's past control flow on certain events. The LBR can only be enabled and accessed from kernel mode. It can be configured to only track certain types of branches. Both kBouncer and ROPecker utilize this feature and they limit the LBR to indirect branches in user mode. For each recorded branch, an entry containing the *start* and *destination* address is written to the corresponding processor core's *LBR stack*. In Intel's latest Haswell architecture, an LBR stack is limited to only 16 entries. For each newly recorded branch, the oldest entry in an LBR stack is overwritten. At any given time, an LBR stack may not only contain entries from a single process/thread, but from multiple ones running on the same core [54]. In the following, we do not consider this effect, though, it might in practice facilitate attacks. Instead, for simplicity, we assume that the LBR stack is always saved/restored on context switches.

### 2.4.1 Security Assessment of kBouncer

The latest version of the kBouncer runtime ROP exploit mitigation approach was presented by Pappas et al. in 2013 [152]. kBouncer checks for suspicious branch sequences hinting at a ROP exploit whenever a Windows API (WinAPI) [171] function considered as possibly harmful is invoked in a monitored process. kBouncer's authors list 52 WinAPI functions which they consider as possibly harmful. Among these functions are for example `VirtualAlloc()` and `VirtualProtect()` that are notoriously abused by attackers. Pappas et al. acknowledge that the list is possibly not complete and could be extended in the future.

kBouncer is composed of a user mode component and a kernel driver. The user mode component hooks all to-be-protected WinAPI functions in a monitored process. Whenever the control flow reaches one of these hooks, the kernel driver is informed via the WinAPI function `DeviceIoControl()`. Subsequently, the driver examines the LBR stack for traces of a ROP chain. Since kBouncer's user mode component uses two indirect branches to inform the driver, only 14 of the LBR stack's 16 entries are of value to the driver's ROP detection logic [152]. In case no attack is detected, the driver saves a corresponding "checkpoint" in kernel memory for the respective thread. Whenever a system call corresponding to a hooked WinAPI function is invoked, the driver consumes the matching checkpoint; if none is found, an attack is reported. According to Pappas et al., the purpose of the checkpoint system is to prevent exploit code from simply skipping over the top-level WinAPI functions and calling similar lower level functions (e. g., `NtCreateFile()` instead of `CreateFileW()`). The reason for kBouncer not monitoring system calls directly is the observation that between WinAPI functions' and their corresponding system call often many legitimate indirect branches are executed that would often overwrite traces of ROP chains in the LBR stack [152].

In order to evaluate kBouncer's practical applicability and defensive strength, we created a standalone emulator for kBouncer based on certain pieces of source code generously provided to us by Pappas et al. The emulator uses the *Pin* [127] dynamic analysis framework to instrument monitored applications at runtime. To the best of our judgment, the emulator accurately captures all of kBouncer's core concepts as described by Pappas et al. [152].

```
int factorial (int n) {
 if (n <= 1) return 1;
 return factorial(n-1)*n;
}
```

Listing 2.1: Recursive C function that calculates the factorial of an integer

```
[...]
lea     ecx, [edi-1]
call    factorial
mul     edi
pop     edi
retn
```

Listing 2.2: Disassembly of epilogue of function `factorial()`

#### 2.4.1.1 Examination of Indirect Branch Sequences

When examining the LBR stack corresponding to the invocation of a WinAPI function, kBouncer's kernel driver assumes an attack if at least one of the following is encountered: *(i)* a *return* to an instruction not preceded by a `call` instruction or *(ii)* a chain of a certain number of *gadgets* ending in the latest LBR stack entry. For kBouncer, gadgets are up to 20 instructions long and may contain conditional or unconditional relative jumps [152]. In the following, we refer to gadgets under this definition as k-gadgets. Gadgets outside this definition are conversely denoted as non-k-gadgets.

**Gadget Chain Detection Threshold** The maximum gadget chain length kBouncer can identify is 13. This is due to only 14 LBR stack entries being of value to kBouncer's detection logic and the latest effective entry always corresponding to a branch to a WinAPI function [152].

In order to determine a suitable detection threshold for the length of gadget chains, Pappas et al. examined a set of popular Windows applications (e. g., Microsoft Word and Internet Explorer) at runtime while executing certain tasks [152]. They report on having found the LBR stack to contain chains of at most *five* k-gadgets on entry to any of the 52 possibly harmful WinAPI functions across their experiments. As a result, Pappas et al. defined kBouncer to consider chains of *eight* or more k-gadgets as harmful, leaving a security margin of three against false positives.

However, longer chains of k-gadgets can easily occur in practice in benign and unsuspicious control flows. Consider for example a simple recursive function calculating the factorial of an integer as shown in Listing 2.1 and Listing 2.2. After the termination of `factorial(n)`, the LBR stack contains a legitimate chain of $n - 1$ k-gadgets of the following form:

```
mul     edi
pop     edi
retn
```

This makes the control flow appear to contain a ROP chain under the kBouncer definition. Many other possible scenarios exists where legitimate control flow resembles a ROP chain under the kBouncer definition as well.

In fact, our kBouncer emulator detected k-gadget chains longer than the given detection threshold for all non-trivial applications we executed on Windows 7 SP1 64-bit while

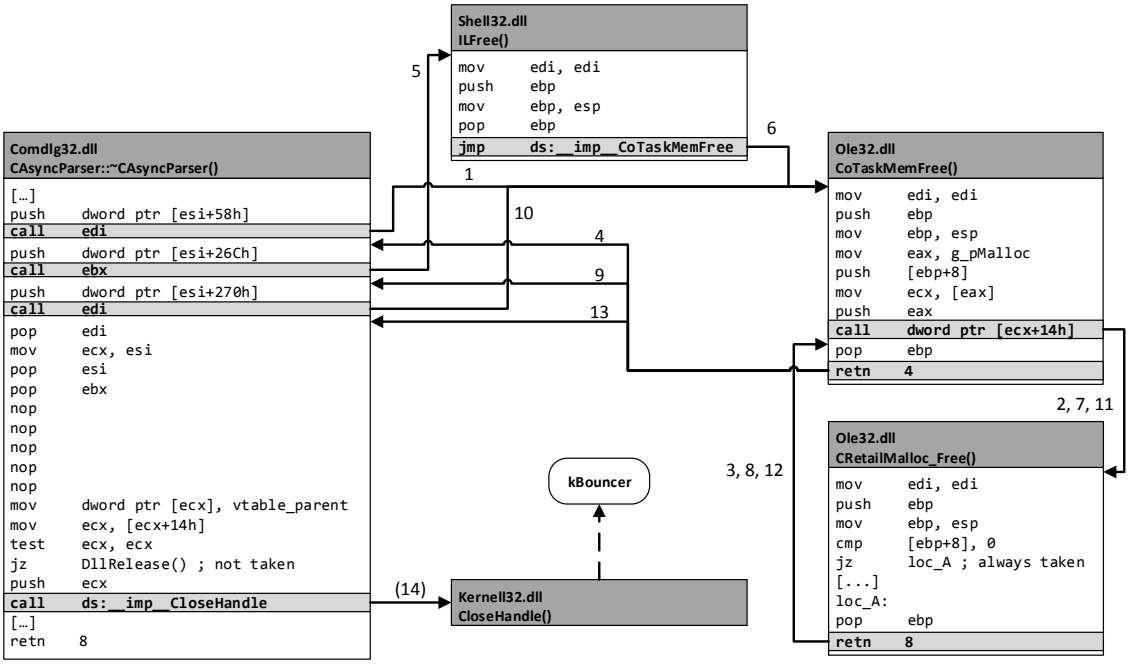


Figure 2.4: Exemplary false-positive chain of 13 k-gadgets as detected by our kBouncer emulator for the "Save File As" dialogue in Notepad++ 5.9.8 (32-bit) on Windows 7 64-bit. Taken indirect branches are highlighted in light gray. Branches are labeled according to the order they are executed.

monitoring the discussed 52 WinApi functions. For example, saving a text file using the popular editor Notepad++ 5.9.8 (32-bit) reliably resulted in one detected chain of the maximum length 13. The chain is depicted in Figure 2.4: the chain starts towards the end of the destructor of the class `CAsyncParser` in comdlg32.dll and spans over ole32.dll and shell32.dll before ending in the protected WinAPI function `CloseHandle()`. The characteristic of the chain is that several short functions are invoked in a nested manner using indirect calls only.

Note that the discrepancy in quality and quantity of false positives detected by our emulator and the original kBouncer could have many reasons. Possibly, the dynamic disassembly provided by Pin to our emulator is more comprehensive than the static disassembly available to kBouncer's *offline gadget extraction toolkit*. It is also very well possible that kBouncer employs certain additional filtering techniques in practice. Of course we can also not entirely rule out inaccurate assumptions on our side.

#### 2.4.1.2 Circumventing kBouncer

We now explore ways an *aware attacker* can follow to circumvent kBouncer. We consider kBouncer as bypassed when it is possible (with respect to the actual limits imposed by a vulnerability) to reliably and repeatedly conduct the following two consecutive steps without kBouncer noticing:

S1 execution of arbitrary ROP chain

S2 successful invocation of a WinAPI function protected by kBouncer

Obviously kBouncer can be safely bypassed if the last 14 indirect branches leading to a protected WinAPI function cannot be distinguished from benign control flow; regardless of the actually deployed gadget chain detection policy. This is due to kBouncer's driver being effectively only able to look at most 14 LBR stack entries into the past.

In view of this fact, Pappas et al. discuss the possibility of an attack based on a seemingly legitimate gadget chain (returns leading to call-preceded locations only and at least every eighth gadget being a non-k-gadget). They allude that such an attack would be difficult and state that "if evasion becomes an issue, longer gadgets could be considered during the gadget chaining analysis of an LBR snapshot" [152]. Furthermore, they also discuss the possibility of an attacker looking "[...] for a long-enough execution path that leads to the desired API call as part of the application's logic". They expect this kind of attack to be "[...] quite challenging, as in many cases the desired function might not be imported at all, and the path should end up with the appropriate register values and arguments to properly invoke the function".

We find that an attacker could instead also employ a simpler *third* method: the code executed between a ROP chain (step S1) and a protected WinAPI function (step S2) does not necessarily need to be *meaningful*; not in the context of the ROP chain and neither in the context of the attacked application. Hence, an attacker can simply execute arbitrary *meaningless* code between both steps in order to flush the LBR stack prior to the inspection through kBouncer's driver. The only requirements such *LBR-flushing code* has to fulfill are:

- Sufficiently many (e. g., 14) unsuspicious indirect branches must be executed.
- The arguments to the to-be-invoked WinAPI function must not be altered.
- Other WinAPI functions protected by kBouncer must not be invoked.
- The execution environment must not be rendered uncontrollable; e. g., by access violation exceptions or manipulation of the ROP chain on the stack.

In the following we *(i)* discuss suitable LBR-flushing code sequences and *(ii)* explain how attackers can generically circumvent kBouncer by incorporating them into ROP chains. Attacks for 32-bit and 64-bit environments are discussed separately as they require slightly different approaches due to divergent default calling conventions: in 32-bit applications, arguments to WinAPI functions are passed over the stack (`stdcall` calling convention), whereas the first four arguments are passed in registers in 64-bit applications (`fastcall` calling convention) [136].

We limit ourselves to gadgets/code sequences that are likely to be present in almost every process on Windows. In fact, all required gadgets/code sequences can be found in standard Windows libraries and, at least for 32-bit, in every C/C++ program created with default/common compiler and linker settings (at least *Release* or *Debug* configuration; `/Od`, `/O1`, or `/O2` optimization) using Microsoft Visual Studio versions 2010, 2012, or 2013. This

is even valid for the minimal C/C++ program with an empty `main()` whose `.text` section typically has an effective size of under 1 KB. We refer to this executable (*Release*, `/O2`) as minpe-32 and minpe-64, respectively. All code that is present in minpe-32/minpe-64 should also be present in virtually every other program compiled and linked with default settings using Visual Studio.

#### 2.4.1.3 Circumvention for 32-bit Applications

**LBR-Flushing Code Sequences** For 32-bit programs, finding suitable LBR-flushing code sequences is easy: basically most functions that make a certain amount of sub-calls (each sub-call terminates in an indirect branch) and do not much depend on or interfere with the global state of a program comply with the listed requirements. In the following, we refer to a function with these properties as *LBR-flushing function* (lbr-ff). We found for example `lstrcmpiW()`[2] in kernel32.dll to be such a function. When supplied with two identical pointers to (almost) arbitrary data as arguments, we found that it reliably executed more than 20 unsuspicious indirect branches. The fact that the function expects two arguments is of course disadvantageous for an attacker, as this wastes precious space on the (fake) stack. In practice, an attacker could ideally choose an lbr-ff without arguments. For example, we identified the two standard runtime library functions `pre_c_init()` (statically contained in minpe-32) and `EtwInitializeProcess()` (contained in ntdll.dll) as lbr-ffs with zero arguments. It should be clear that suitable lbr-ffs are available in abundance in most real-world applications.

**Invocation Gadgets** Given an lbr-ff, the attacker's goal is to execute it between the ROP chain (step S1) and the invocation of a protected WinAPI function (step S2) in order to flush the LBR stack just before kBouncer's detection logic is triggered. Executing the lbr-ff itself is trivial: it can be part of the ROP chain just like any other gadget. Obviously though, the lbr-ff cannot simply "return" in ROP-manner to the entry point of a protected WinAPI function; kBouncer would certainly detect an attack, as entry points of WinAPI functions are never preceded by a `call` in the static instruction stream.

Instead, the control flow needs to transition from the lbr-ff to the protected WinAPI function in such a way that kBouncer cannot distinguish it from legitimate control flow. We found that for an attacker to achieve this, the availability of a call-preceded and controllable jump-based or call-based *invocation gadget* as depicted in Figure 2.5 is sufficient. In the following, we refer to gadgets of these formats as i-jump-gadgets and i-call-gadgets, respectively.

Given an i-jump-gadget or an i-call-gadget, a protected WinAPI function can be invoked right after an lbr-ff in such a way that the control flow appears legitimate to kBouncer. Figure 2.6 schematically depicts the control flows for both types of gadgets:

⓪ From the ROP chain, the control flow is transferred to the lbr-ff of choice via a traditional `retn` terminated gadget. We need to make sure that at this point the address of the instruction sequence **A** (see Figure 2.5) lies on top of the stack. ① This makes the lbr-ff return to **A** right behind the leading dummy `call` instruction of the i-jump-

[2] `lstrcmpiW()` compares two Unicode strings in a case-insensitive manner.

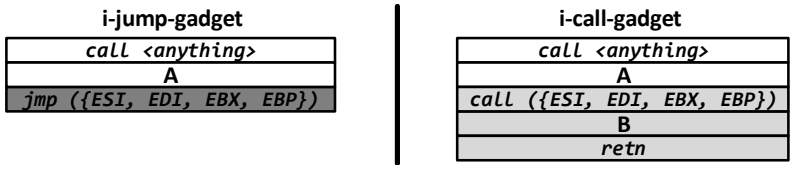


Figure 2.5: Formats of the 32-bit invocation gadget types i-jump-gadget (left) and i-call-gadget (right); blocks labeled **A** and **B** may be empty or contain any sequence of instructions not rendering the execution context uncontrollable.

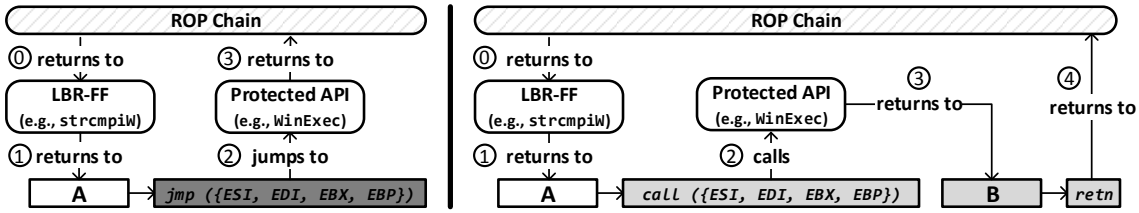


Figure 2.6: Schematic control flow of the invocation of a protected WinAPI (32-bit); **left:** i-jump-gadget **right:** i-call-gadget

gadget/i-call-gadget. ② The protected WinAPI function is then invoked via the indirect `jmp`/`call` instruction following **A**. Typically, this instruction should branch relative to the registers `esi`, `edi`, `ebx`, or `ebp` (e. g., `jmp [ebx*4+edi]` or `call esi`). These registers are premiere choices here, because they are defined to be callee-saved in all common C/C++ calling conventions for x86-32 [136]. Hence, these registers can be assumed to be unaltered by the invocation of virtually any lbr-ff. This allows the attacker to set the registers using regular gadgets (*before* step ⓪). ③, ④ Depending on the invocation gadget type, the WinAPI function either returns directly to the ROP chain (i-jump-gadget) or a detour is taken over the instruction sequence **B** (i-call-gadget).

kBouncer's detection logic is triggered between steps ② and ③. At this point kBouncer cannot detect an attack anymore, as the LBR stack exclusively contains entries corresponding to branches executed *after* step ⓪. Note that the instruction sequence **A** is call-preceded. Hence, the return from the (legitimate) lbr-ff to **A** is unsuspicious to kBouncer.

**Passing of Arguments** Typically, the attacker would align arguments to the WinAPI function on the stack prior to executing the lbr-ff (before step ⓪). Depending on the nature of an invocation gadget though, arguments might also be written to the stack by the instruction sequence **A**. Of course it is a requirement that the instruction sequence **A** does not alter the stack or register values in such a way that the WinAPI function cannot be invoked as intended or the control flow cannot properly resume afterward. For example, the following i-jump-gadget would allow to invoke a WinAPI function but would inevitably lead to the function returning to the invalid address `0`:

```
call    <anything>
push    0
jmp     edi
```

```
1 call    sub_7DD9D8F5
2 xor     eax, eax
3 xor     ebx, ebx
4 xor     ecx, ecx
5 xor     edx, edx
6 xor     edi, edi
7 jmp     esi
```

Listing 2.3: Aligned i-jump-gadget in `TransferToHandler()` found in multiple Windows DLLs

```
1  call    esi
2  mov     __onexitbegin, eax
3  push    dword ptr [ebp-20h]
4  call    esi
5  mov     __onexitend, eax
6  mov     dword ptr [ebp-4], 0FFFFFFFEh
7  call    $+10h
8  mov     eax, edi
9  call    _SEH_epilog4
10 retn
```

Listing 2.4: Aligned i-call-gadget in `_onexit()` of the standard Visual C/C++ library

Also, instructions triggering exceptions/interrupts must of course not be present in **A**. Naturally, similar requirements apply to the trailing instruction sequence **B** of the i-call-gadget.

**Gadget Examples** An example for a suitable i-jump-gadget is given in Listing 2.3. The gadget's **A** sequence (lines 2–6) is composed of `xor` operations on general purpose registers. This should be unproblematic for the attacker in almost all cases.

We implemented a Python script to statically identify this and multiple other suitable i-jump-gadgets and i-call-gadgets in common Windows DLLs in an automated manner. We found this particular i-jump-gadget to be present in the 32-bit versions of kernel32.dll, kernelbase.dll, ntdll.dll, user32.dll, msvcr100.dll, msvcr110.dll, msvcr120.dll, and msvcrt.dll of both Windows 7 and Windows 8. All these DLLs are without doubt among the most frequently used ones on Windows. In fact, ntdll.dll can be found in every Windows user mode process [171].

An example for an i-call-gadget is given in Listing 2.4. We discovered this gadget in the static runtime library function `_onexit()` [137] contained in minpe-32 (and other executables). While also allowing to generically bypass kBouncer, we found the gadget to be slightly more complicated to handle than the i-jump-gadget in Listing 2.3. Reasons are the presence of the `push` instruction in the gadget's **A** sequence (lines 2–3) and the presence of the two static calls in the **B** sequence (lines 5–9).

Obviously, one of these two gadgets should be available to the attacker in most scenarios. If not, it should in the uttermost cases be simple to find comparable gadgets given that the i-call-gadget was found in less than 1 KB of code. Knowledge of these two gadgets proved to be sufficient when we adapted high-profile real world exploits to be undetectable by kBouncer (see Section 2.4.1.5).

#### 2.4.1.4 Circumvention for 64-bit Applications

The described 32-bit approach for bypassing kBouncer is only to some extent applicable to 64-bit. In the default 64-bit calling convention on Windows, the first four arguments to a function are not passed over the stack but in the registers `rcx`, `rdx`, `r8`, and `r9` [136].

```
@loop:
mov     rax, [rbx]
test    rax, rax
jz      @skip
call    rax
@skip:
add     rbx, 8
cmp     rbx, rdi
jb      @loop
mov     rbx, [rsp+28h+arg_0]
add     rsp, 20h
pop     rdi
retn
```

Listing 2.5: Aligned i-loop-gadget in `RTC_Initialize()` of the standard Visual C/C++ library

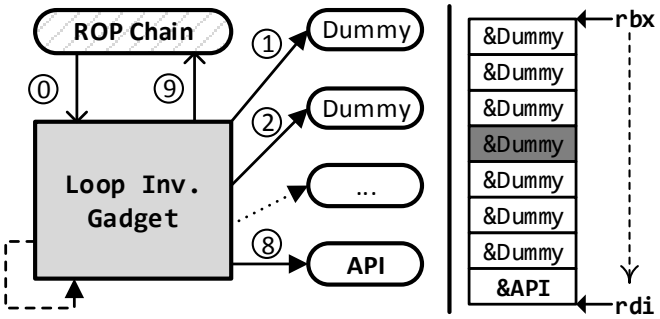


Figure 2.7: Schematic control of the invocation of a WinApi function (i-loop-gadget)

Accordingly, an attacker would in most cases need to preload these registers *before* the invocation of the lbr-ff if the 32-bit approach was followed here. As these four registers are explicitly not callee-saved, they are likely to be altered by almost all lbr-ff. Hence, a different approach is needed for 64-bit systems.

**Loop Invocation Gadget** We found a certain type of 64-bit gadget to be especially suited for both the flushing of the LBR stack and the invocation of protected WinAPI functions. A specimen contained in minpe-64 is given in Listing 2.5. The gadget is comparable to the *dispatcher gadget* that was discussed as foundation for JOP by Bletsch et al. [35]. The gadget interprets `rbx` as an index into a table of code pointers. `rbx` is gradually increased and all pointers are called until `rbx` equals `rdi`. The gadget allows an attacker to execute an arbitrary number of gadgets/functions in a manner that replicates benign control flow. Of course invoked gadgets must generally not alter `rbx` or `rdi`. A very similar loop invocation gadget is for example also contained in `LdrpCallTlsInitializers()` in the 64-bit ntdll.dll. We refer to this type of gadget as i-loop-gadget. An i-loop-gadget can be used to flush the LBR stack and to invoke a protected API subsequently as depicted in Figure 2.7: if a return-succeeded *dummy* gadget is executed at least seven times before the invocation of a protected API, the LBR stack does not contain any traces of the actual ROP chain when kBouncer's detection logic is triggered (for each dummy gadget an indirect call/return pair is executed).

However, finding a suitable dummy gadget is not as easy as it might seem. Obviously, the dummy gadget must be a non-k-gadget as the i-loop-gadget in Listing 2.5 already is a k-gadget. If both are k-gadgets, then an attack is detected by kBouncer. Furthermore, the dummy gadget must neither alter the registers `rbx` and `rdi` nor the registers `rcx`, `rdx`, `r8`, and `r9` carrying the arguments for the WinAPI function. Also, the dummy gadget of course must not render the program state uncontrollable to the attacker. We implemented a Python script to identify appropriate dummy gadgets in standard 64-bit Windows DLLs. We found a variety of long and aligned math related gadgets/functions in ntdll.dll and

msvcr*.dll accessing (almost) exclusively the specialized SSE [107] floating-point registers `xmm0` to `xmm7`. For example, `_remainder_piby2_cw_forAsm()` in msvcr120.dll contains a gadget that does not write to memory and only touches SSE registers and `rax` while executing at least 26 instructions. We also found several long sequences (20+) of `nop` instructions terminated by a return in ntdll.dll. Unfortunately, we did not find a suitable dummy gadget in the `.text` section of minpe-64.

In practice, the attacker might very well interleave dummy gadgets with meaningful k-gadgets, which do not alter `rbx` or `rdi`, in the invocation loop. In fact, as kBouncer per default only considers chains of more than seven k-gadgets harmful, it would be sufficient to execute a single dummy gadget at the fourth position (marked dark gray in Figure 2.7). This would enable the attacker to use the last three gadgets before the invocation of the WinAPI function to conveniently write arguments to the registers `rcx`, `rdx`, `r8`, or `r9`. This would result in less constraints regarding register usage for the employed dummy gadget. Generically bypassing kBouncer using an i-loop-gadget is also possible for 32-bit applications. We found for example the 32-bit equivalent of the i-loop-gadget in Listing 2.5 to be also present in minpe-32. Using the i-jump-gadgets or i-call-gadgets discussed in Section 2.4.1.3 should though in most cases incur less overhead in 32-bit environments. Also, we found suitable dummy gadgets to be relatively sparse compared to lbr-ffs.

#### 2.4.1.5 Example Exploits

To demonstrate the practicality of the described kBouncer bypasses and to assess the resulting overhead, we developed a set of example exploits which we briefly discuss now. As it is tradition, our exploits launch the Windows calculator via an invocation of `WinExec()`. We stress that in all cases much more complicated exploits with multiple WinAPI calls would have been easily possible. No standard Windows defensive mechanisms like ASLR and DEP were disabled or manipulated. We confirmed that our exploits would indeed circumvent kBouncer using our emulator where possible. Due to technical constraints we resorted to manual confirmation using a debugger for Internet Explorer and Firefox.

**Minimal Vulnerable Programs** We extended the discussed minimal executables minpe-32 and minpe-64 to contain a simple buffer overflow vulnerability. We assumed that the attacker knew the base addresses of the main module and msvcr120.dll. In both cases we used common gadgets from msvcr120.dll like `pop eax; ret;` to construct a conventional ROP chain. We then used the discussed i-call-gadget and the lbr-ff in minpe-32 to invoke `WinExec()`; respectively for the 64-bit variant we leveraged the i-loop-gadget in minpe-64 and the discussed dummy gadget in msvcr120.dll. For 32-bit ten extra dwords (32-bit words) were needed in the ROP payload to bypass kBouncer (25 dwords vs. 35 dwords); for 64-bit 20 additional qwords (64-bit words) were required (29 qwords vs. 49 qwords). The relatively large overhead for 64-bit stems from the inclusion of the eight qword long code pointer table.

The two ROP chains for minpe-32 are described in detail in Table 2.1 and Table 2.2; descriptions for the two minpe-64 ROP chains are given in Table 2.3 and Table 2.4. For brevity, the initial *stack pivoting* part that loads `esp`/`rsp` with a memory address under attacker control is omitted for each ROP chain depicted in this section. Each line in a

table describing a ROP chain corresponds to one dword/qword in the attack payload. Lines corresponding to gadgets are highlighted in gray. All regular gadgets are located in msvcr120.dll (32-bit, 64-bit). The i-call-gadget and i-loop-gadget are located in minpe-32 and minpe-64 respectively.

**MPlayer Lite** Pappas et al. used a stack buffer overflow vulnerability in *MPlayer Lite* version r33064 for Windows [78] to evaluate the effectiveness of kBouncer. MPlayer Lite is compiled with MinGW's GCC version 4.5.1. We used gadgets from the bundled avcodec-52.dll to build a conventional ROP-based exploit for the same vulnerability. To circumvent kBouncer, we augmented the ROP chain by an i-loop-gadget located in the static runtime library function `TlsCallback_0()` in mplayer.exe. As corresponding *dummy gadget* we chose another one of MinGW's static runtime library function. Altogether, 37 additional dwords were needed for the augmented ROP chain (21 dwords vs. 58 dwords). We found similar gadgets also in binaries compiled with different MinGW GCC versions.

**Internet Explorer 10** We modified a publicly available exploit for an integer signedness error in Internet Explorer 10 32-bit for Windows 8 by *VUPEN Security* [111]. The original exploit was a winning entry at the popular 2013 Pwn2Own contest. It uses JavaScript code to dynamically construct a ROP chain consisting of 10 dwords to invoke `WinExec()`. In our modified version, four extra dwords are used to incorporate the i-jump-gadget in Listing 2.3 (kernel32.dll) and `lstrcmpiW()` as lbr-ff.

**TorBrowser Bundle / Firefox 17** We modified the exploit allegedly used by the FBI to target users of the *TorBrowser Bundle* [51]. The TorBrowser Bundle is based on Firefox version 17.0.6 for Windows 7 32-bit. We use a ROP payload of 54 dwords to invoke `WinExec()`. The version bypassing kBouncer includes five additional dwords and uses the i-jump-gadget in Listing 2.3 (ntdll.dll) and `EtwInitializeProcess()` (ntdll.dll) as lbr-ff.

#### 2.4.1.6 Possible Improvements

We now briefly review three potential improvements to address our bypasses and discuss their effectiveness.

**Broadening of Gadget Definition** Pappas et al. propose that kBouncer could be improved by considering gadgets longer than 20 instructions if evasion became an issue [152]. We note that such an extension could not substantially tackle the described 32-bit attacks using i-jump-gadgets or i-call-gadgets in conjunction with lbr-ffs (see Section 2.4.1.3): when kBouncer's detection logic is triggered, the effective LBR stack contains one entry corresponding to the invocation gadget and 13 to the lbr-ff. The lbr-ff's LBR entries cannot reasonably be distinguished from benign control flow, as the lbr-ff is a legit function of the attacked application (e. g., `lstrcmpiW()`). A broader definition of k-gadgets could make it harder to find dummy gadgets suitable for the (64-bit) attack approach based on i-loop-gadgets (see Section 2.4.1.4). In practice though, increasing the maximum gadget length such that most suitable dummy gadgets are eliminated, would probably result in

| # | Gadget/Value | Remarks |
|---|---|---|
| 1 | `pop ebx; retn;` | Load random pointer to `.data` section of minpe-32 into `ebx` to prevent an access violation in #10. |
| 2 | *pDataSection* | |
| 3 | `pop ecx; retn;` | Load pointer to a pointer to kernel32.dll in minpe-32's IAT into `ecx`. |
| 4 | *pImportKernel32* | |
| 5 | `mov eax, dword ptr ds:[ecx] retn;` | Read pointer to kernel32.dll from minpe-32's IAT into `eax`. |
| 6 | `pop ecx; retn;` | Load static offset from pointer in `eax` to kernel32!WinExec into `ecx`. |
| 7 | *offsetEaxToWinExec* | |
| 8 | `add eax, ecx; pop ebp; retn;` | Add static offset to `eax`. `eax` now points to kernel32!WinExec. |
| 9 | ? | Arbitrary compensator (`pop ebp` in #8) |
| 10 | `push eax; add al, 0x5f; mov dword ptr ds[ebx+0x2c],eax; mov eax, ebx; pop esi; pop ebx; pop ebp; retn 8;` | Move pointer to kernel32!WinExec to `esi`. |
| 11 | ? | Arbitrary compensator (`pop ebx;` in #10) |
| 12 | ? | Arbitrary compensator (`pop ebp;` in #10) |
| 13 | `pop eax; retn;` | Load static offset from `edx` to #22 into `eax`. |
| 14 | ? | Arbitrary compensator (`retn 8;` in #10) |
| 15 | ? | Arbitrary compensator (`retn 8;` in #10) |
| 16 | *offsetEdx* | Static offset from edx to #22 |
| 17 | `add eax, edx; retn;` | Add `edx` to `eax`. `eax` now points to #22 (first argument to WinExec). |
| 18 | `pop ecx; retn;` | Load address of gadget to execute after WinExec into `ecx`. |
| 19 | pNextGadget | |
| 20 | `pushad; add al, 0; pop ebp; retn 0x10;` | `eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi` and `edi` are pushed to the stack (in this order). The stack pointer is manipulated in such a way that the address in esi is executed with eax's value (1st argument) lying on top of the stack in front of #21 (2nd argument). |
| 21 | *uCmdShow* | 2nd argument to WinExec specifying the display option. |
| 22/23 | "calc.exe" | 1st (dereferenced) argument to WinExec specifying the command line. |

Table 2.1: Basic ROP chain for minpe-32 that is detected by kBouncer; `edx` is expected to point to a certain location on the stack.

| # | Gadget/Value | Remarks |
|---|---|---|
| 1 | `pop ebx; retn;` | Load random pointer to `.data` section of minpe-32 into `ebx` to prevent an access violation in #10. |
| 2 | *pDataSection* | |
| 3 | `pop ecx; retn;` | Load pointer to a pointer to kernel32.dll in minpe-32's IAT into `ecx`. |
| 4 | *pImportKernel32* | |
| 5 | `mov eax, dword ptr ds:[ecx] retn;` | Read pointer to kernel32.dll from minpe-32's IAT into `eax`. |
| 6 | `pop ecx; retn;` | Load static offset from pointer in `eax` to kernel32!WinExec into `ecx`. |
| 7 | *offsetEaxToWinExec* | |
| 8 | `add eax, ecx; pop ebp; retn;` | Add static offset to `eax`. `eax` now points to kernel32!WinExec. |
| 9 | ? | Arbitrary compensator (`pop ebp` in #8) |
| 10 | `push eax; add al, 0x5f; mov dword ptr ds[ebx+0x2c], eax; mov eax, ebx; pop esi; pop ebx; pop ebp; retn 8;` | Move pointer to kernel32!WinExec to `esi`. |
| 11 | ? | Arbitrary compensator (`pop ebx;` in #10) |
| 12 | *subtrahendEbpICallGadget* = 0x20 | The value that is subtracted from `ebp` in line 3 of the i-call-gadget in Listing 2.4. Loaded into `ebp` via `pop ebp;` in #10. |
| 13 | `pop eax; retn;` | Load static offset from `edx` to #22 into `eax`. |
| 14 | ? | Arbitrary compensator (`retn 8;` in #10) |
| 15 | ? | Arbitrary compensator (`retn 8;` in #10) |
| 16 | *offsetEdx* | Static offset from edx to #28 |
| 17 | `add eax, edx; retn;` | Add `edx` to `eax`. `eax` now points to #22 (first argument to WinExec). |
| 18 | `add ebp, eax; retn;` | Add `eax` to `ebp`. `ebp` now points 0x20 bytes behind #22 (1st argument to WinExec); making line 3 of the i-call-gadget push the address of #22 onto the stack prior to the invocation of WinExec. |
| 19 | `pop ecx; retn;` | Load 2nd argument to WinExec into `ecx`. |
| 20 | *uCmdShow* | 2nd argument to WinExec specifying the display option. |
| 21 | `pop ebx; retn;` | Load address of lbr-ff `pre_c_init()` into `ebx`. |
| 22 | *pLbrFlushFunc* | |
| 23 | `pop edx; retn;` | Load address of i-call-gadget (Listing 2.4) into `edx`. |
| 24 | *pICallGadget* | |
| 25 | `pop edi; retn;` | Load address of gadget #26 into `edi`. Gagdet #26 is initially skipped. |
| 26 | `pop ecx; pop edi; mov eax, ebx; pop ebx; retn;` | Gadget is executed directly after gadget #27. Purpose is the increment of `esp` by 12. The lbr-ff is executed next. |
| 27 | `pushad; retn;` | `eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi` and `edi` are pushed to the stack (in this order). The 2nd argument of WinExec and the addresses of i-call-gadget, lbr-ff, and gadget #26 are written to the stack. Gadget #26 is executed next. |
| 28/29 | "calc.exe" | 1st (dereferenced) argument to WinExec specifying the command line. |

Table 2.2: Augmented ROP chain for minpe-32 that bypasses kBouncer; `edx` is expected to point to a certain location on the stack. The augmented chain is identical to the basic chain from gadget #1 to #17.

| # | Gadget/Value | Remarks |
|---|---|---|
| 1 | `pop rax; ret;` | Load static offset from `R8` to #21 (1st argument to WinExec) to `rax`. |
| 2 | offsetR8 | |
| 3 | `add rax, r8; ret;` | Add `r8` to `rax`. `rax` now points to #21. |
| 4 | `mov rcx, rax; mov eax, dword [rcx+4]; add rsp, 0x28; ret;` | Move pointer to #21 to `rcx`. |
| 5–9 | ? | Arbitrary compensators for `add rsp, 0x28;` in #4 |
| 10 | `pop rdx; ret;` | Load pointer to a pointer to kernel32.dll in minpe-64's IAT into `rdx`. |
| 11 | pKernel32Import - 0x10 | |
| 12 | `mov rax, qword [rdx+0x10]; ret;` | Read pointer to kernel32.dll from minpe-64's IAT into `rax`. |
| 13 | `pop rdx; ret;` | Load static offset from pointer in `rax` to kernel32!WinExec. |
| 14 | *offsetRaxToWinExec* | |
| 15 | `add rax, rdx; ret;` | Add `rdx` to `rax`. `rax` now points to kernel32!WinExec. |
| 16 | `pop rdx; ret;` | Load 2nd argument to WinExec to `rdx`. |
| 17 | *uCmdShow* | 2nd argument to WinExec specifying the display option. |
| 18 | `jmp rax;` | Branch to WinExec. |
| 19 | ? | Arbitrary compensator (application specific) |
| 20 | ? | Arbitrary compensator (application specific) |
| 21 | "calc.exe" | 1st (dereferenced) argument to WinExec specifying the command line. |

Table 2.3: Basic ROP chain for minpe-64 that is detected by kBouncer; `r8` is expected to point to a certain location on the stack.

| # | Gadget/Value | Remarks |
|---|---|---|
| 1 | `pop rcx; ret;` | Load random pointer to `.data` section of minpe-64 into `rcx` to prevent an access violation in #3. |
| 2 | *pDataSection* | |
| 3 | `add rdx, rax; mov eax, 0x00000004; mov qword [rcx], rdx; retn;` | Add `rax` to `rdx`. `rdx` now points to a certain position in the stack (application specific). |
| 4 | `pop rax; ret;` | Load offset from `rdx` to the beginning of the code pointer table (#12). |
| 5 | *offsetRdxToCodePointerTable* | |
| 6 | `add rax, rdx; ret;` | Add `rdx` to `rax`. `rax` now points to the beginning of the code pointer table (#12). |
| 7 | `push rax; pop rbx; ret;` | Move pointer to code pointer table to `rbx`. |
| 8 | `pop rax; ret` | Load offset from `rdx` to the end of the code pointer table (#20). |
| 9 | *offsetRdxToEndCodePointerTable* | |
| 10 | `add rax, rdx; ret;` | Add `rdx` to `rax`. `rax` now points to the end of the code pointer table (#20). |
| 11 | `push rax; add rsp, 0x40; pop rdi; ret;` | Move pointer to end of code pointer table (#20) to `rdi` and advance `rsp` to #20. Accordingly the next executed gadget is #20. |
| **12–18** | *pDummyGadget* | Pointers to dummy gadget in `msvcr120!_remainder_piby2_cw_forAsm` (see §2.4.1.2). |
| **19** | *pDispatch* | Pointer to gadget `call r8;`. Final entry of code pointer table. Used to call WinExec. |
| 20 | `pop rcx; ret;` | Load pointer to a pointer to kernel32.dll in minpe-64's IAT into `rcx`. |
| 21 | pKernel32Import - 0x28 | |
| 22 | `pop r9; pop r8; ret;` | Load offset from pointer to kernel32.dll to kernel32!WinExec into `r9`. |
| 23 | offsetKernel32ToWinExec | |
| 24 | ? | Arbitrary compensator (`pop r8;` in #22) |
| 25 | `mov r8, qword [rcx+0x28]; mov rax, r8; ret;` | Read pointer to kernel32.dll from minpe-64's IAT into `r8`. |
| 26 | `add r8, r9; add rax, r8; ret;` | Add `r9` to `r8`. `r8` now points to WinExec. |
| 27 | `pop rax; ret;` | Load offset from `rdx` to #39 (1st argument to WinExec) into `rax`. |
| 28 | *offsetRdxToArg0* | |
| 29 | `add rax, rdx; ret;` | Add `rdx` to `rax`. `rax` now points to #39. |
| 30 | `mov rcx, rax; mov eax, dword [rcx+0x04]; add rsp, 0x28; ret;` | Move 1st argument to `rcx`. |
| 31–34 | ? | Arbitrary compensator (`add rsp, 0x28;` in #30) |
| 35 | `pop rdx; ret;` | Load 2nd argument into `rdx`. |
| 36 | *uCmdShow* | 2nd argument to WinExec specifying the display option. |
| 37 | i-loop-gadget | The i-loop-gadget in minpe-64 is used to execute the function pointer table (#12 to #19). |
| 38 | *pNextGadget* | Address of gadget the i-loop-gadget should return to after having invoked WinExec. |
| 38 | ? | Arbitrary compensator (application specific) |
| 39 | "calc.exe" | 1st (dereferenced) argument to WinExec specifying the command line. |

Table 2.4: Augmented ROP chain for minpe-64 that bypasses kBouncer; `rdx` is expected to point to a certain location on the stack. The code pointer table used in the i-loop-gadget is highlighted in light gray.

unacceptable high numbers of overall false positives. Even for a maximum length of 20, entire non-trivial functions fall already under the k-gadget definition.

**Larger LBR Stack** Pappas et al. suggest that future processor generations with larger LBR stacks "would allow kBouncer to achieve even higher accuracy by inspecting longer execution paths [...]" [152]. In such a case, our described approaches could easily be adapted to create longer sequences of indirect branches resembling benign ones. For example, the described i-loop-gadget can be used to create such sequences of almost arbitrary length. Also, finding lbr-ffs which do so is easy. The discussed `lstrcmpiW()` can for example be used to create dozens of legit indirect branches.

**Heuristic Detection of Invocation Gadgets** One could attempt to extend kBouncer to heuristically check for LBR entries corresponding to the discussed types of invocation gadgets. This could, depending on the actual implementation, very well fend off the described attacks. However, we expect high numbers of false positives from such a measure, as the same invocation patterns can very well occur for benign control flows.

### 2.4.2 Security Assessment of ROPGuard

ROPGuard is a runtime ROP detection approach for user mode applications on Windows [83]. It placed 2nd to kBouncer at the BlueHat Prize and is incorporated into the *Enhanced Mitigation Experience Toolkit* (EMET) [138] that is provided as optional security enhancement for Windows.

Similar to kBouncer, ROPGuard hooks a set of critical WinAPI functions in user mode processes. Whenever such a hook is triggered, ROPGuard as implemented in EMET 4.1—the most recent version at the time of this writing—tries to detect ROP-based exploits via a variety of checks. We describe the two most relevant ones now briefly [83, 135]:

- *Past and Future Control Flow Analysis*: ROPGuard verifies that the return address of a protected WinAPI function is call-preceded. Furthermore, it simulates the control flow in a simple manner from the return address onwards and checks for future non call-preceded returns. Simulation is performed until a certain threshold number of future instructions was examined or any call or jump instruction is encountered.
- *Stack Checks*: ROPGuard checks if the stack pointer points within the expected memory range for the given thread. It is common practice for attackers to divert the stack pointer to a memory region (e. g., the heap) under their control. ROPGuard also blocks attempts to make the stack executable.

We found that our kBouncer example exploits that rely either on i-call-gadgets or on i-loop-gadgets (both *minimal vulnerable programs* and *MPlayer*) already bypassed ROPGuard's implementation in EMET. In turn, ROPGuard successfully stopped all of the three corresponding unmodified exploits. For ROPGuard, the discussed i-call-gadgets and i-loop-gadgets invoke the protected `WinExec()` via seemingly legitimate calls. These gadgets also make ROPGuard's future control flow simulation stop early due to subsequent jumps/calls. The stack-related checks do not apply to our example exploits.

### 2.4.3 Security Assessment of ROPecker

*ROPecker*, a runtime ROP exploit mitigation system, was presented by Cheng et al. in 2014 [54]. ROPecker aperiodically checks for abnormal branch sequences in an application's control flow. For that, ROPecker combines kBouncer-like examination of the LBR stack with ROPGuard-like future control flow simulation. Cheng et al. specifically report on a prototype implementation of ROPecker as a kernel module for x86-32 Linux systems. Hence, we also only consider this platform. For evaluation purposes, we implemented an experimental standalone Pin-based emulator for ROPecker. We are confident that this emulator accurately captures most of ROPecker's aspects. All experiments we report on in the following were conducted on either Ubuntu 12.0.4 or Debian 7.4.0 systems.

#### 2.4.3.1 Triggering of Detection Logic

Other than comparable approaches, ROPecker does not apply any form of binary rewriting such as API function hooking to inspect an application's control flow. Instead, ROPecker ensures that only a small fixed-size dynamic set of code pages is executable at any given time within a process. ROPecker's ROP detection logic is invoked every time an access violation is triggered due to the target application's control flow reaching a new page *outside* the set of executable pages. If no attack is detected, ROPecker replaces the oldest page in the set of executable pages with the newly reached page and resumes the execution of the corresponding thread/process. Cheng et al. refer to this technique as a "sliding window mechanism". They suggest using a window/set size of two to four executable pages, corresponding to 8 to 16 KB of executable code, because it is supposedly hard to find enough gadgets for a meaningful attack in less than 20 KB of code [54]. The pages inside the sliding window do not necessarily need to be adjacent.

For our emulator, we use a fixed sliding window size of exactly one page to achieve fine-granular capturing. Note that a smaller sliding window size results in ROPecker's detection logic being triggered more often. Hence, chances for false negatives decrease while in turn chances for false positives increase.

#### 2.4.3.2 Examination of Indirect Branch Sequences

Each time it is triggered, ROPecker's detection logic tries to identify attacks by analyzing the past and the (simulated) future control flow of a thread/process for chains of ROP gadgets. Per default, ROPecker considers a sequence of instructions to be a gadget in case it meets the following criteria [54]: *(i)* the last instruction is an indirect branch; *(ii)* no other branch (e. g., `call` or `jnz`) is contained; *(iii)* it consists of at most six instructions. This limit was arbitrarily chosen by Cheng et al. ROPecker can be configured to consider longer gadgets. We refer to gadgets that comply with ROPecker's definition as r-gadgets.

**Analysis of Past and Future Indirect Branches** Like kBouncer, ROPecker configures the processor's LBR facility to only track indirect branches in user mode. Whenever execution reaches a page outside the sliding window, ROPecker first examines the thread's/process' *past* indirect branches for a chain of r-gadgets via the LBR stack: going backward from the most recent one, it is checked for each LBR entry (which necessarily ends in an indirect

| **Application** | $max_{nor}$ | **Activity** |
|---|---|---|
| Nginx 1.4.0 | 5 | delivery of small web page |
| Adobe Reader 9.5.5 | 9 | opening of document |
| Pidgin 2.10.9 | 9 | IRC chat |
| Gimp 2.8.2 | 9 | simple drawing |
| VLC 2.0.8 | 11 | playback of short OGG video |
| LibreOffice Calc 3.5.7.2 | 17 | creation of simple spreadsheet |

Table 2.5: Exemplary $max_{nor}$ values as determined by our ROPecker emulator

branch) if its branch destination is an r-gadget. The *past* detection stops with the first entry not matching this characteristic. After that, ROPecker simulates the thread's/process' *future* indirect branches using rather complex emulation techniques going forward from the most recent LBR entry's branch destination. As soon as a code sequence is encountered that does not qualify as r-gadget, the *future* detection stops. If the accumulated length of the *past* and the *future* gadget chains is above a certain threshold, an attack is assumed.

**Gadget Chain Detection Threshold** Cheng et al. suggest using a chain detection threshold between 11 and 16 r-gadgets where "an ideal threshold should be smaller than the minimum length $min_{rop}$ of all ROP gadget chains, and at the same time, be larger than the maximum length $max_{nor}$ of the gadget chains identified from normal execution flows" [54]. They report that various real world and artificial ROP chains analyzed by them consisted of 17 to 30 gadgets. Hence, they universally assume $min_{rop} = 17$. To assess $max_{nor}$, Cheng et al. examined a variety of applications (certain Linux coreutils, SPEC INT2006, ffmpeg, graphics-magick, and Apache web server) during runtime. For the code paths triggered in their experiments, they found $max_{nor}$ overall to be 10 and for Apache even only 4; values well below their empirically determined $min_{rop} = 17$.

In practice, higher values for $max_{nor}$ are not totally unlikely though. Consider for example again the simple recursive function `factorial()` from Listing 2.1 in Section 2.4.1 whose epilogue qualifies as r-gadget. We used our experimental emulator to explore the range of $max_{nor}$ for popular applications not covered by experiments conducted by Cheng et al. The results are listed in Table 2.5. The encountered chain of 17 r-gadgets for LibreOffice Calc resulted from a long chain of returns from nested function calls (similar to the `factorial()` example). We emphasize that our emulator with a sliding window size of only one page naturally catches more false positives and produces higher $max_{nor}$ than configurations with larger sliding windows. However, these numbers suggest that ROPecker might not be equally well applicable to all kinds of applications, as in certain cases $max_{nor}$ could be too high to allow for a reasonably low detection threshold $min_{rop}$.

#### 2.4.3.3 Circumvention

We now discuss methods for the generic circumvention of ROPecker. In general, we find that the narrow definition of r-gadgets makes ROPecker only a small hurdle for aware attackers.

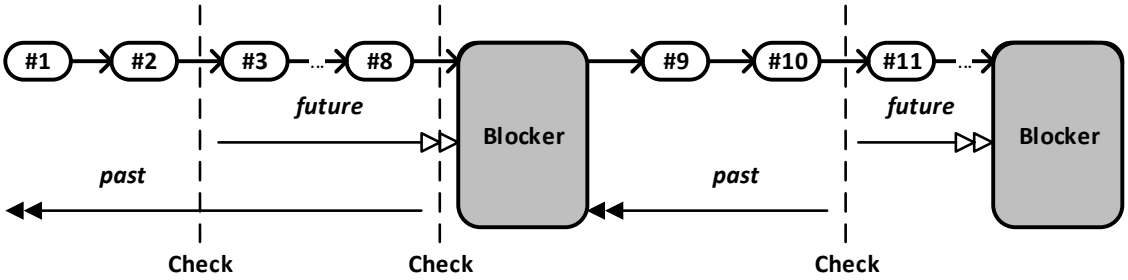


Figure 2.8: Generic layout of a gadget chain bypassing ROPecker; conventional gadgets (white) are interleaved with gadgets stopping the *past* and *future* detection logic (gray).

Cheng et al. state that ROPecker's "[...] payload detection algorithm is designed based on the assumption that a gadget does not contain direct branch instructions, which is also used in the many previous work [...]. Therefore, the gadget chain detection stops when a direct branch instruction is encountered" [54]. They also acknowledge that an "[...] adversary may carefully insert long gadgets into consecutive short gadgets to make the length of each segmented gadget chain not exceed the gadget chain threshold [...]" to achieve the same. Note that these statements already describe all that is necessary in order to successfully bypass ROPecker in a generic manner. As depicted in Figure 2.8, attackers simply need to take care to periodically mix in a non-r-gadget (containing a branch or more than six instructions) into their gadget chains in order to stop ROPecker's *past* and *future* detection logic before the given detection threshold is reached. In the following, we refer to such a gadget as blocker-gadget.

Cheng et al. argue that to the best of their knowledge an attack using jump-containing gadgets "[...] has not been found in real-life". We note that this observation does not necessarily imply that jump-containing (or long) gadgets are hard to use. Instead, it is in the uttermost cases trivial for an attacker to find and use such gadgets, as they do not need to be meaningful in any context. The only requirement is that they do not render the program state uncontrollable as already discussed in Section 2.4.1 for kBouncer. Even entire regular functions as the ones discussed in Section 2.4.1.3 can be misused by attackers here. In our example exploit against ROPecker (see Section 2.4.3.4) we use for example the standard POSIX function `usleep()` as blocker-gadget.

#### 2.4.3.4 Example Exploit

To demonstrate the applicability of the discussed ROPecker bypassing strategy, we created a ROP-based exploit for a stack buffer overflow vulnerability (CVE-2013-2028) [167] in the popular web server Nginx version 1.4.0. We inserted the function `usleep()` as blocker-gadget into the ROP chain after at least every seventh regular gadget. The entire resulting ROP payload is 107 dwords long—92 dwords are needed without ROPecker evasion—and creates a file on the target system using the `system()` function. Our ROPecker emulator detects a maximum chain length of nine for the exploit due to the epilogue of `usleep()`

containing two chained r-gadgets. As this is below the default detection threshold of 11, the attack goes unnoticed.

#### 2.4.3.5 Possible Improvements

We again briefly review potential improvements to address our bypasses and discuss their effectiveness.

**Detection of Unaligned Gadgets** Cheng et al. propose that ROPecker could be improved by considering the execution of unaligned instructions as attack [54]. They note though, that it may not always be possible to decide if a given x86 instruction sequence is aligned or not. Attackers restricted to aligned gadgets would probably need longer gadget chains on average to achieve compromise. Also, finding suitable gadgets in general would be more complicated. The generic circumvention approach described in Section 2.4.3.3 could though not be prevented.

**Accumulation of Chain Lengths** To tackle attacks relying on blocker-gadgets, Cheng et al. suggest an extension to ROPecker that accumulates the detected chain lengths for multiple (e. g., three) consecutive sliding window updates. However, we find that an attacker could still generically avoid detection by using a (meaningless) function as blocker-gadget which updates the sliding window several times. When such a function returns to the next r-gadget, the accumulated chain length should in the uttermost cases be well below the detection threshold. We found for example the already mentioned `usleep()` to be a suitable function for this purpose. In our experiments, the function reliably switched pages several times before finally executing a system call.

**Broadening of Gadget Definition** Lastly, Cheng et al. propose extending ROPecker in such a way that instruction sequences connected by direct jumps are also considered as gadgets, but also state that this might increase the number of false positives. In order to evaluate the practicality of such an extension, we experimentally modified our ROPecker emulator to consider kBouncer's k-gadgets (up to 20 instructions including direct jumps) instead of r-gadgets. With this hypothetical extension in place, we generally encountered high numbers of false positives often corresponding to astonishingly long benign chains of k-gadgets. For example, our emulator detected a chain of length 14 in libc for a small *hello world* application. While monitoring VLC during the playback of a short OGG video, the emulator even detected chains of lengths 77 and 82 in librsvg2 and libexpat respectively; the first being induced by a long static sequence of indirect calls to a very short function and the latter by a compact looped switch-case statement implemented using a central indirect jump. This hints at ROPecker possibly not being reasonably extendable to consider significantly more complex gadgets.

**Checking for Illegal Returns.** We believe that ROPecker's defensive strength could indeed be increased if it would consider returns to non call-preceded locations as indicator for an attack like kBouncer and ROPGuard do. Such an extension would effectively require

attackers to largely resort to call-preceded gadgets or JOP-like concepts such as i-loop-gadgets (see Section 2.4.1.4). While this would not prevent bypasses, it could significantly raise the bar. We would expect negligible overhead and close to zero additional false positives from such an extension as to the best of our knowledge returns to not call-preceded locations virtually never occur in benign control flows.

## 2.5 Challenging Defenses with Counterfeit Object-oriented Programming

After the discussion of different instantiations of ROP, we now present a new form of code-reuse attack that we dub *counterfeit object-oriented programming* (COOP). Typically, code-reuse attacks exhibit unique characteristics in the control flow (and the data flow) that allow for generic protections regardless of the language an application was programmed in (see Section 2.2.4.2). For example, if one can afford to monitor all return instructions in an application while maintaining a full shadow call stack, even advanced ROP-based attacks [47, 65, 90, 91, 180], including the ones presented in Section 2.4, cannot be mounted [3, 66, 82].This is different for COOP: it exploits the fact that each C++ virtual function is address-taken, which means that a static code pointer exists to it. Accordingly, C++ applications usually contain a high ratio of address-taken functions; typically a significantly higher one compared to C applications. If for example an imprecise CFI solution does not consider C++ semantics, then these functions are all likely valid indirect call targets [1] and can thus be abused (see Section 2.2.5.2).

COOP exclusively relies on C++ virtual functions that are invoked through corresponding calling sites as *gadgets*. Hence, without deeper knowledge of the semantics of an application developed in C++, COOP's control flow cannot reasonably be distinguished from a benign one. Another important difference to existing code-reuse attacks is that in COOP conceptually no code pointers (e. g., return addresses or function pointers) are injected or manipulated. As such, COOP is immune against defenses that protect the integrity and authenticity of code pointers. Moreover, in COOP, gadgets do not work relative to the stack pointer. Instead, gadgets are invoked relative to the this-ptr on a set of adversary-defined *counterfeit objects*. Note that in C++, the this-ptr allows an object to access its own address. Addressing relative to the this-ptr implies that COOP cannot be mitigated by defenses that prevent the stack pointer to point to the program's heap [83], which is typically the case for ROP-based attacks launched through a heap-based memory corruption vulnerability.

The counterfeit objects used in a COOP attack typically overlap such that data can be passed from one gadget to another. Even in a simple COOP program, positioning counterfeit objects manually can become complicated. Hence, we implemented a programming framework that leverages the Z3 SMT solver [67] to derive the object layout of a COOP program automatically.

### 2.5.1 Approach

COOP is a code-reuse attack approach targeting applications developed in C++ or possibly other object-oriented languages. We note that many of today's notoriously attacked applications are written in C++ (or contain major parts written in C++); examples include, Microsoft Internet Explorer, Google Chromium, Mozilla Firefox, Adobe Reader, Microsoft Office, LibreOffice, and OpenJDK.

In the following, we first state our design goals and our attacker model for COOP before we describe the actual building blocks of a COOP attack. For brevity reasons, the rest of this section focuses on Microsoft Windows and the x86-64 architecture as runtime environment. The COOP concept is generally applicable to C++ applications running on any operating system; it also extends to other architectures. Interestingly, mounting a COOP attack on a RISC architecture can even be simpler than on a CISC architecture because calling conventions that pass function arguments through registers rather than over the stack facilitate COOP (further discussed in Section 2.5.1.4).

#### 2.5.1.1 Goals

With COOP we aim to demonstrate the feasibility of creating powerful code-reuse attacks that do not exhibit the revealing characteristics of existing attack approaches. Even advanced existing variants of return-into-libc, ROP, JOP, or COP [32,37,47,65,90,91,180,212] rely on control flow and data-flow patterns that are rarely or never encountered for regular code; among these are typically one or more of the following:

**C-1** indirect *calls/jumps* to non address-taken locations

**C-2** *returns* not in compliance with the call stack

**C-3** excessive use of indirect branches

**C-4** pivoting of the stack pointer (possibly temporarily)

**C-5** injection of new code pointers or manipulation of existing ones

These characteristics still allow for the implementation of effective, low-level, and programming language-agnostic protections. For instance, maintaining a full shadow call stack [3, 66, 82] suffices to fend off virtually all ROP-based attacks.

With COOP we demonstrate that it is not sufficient to generally rely on the characteristics **C-1**–**C-5** for the design of code-reuse defenses; we define the following goals for COOP accordingly:

**G-1** do not expose the characteristics **C-1**–**C-5**.

**G-2** exhibit control flow and data flow similar to those of benign C++ code execution.

**G-3** be widely applicable to C++ applications.

**G-4** achieve Turing-completeness under realistic conditions.

#### 2.5.1.2 Attacker Model

In general, code-reuse attacks against C++ applications oftentimes start by hijacking a C++ object and its vptr. Attackers achieve this by exploiting a spatial or temporal memory corruption vulnerability such as an overflow in a buffer adjacent to a C++ object or a use-after-free condition. When the application subsequently invokes a virtual function on the hijacked object, the attacker-controlled vptr is dereferenced and a vfptr is loaded from a memory location of the attacker's choice. At this point, the attacker effectively controls the *program counter* (`rip` in x86-64) of the corresponding thread in the target application. Generally for code-reuse attacks, controlling the program counter is one of the two basic requirements. The other one is gaining (partial) knowledge on the layout of the target application's address space. Depending on the context, there may exist different techniques to achieve this [32, 105, 186, 194].

For COOP, we assume that the attacker controls a C++ object with a vptr and that she can infer the base address of this object or another auxiliary buffer of sufficient size under her control. Further, she needs to be able to infer the base addresses of a set of C++ modules whose binary layouts are (partly) known to her. For instance, in practice, knowledge on the base address of a single publicly available C++ library in the target address space can be sufficient.

These assumptions conform to the attacker settings of most defenses against code-reuse attacks. In fact, many of these defenses assume far more powerful adversaries that are, e. g., able to read and write large (or all) parts of an application's address space with respect to page permissions.

#### 2.5.1.3 Basic Approach

Every COOP attack starts by hijacking one of the target application's C++ objects. We call this the *initial object*. Up to the point where the attacker controls the program counter, a COOP attack does not deviate much from other code-reuse attacks: in a conventional ROP attack, the attacker typically exploits her control over the program counter to first manipulate the stack pointer and to subsequently execute a chain of short, return-terminated gadgets. In contrast, in COOP, virtual functions existing in an application are repeatedly invoked on counterfeit C++ objects carefully arranged by the attacker.

**Counterfeit Objects** Typically, a counterfeit object carries an attacker-chosen vptr and a few attacker-chosen data fields. Counterfeit objects are *not* created by the target application, but are injected in bulk by the attacker. Whereas the *payload* in a ROP-based attack is typically composed of fake return addresses interleaved with additional data, in a COOP attack, the payload consists of counterfeit objects and possibly additional data. Similar to a conventional ROP payload, the COOP payload containing all counterfeit objects is typically written as one coherent chunk to a single attacker-controlled memory location.

**Vfgadgets** We call the virtual functions used in a COOP attack *vfgadgets*. As for other code-reuse attacks, the attacker identifies useful vfgadgets in an application prior to the

| Vfgadget type | Purpose | Code example |
|---|---|---|
| ML-G | The main loop; iterate over container of pointers to counterfeit object and invoke a virtual function on each such object. | see Figure 2.9 |
| ARITH-G | Perform arithmetic or logical operation. | see Figure 2.12 |
| W-G | Write to chosen address. | see Figure 2.12 |
| R-G | Read from chosen address. | no example given, similar to W-G |
| INV-G | Invoke C-style function pointer. | see Figure 2.16 |
| W-COND-G | Conditionally write to chosen address. Used to implement conditional branching. | see Figure 2.14 |
| ML-ARG-G | Execute vfgadgets in a loop and pass a field of the *initial object* to each as argument. | see Figure 2.14 |
| W-SA-G | Write to address pointed to by first argument. Used to write to *scratch area*. | see Figure 2.14 |
| MOVE-SP-G | Decrease/increase stack pointer. | no example given |
| LOAD-R64-G | Load x86-64 argument register `rdx`, `r8`, or `r9` with value. | see Figure 2.12 |

Table 2.6: Overview of COOP vfgadget types that operate on object fields or arguments; general purpose types are atop; auxiliary types are below the double line.

actual attack through source code analysis or reverse engineering of binary code. Even when source code is available, it is necessary to determine the actual object layout of a vfgadget's class on binary level as the compiler may remove or pad certain fields. Only then the attacker is able to inject compatible counterfeit objects.

We identified a set of vfgadget types that allows to implement expressive (and Turing-complete) COOP attacks in x86-32 and x86-64 environments. These types are listed in Table 2.6. In the following, we gradually motivate our choice of vfgadget types based on typical code examples. These examples revolve around the simple C++ classes `Student`, `Course`, and `Exam`, which reflect *some* common code patterns that we found to induce useful vfgadgets. From Section 2.5.1.3 to Section 2.5.1.3, we first walk through the creation of a COOP attack code that writes to a dynamically calculated address; along the way, we introduce COOP's integral concepts of *The Main Loop*, *Counterfeit Vptrs*, and *Overlapping Counterfeit Objects*. After that, from Section 2.5.1.4 to Section 2.5.1.6, extended concepts for *Passing Arguments to Vfgadgets*, *Calling API Functions*, and *Implementing Conditional Branches and Loops* in COOP are explained.

The reader might be surprised to find more C++ code listings than actual assembly code in the following. This is owed to the fact that most of our vfgadgets types are solely defined by their high-level C++ semantics rather than by the side effects of their low level assembly code. These types of vfgadgets are thus likely to survive compiler changes or even the transition to a different operating system or architecture. In the cases where assembly code is given, it is the output of the Microsoft Visual C++ compiler (MSVC) version *18.00.30501* that is shipped with Microsoft Visual Studio 2013.

```
class Student {
public:
    virtual void incCourseCount() = 0;
    virtual void decCourseCount() = 0;
};

class Course {
private:
    Student **students;
    size_t nStudents;
public:
    /* ... */
    virtual ~Course() {
        for (size_t i = 0; i < nStudents; i++)
            students[i]->decCourseCount();
        delete students;
    }
};
```

ML-G

Figure 2.9: Example for ML-G: the virtual destructor of the class `Course` invokes a virtual function on each object pointer in the array `students`.

**The Main Loop** To repeatedly invoke virtual functions without violating goals **G-1** and **G-2**, every COOP program essentially relies on a special *main loop vfgadget* (ML-G). The definition of an ML-G is as follows:

**Definition:** *A virtual function that iterates over a container (e. g., a C-style array or a vector) of pointers to C++ objects and invokes a virtual function on each of these objects.*

Virtual functions that qualify as ML-G are common in C++ applications. Consider for example the code in Figure 2.9: the class `Course` has a field `students` that points to a C-style array of pointers to objects of the abstract base class `Student`. When a `Course` object is destroyed (e. g., via `delete`), the virtual destructor[3] `Course::~Course` is executed and each `Student` object is informed via its virtual function `decCourseCount()` that one of the courses it was subscribed to does not exist anymore.

The idea of employing an ML-G as the central dispatcher in COOP directly emerged from the analysis of the i-loop-gadget which we use in our attack against kBouncer on x86-64 (see Section 2.4.1.4). Conceptually, ML-Gs and i-loop-gadgets are very similar as both can be misused to invoke attacker-chosen code pointers in a loop. However, ML-Gs are C++ specific and probably considerably more common than i-loop-gadgets. Furthermore, i-loop-gadgets typically require the attacker to inject code pointers which we aim to avoid with COOP.

**Layout of the Initial Object** The attacker shapes the initial object to resemble an object of the class of the ML-G. For our example ML-G `Course::~Course`, the initial object should look as depicted in Figure 2.10: its vptr is set to point into an existing vtable that

[3]It is common practice to declare a virtual destructor when a C++ class has virtual functions.

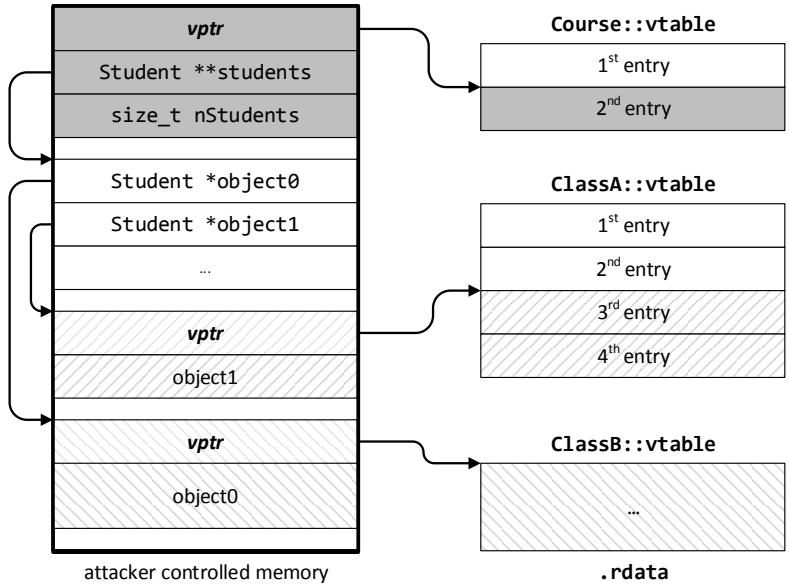


Figure 2.10: Basic layout of attacker controlled memory (left) in a COOP attack using the example ML-G `Course::~Course`. The initial object (dark gray, top left) contains two fields from the class `Course`. Arrows indicate a *points-to* relation.

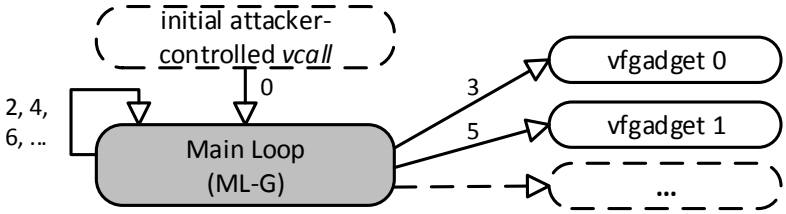


Figure 2.11: Schematic control flow in a COOP attack; transitions are labeled according to the order they are executed.

contains a reference to the ML-G such that the first vcall under attacker control leads to the ML-G. In contrast, in a ROP-based attack, this first vcall under attacker control typically leads to a gadget moving the stack pointer to attacker controlled memory. The initial object contains a subset of the fields of the class of the ML-G; i. e., all data fields required to make the ML-G work as intended. For our example ML-G, the initial object contains the fields `students` and `nStudents` of the class `Course`; the field `students` is set to point to a C-style array of pointers to counterfeit objects (*object0* and *object1* in Figure 2.10) and `nStudents` is set to the total number of counterfeit objects. This makes the `Course::~Course` ML-G invoke a vfgadget of the attacker's choice for each counterfeit object. Note how the attacker controls the vptr of each counterfeit object. Figure 2.11 schematically depicts the control-flow transitions in a COOP attack.

**Counterfeit Vptrs** The control flow and data flow in a COOP attack should resemble those of a regular C++ program (**G-2**). Hence, we avoid introducing fake vtables and reuse existing ones instead. Ideally, the vptrs of all counterfeit objects should point to the *beginning* of existing vtables. Depending on the target application, it can though be difficult to find vtables with a useful entry at the offset that is fixed for a given vcall site. Consider for example our ML-G from Figure 2.9: counterfeit objects are treated as instances of the abstract class `Student`. For each counterfeit object, the $2^{nd}$ entry—corresponding to `decCourseCount()`—in the supplied vtable is invoked. (The $1^{st}$ entry corresponds to `incCourseCount()`.) Here, a COOP attack would ideally only use vfgadgets that are the $2^{nd}$ entry in an existing vtable. Naturally, this largely shrinks the set of available vfgadgets.

This constraint can be sidestepped by relaxing goal **G-2** and letting vptrs of counterfeit objects not necessarily point to the exact beginning of existing vtables but to certain positive or negative offsets as is shown for *object1* in Figure 2.10. When such *counterfeit vptrs* are used, any available virtual function can be invoked from a given ML-G.

E. g., to invoke the $4^{th}$ entry in a certain vtable under the given ML-G, the attacker makes a counterfeit object's vptr point to the $3^{rd}$ entry of that vtable as Figure 2.10 depicts for *object1* and `ClassA::vtable`. The vcall in the ML-G then interprets the $4^{th}$ entry of that vtable as the $2^{nd}$ entry of a `Student` vtable.

**Overlapping Counterfeit Objects** So far we have shown how, given an ML-G, an arbitrary number of virtual functions (vfgadgets) can be invoked while control flow and data flow resemble those of the execution of benign C++ code.

Two exemplary vfgadgets of types ARITH-G (arithmetic) and W-G (writing to memory) are given in Figure 2.12: in `Exam::updateAbsoluteScore()` the field `score` is set to the sum of three other fields; in `SimpleString::set()` the field `buffer` is used as destination pointer in a write operation. In conjunction, these two vfgadgets can be used to write attacker-chosen data to a dynamically calculated memory address. For this, two *overlapping* counterfeit objects are needed and their alignment is shown in Figure 2.13.

The key idea here is that the fields `score` in *object0* and `buffer` in *object1* share the same memory. This way, the result of the summation of the fields of *object0* in `Exam::updateAbsoluteScore()` is written to the field `buffer` of *object1*. For example, *object0.scoreA* could hold a previously determined base pointer (*base-ptr*) to a memory region, *object0.scoreB* could hold a fixed offset into that region, and *object0.scoreC* would simply be set to 0. The write operation in `SimpleString::set()` would then use

$$\begin{aligned} &\textit{objcect0.scoreA} + \textit{object0.scoreB} + \textit{object0.scoreC} = \\ &\textit{base-ptr} + \textit{offset} \end{aligned}$$

as destination pointer in `strncpy()`. Note how here, technically, also *object0.topic* and *object1.vptr* overlap. As the attacker does not use *object0.topic* this not a problem and she can simply make the shared field carry *object1.vptr*. Of course, in our example, the attacker would likely not only wish to control the *destination address* of the write operation through *object1.buffer* but also the *source address*. For this, she needs to be able to set the

```
class Exam {
private:
     size_t scoreA, scoreB, scoreC;
public:
     /* ... */
     char *topic;
     size_t score;
     virtual void updateAbsoluteScore() {
         score = scoreA + scoreB + scoreC;
     }                                                    ARITH-G

     virtual float getWeightedScore() {
         return (float)(scoreA*5+scoreB*3+scoreC*2) / 10;
     }                                                    LOAD-R64-G
};

struct SimpleString {
     char* buffer;
     size_t len;
     /* ... */
     virtual void set(char* s) {
         strncpy(buffer, s, len);                         W-G
     }
};
```

Figure 2.12: Examples for ARITH-G, LOAD-R64-G, and W-G; for simplification, the native integer type `size_t` is used.

argument for the vfgadget `SimpleString::set()`. How this can be achieved in COOP is described next.

#### 2.5.1.4 Passing Arguments to Vfgadgets

The overlapping of counterfeit objects is an important concept in COOP. It allows for data to flow between vfgadgets through object fields regardless of compiler settings or calling conventions. Unfortunately, we found that useful vfgadgets that operate exclusively on object fields are rare in practice. In fact, most vfgadgets we use in our real world exploits (see Section 2.5.4) operate on both fields and arguments as is the case for `SimpleString::set()`.

Due to divergent default calling conventions, we describe different techniques for passing arguments to vfgadgets for x86-64 and x86-32 in the following. Other than in Section 2.4, we begin with x86-64 and not with x86-32 as the technique for the former is simpler.

**Approach Windows x86-64** In the default x86-64 calling convention on Windows, the first four (non-floating point) arguments to a function are passed through the registers `rcx`, `rdx`, `r8`, and `r9` [136]. In case there are more than four arguments, the additional arguments are passed over the stack. For C++ code, the this-ptr is passed through `rcx` as the first argument. All four argument registers are defined to be caller-saved; regardless

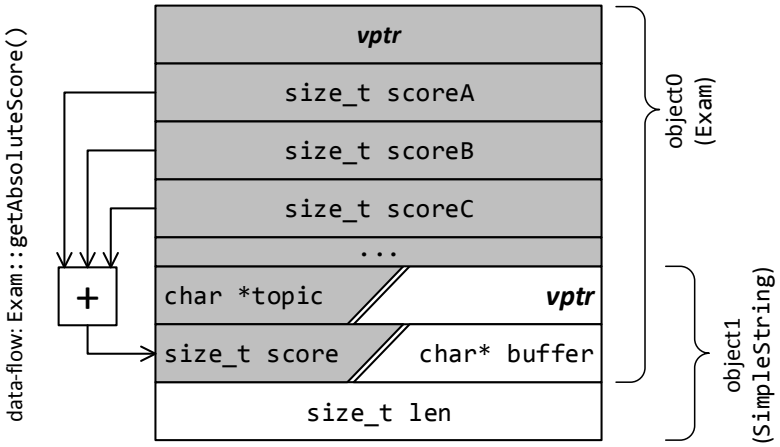


Figure 2.13: Overlapping counterfeit objects of types `Exam` and `SimpleString`

```
mov             rax, qword ptr [rcx+10h]
mov             r8, qword ptr [rcx+18h]
xorps           xmm0, xmm0
lea             rdx, [rax+rax*2]
mov             rax, qword ptr [rcx+8]
lea             rcx, [rax+rax*4]
lea             r9, [rdx+r8*2]
add             r9, rcx
cvtsi2ss        xmm0, r9
addss           xmm0, dword ptr [__real0]
divss           xmm0, dword ptr [__real1]
ret
```

Listing 2.6: x86-64 assembly code produced by MSVC for `Exam::getWeightedScore()` (example for a LOAD-R64-G)

of the actual number of arguments a callee takes. Accordingly, virtual functions often use `rdx`, `r8`, and `r9` as scratch registers and do not restore or clear them on returning. This circumstance makes passing arguments to vfgadgets simple on x86-64: first, a vfgadget is executed that loads one of the corresponding counterfeit object's fields into `rdx`, `r8`, or `r9`. Next, a vfgadget is executed that interprets the contents of these registers as arguments.

We refer to vfgadgets that can be used to load argument registers as LOAD-R64-G. For the x86-64 arguments passing concept to work, a ML-G is required that itself does not pass arguments to the invoked virtual functions/vfgadgets. Of course, the ML-G must also not modify the registers `rdx`, `r8`, and `r9` between such invocations. In our example, the attacker can control the source pointer `s` of the write operation (namely `strncpy()`) by invoking a LOAD-R64-G that loads `rdx` *before* `SimpleString::set()`.

As an example for a LOAD-R64-G, consider `Exam::getWeightedScore()` from Figure 2.12; MSVC compiles this function to the following assembly code shown in Listing 2.6. In condensed from, this LOAD-R64-G provides the following useful semantics to

the attacker:

$$\begin{aligned} \texttt{rdx} &\leftarrow 3 \cdot [\mathit{this} + 10h] \\ \texttt{r8} &\leftarrow [\mathit{this} + 18h] \\ \texttt{r9} &\leftarrow 3 \cdot [\mathit{this} + 18h] + 2 \cdot [\mathit{this} + 10h] \end{aligned}$$

Thus, by carefully choosing the fields at offsets $10h$ and $18h$ from the this-ptr of the corresponding counterfeit object, the attacker can write arbitrary values to the registers `rdx`, `r8`, and `r9`. Note that the attacker here also controls the registers `rax` and `rcx`. This is however of no value to her as `rax` is not an argument register and is thus virtually never read without having been initialized before in a function; and `rcx` is necessarily always updated by the ML-G to point to the next counterfeit object when a vfgadget returns.

In summary, to control the source pointer in the writing operation in `SimpleString::set()`, the attacker would first invoke `Exam::getWeightedScore()` for a counterfeit object carrying the desired source address divided by 3 at offset $10h$. This would load the desired source address to `rdx`, which would next be interpreted as the argument `s` in the vfgadget `SimpleString::set()`.

**Other Platforms** In the default x86-64 C++ calling convention used by GCC [129], e. g., on Linux, the first six arguments to a function are passed through registers instead of only the first four registers. In theory, this should make COOP attacks simpler to create on Linux x86-64 than on Windows x86-64, as two additional registers can be used to pass data between vfgadgets. In practice, during the creation of our example exploits (see Section 2.5.4), we did not experience big differences between the two platforms.

Although we did not conduct experiments on RISC platforms such as ARM or MIPS, we expect that our x86-64 approach directly extends to these because in RISC calling conventions arguments are also primarily passed through registers.

**Approach Windows x86-32** The standard x86-32 C++ calling convention on Windows is *thiscall* [136]: all regular arguments are passed over the stack whereas the this-ptr is passed in the register `ecx`; the callee is responsible for removing arguments from the stack. Thus, the described approach for x86-64 does not work for x86-32. In our approach for Windows x86-32, contrary to x86-64, we rely on a *main loop* (ML-G) that *passes* arguments to vfgadgets. More precisely, a 32-bit ML-G should pass one field of the *initial object* as argument to each vfgadget. In practice, any number of arguments may work; for brevity we only discuss the simplest case of *one* argument here. We call this field the *argument field* and refer to this variant of ML-G as ML-ARG-G. For an example of an ML-ARG-G, consider the virtual destructor of the class `Course2` in Figure 2.14: the field `id` is passed as argument to each invoked virtual function. Given such an ML-ARG-G, the attacker can employ one of the two following approaches to pass chosen arguments to vfgadgets:

**A-1** fix the *argument field* to point to a writable *scratch area*.

**A-2** dynamically rewrite the *argument field*.

```
class Student2 {
private:
   std::list<Exam> exams;
public:
   /* ... */
   virtual void subscribeCourse(int id) { /* ... */ }
   virtual void unsubscribeCourse(int id) { /* ... */ }

   virtual bool getLatestExam(Exam &e) {
      if (exams.empty()) return false;          W-SA-G
      e = exams.back();
      return true;                              W-COND-G
   }
};

class Course2 {
private:
   Student2 **students;
   size_t nStudents;
   int id;
public:
   /* ... */
   virtual ~Course2() {
      for (size_t i = 0; i < nStudents; i++)
         students[i]->unsubscribeCourse(id);
      delete students;                          ML-ARG-G
   }
};
```

Figure 2.14: Examples for W-SA-G, W-COND-G, ML-ARG-G

In approach **A-1**, the attacker relies on vfgadgets that interpret their first argument not as an immediate value but as a *pointer* to data. Consider for example the virtual function `Student2::getLatestExam()` from Figure 2.14 that copies an `Exam` object; MSVC produces the optimized x86-32 assembly code shown in Listing 2.7 for the function. In condensed form, lines 9–22 of the assembly code provide the following semantics:

$$
\begin{aligned}
[arg0 + 4] &\leftarrow [[[this + 4] + 4] + Ch] \\
[arg0 + 8] &\leftarrow [[[this + 4] + 4] + 10h] \\
[arg0 + Ch] &\leftarrow [[[this + 4] + 4] + 14h] \\
[arg0 + 10h] &\leftarrow [[[this + 4] + 4] + 18h]
\end{aligned}
$$

Note that for approach **A-1**, *arg0* always points to the *scratch area*. Accordingly, this vfgadget allows the attacker to copy 16 bytes (corresponding to the four 32-bit fields of `Exam`) from the attacker-chosen address $[[this + 4] + 4+] + Ch$ to the scratch area. We refer to this type of vfgadget that writes attacker-controlled fields to the scratch area as W-SA-G.

```
push    ebp
mov     ebp, esp
cmp     dword ptr [ecx+8], 0
jne     copyExam
xor     al, al
pop     ebp
ret     4
copyExam:
mov     eax, dword ptr [ecx+4]
mov     ecx, dword ptr [ebp+8]
mov     edx, dword ptr [eax+4]
mov     eax, dword ptr [edx+0Ch]
mov     dword ptr [ecx+4], eax
mov     eax, dword ptr [edx+10h]
mov     dword ptr [ecx+8], eax
mov     eax, dword ptr [edx+14h]
mov     dword ptr [ecx+0Ch], eax
mov     eax, dword ptr [edx+18h]
mov     dword ptr [ecx+10h], eax
mov     al, 1
pop     ebp
retn    4
```

Listing 2.7: Optimized x86-32 assembly code produced by MSVC for `Student2::getLatestExam()`

Using `Student2::getLatestExam()` as W-SA-G in conjunction with a ML-ARG-G allows the attacker, for example, to pass a string of up to 16 characters as first argument to the vfgadget `SimpleString::set()`.

In approach **A-2**, the argument field of the initial object is not fixed as in approach **A-1**. Instead, it is dynamically rewritten during the execution of a COOP attack. This allows the attacker to pass *arbitrary* arguments to vfgadgets; as opposed to *a pointer to arbitrary data* for approach **A-1**. For this approach, naturally, a usable W-G is required. As stated above, we found vfgadgets working solely with fields to be rare. Hence, the attacker would typically initially follow approach **A-1** and implement **A-2**-style argument writing on top of that when required.

**Passing Multiple Arguments and Balancing the Stack** So far, we have described how a single argument can be passed to each vfgadget using a ML-ARG-G main loop gadget on Windows x86-32. Naturally, it can be desirable or necessary to pass more than one argument to a vfgadget. Doing so is simple: the ML-ARG-G pushes one argument to each vfgadget. In case a vfgadget does not expect any arguments, the pushed argument remains on the top of the stack even after the vfgadget returned. This effectively moves the stack pointer permanently one slot up as depicted in Figure 2.15 ③. This technique allows the attacker to gradually "pile up" arguments on the stack as shown in Figure 2.15 ④ before invoking a vfgadget that expects multiple arguments. This technique only works for ML-ARG-Gs that use `ebp` and not `esp` to access local variables on the stack (i. e., no *frame-pointer omission*) as otherwise the stack frame of the ML-ARG-G is destroyed.

Analogously to how vfgadgets without arguments can be used to move the stack pointer *up* under an ML-ARG-G, vfgadgets with more than one argument can be used to move

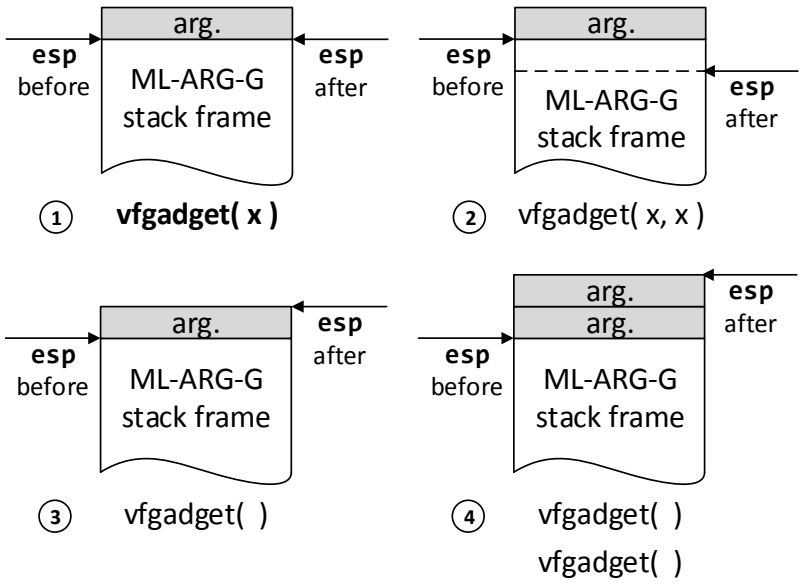


Figure 2.15: Examples for stack layouts *before* and *after* invoking vfgadgets under an ML-ARG-G (*thiscall* calling convention). The stack grows upwards. ① vfgadget with one argument: the stack is balanced. ② vfgadget with two arguments: `esp` is moved down. ③ vfgadget without arguments: `esp` is moved up. ④ two vfgadgets without arguments: two arguments are piled up.

the stack pointer *down* as shown in Figure 2.15 ②. This may be used to compensate for vfgadgets without arguments or to manipulate the stack. We refer to vfgadgets with little or no functionality that expect less or more than one argument as MOVE-SP-Gs. Ideally, a MOVE-SP-G is an empty virtual function that just adjusts the stack pointer.

**Other Platforms** The default x86-32 C++ calling convention used by GCC, e. g., on Linux, is not *thiscall* but *cdecl* [136]: all arguments including the this-ptr are passed over the stack; instead of the callee, the caller is responsible for cleaning the stack. The described technique of "piling up" arguments does thus not apply to GCC-compiled (and compatible) C++ applications on Linux x86-32 and other POSIX x86-32 platforms. Instead, for these platforms, we propose using ML-ARG-Gs that do not pass one but many controllable arguments to vfgadgets. Conceptually, passing too many arguments to a function does not corrupt the stack in the *cdecl* calling convention. Alternatively, ML-ARG-Gs could be switched during an attack depending on which arguments to a vfgadget need to be controlled.

#### 2.5.1.5 Calling API Functions

The ultimate goal of code-reuse attacks is typically to pass attacker-chosen arguments to critical API functions or system calls, e. g., WinAPI functions such as `WinExec()` or

`VirtualProtect()`. We identified the following ways to call a WinAPI function in a COOP attack:

**W-1** use a vfgadget that legitimately calls the WinAPI function of interest.

**W-2** invoke the WinAPI function like a virtual function from the COOP main loop.

**W-3** use a vfgadget that calls a C-style function pointer.

While approach **W-1** may be practical in certain scenarios and for certain WinAPI functions, it is unlikely to be feasible in the majority of cases. For example, virtual functions that call `WinExec()` should be close to non-existent.

Approach **W-2** is simple to implement: a counterfeit object can be crafted whose vptr does not point to an actual vtable but to the *import table* (IAT) or the *export table* (EAT) [171] of a loaded module such that the ML-G invokes the WinAPI function as a virtual function. Note that IATs, EATs, and vtables are all arrays of function pointers typically lying in read-only memory; they are thus in principle compatible data structures. As simple as it is, the approach has two important drawbacks: *(i)* it goes counter to our goal **G-2** as a C function is called at a vcall site without a legitimate vtable being referenced; and *(ii)* for x86-64, the this-ptr of the corresponding counterfeit object is always passed as the first argument to the WinAPI function due to the given C++ calling convention. This circumstance for example effectively prevents the passing of a useful command line to `WinExec()`. (`WinExec()` expects the pointer to an ASCII command line as first argument. In case a this-ptr is passed as first argument, the corresponding vptr is interpreted as command line which is likely not useful.) However, this can be different for other WinAPI functions. For example, calling `VirtualProtect()` with a this-ptr as first argument still allows the attacker to mark the memory of the corresponding counterfeit object as *executable*. Note that `VirtualProtect()` changes the memory access rights for a memory region pointed to by the first argument. Other arguments than the first one can be passed as described in Section 2.5.1.4 for x86-64. For x86-32, *all* arguments can be passed using the technique from Section 2.5.1.4.

For approach **W-3** a special type of vfgadget is required: a virtual function that calls a C-style function pointer with non-constant arguments. We refer to this type of vfgagdet as INV-G, an example is given in Figure 2.16: the virtual function `GuiButton::clicked()` invokes the field `GuiButton::callbackClick` as C-style function pointer. This particular vfgadget allows for the invocation of arbitrary WinAPI functions with at least three attacker-chosen arguments. Note that, depending on the actual assembly code of the INV-G, a fourth argument could possibly be passed through `r9` for x86-64. Additional stack-bound arguments for x86-32 and x86-64 may also be controllable depending on the actual layout of the stack.

Calling WinAPI functions through INV-Gs should generally be the technique of choice as this is more flexible than approach **W-1** and stealthier than **W-2**. An INV-G also enables seemingly legit transfers from C++ to C code (e. g., to libc) in general. On the downside, we found INV-Gs to be relatively rare overall. For our real-world example exploits discussed in Section 2.5.4, we could though always select from multiple suitable ones.

```
class GuiButton {
private:
      int id;
      void(*callbackClick)(int, int, int);
public:
      void registerCbClick(void(*cb)(int, int, int)) {
        callbackClick = cb;
      }

      virtual void clicked(int posX, int posY) {
            callbackClick(id, posX, posY);
      }                                                    INV-G
};
```

Figure 2.16: Example for INV-G: `clicked` invokes a field of `GuiButton` as C-style function pointer.

#### 2.5.1.6 Implementing Conditional Branches and Loops

Up to this point, we have described all building blocks required to practically mount COOP code-reuse attacks. As we do not only aim for COOP to be stealthy, but also to be *Turing-complete* under realistic conditions (goal **G-4**), we now describe the implementation of *conditional branches* and *loops* in COOP.

In COOP, the *program counter* is the index into the container of counterfeit object pointers. The program counter is incremented for each iteration in the ML-G's main loop. The program counter may be a plain integer index as in our exemplary ML-G `Course::~Course` or may be a more complex data structure such as an iterator object for a C++ linked list. Implementing a conditional branch in COOP is generally possible in two ways: through *(i)* a conditional increment/decrement of the program counter or *(ii)* a conditional manipulation of the next-in-line counterfeit object pointers in the container. Both can be implemented given a conditional write vfgadget, which we refer to as W-COND-G. An example for this vfgadget type is again `Student2::getLatestExam()` from Figure 2.14. As can be seen in lines 3–7 of the function's assembly code in Listing 2.7, the controllable write operation is only executed in case $[\textit{this-ptr}+8] \neq 0$. With this semantics, the attacker can rewrite the COOP program counter or upcoming pointers to counterfeit objects under the condition that a certain value is not null. In case the program counter is stored on the stack (e. g., in the stack frame of the ML-G) and the address of the stack is unknown, the technique for moving the stack pointer described in Section 2.5.1.4 can be used to rewrite it.

Given the ability to conditionally rewrite the program counter, implementing loops with an exit condition also becomes possible.

### 2.5.2 Loopless Counterfeit Object-oriented Programming

The basic COOP code-reuse attack technique as described in Section 2.5.1.3 inherently relies on a *main loop* vfgadget (ML-G). Accordingly, one can think of different possible (partial) defenses against COOP that make ML-Gs unavailable to an attacker or at least

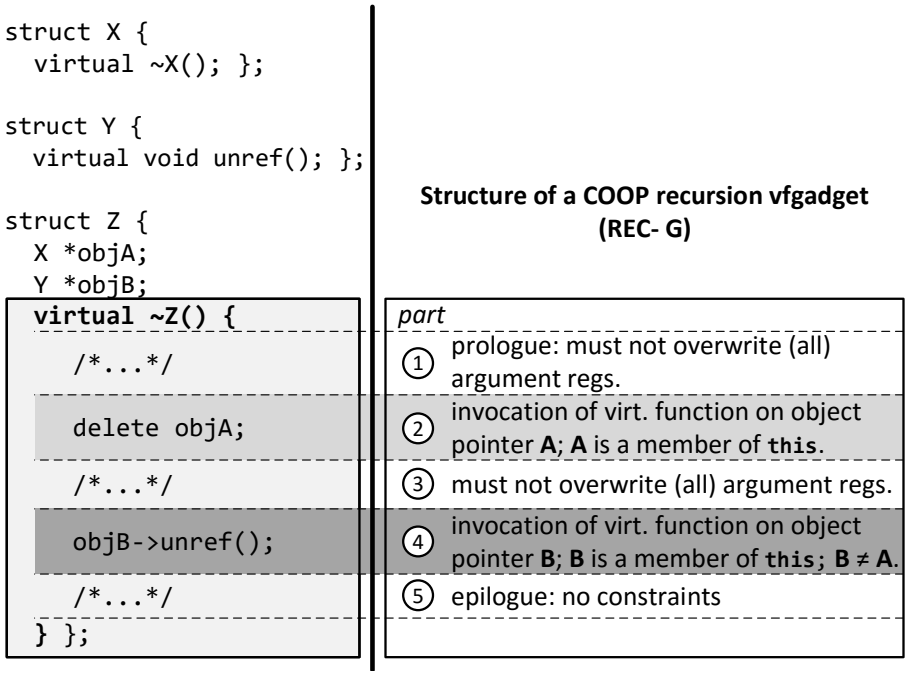


Figure 2.17: Example code (left) and general structure (right) of a REC-G

complicated to misuse. Yet, our observation is that the COOP concept is not necessarily bound to ML-Gs. In the following, we describe two refined versions of COOP that do not require ML-Gs and emulate the original *main loop* through *recursion* and *loop unrolling*. For brevity, we only discuss the x86-64 platform in the following.

Generally, all semantics that can programmatically be expressed through loops can also be expressed through recursive functions. This naturally also applies to COOP's main loop. We identified a certain code pattern that can commonly be found in virtual functions and is especially common within virtual destructors. This code pattern can be misused to emulate an ML-G by means of recursion. We refer to a virtual function that exhibits this pattern as *REC-G* (short for *recursion vfgadget*).

For an example of a REC-G, consider the C++ code in Figure 2.17: `Z::~Z()` is a typical (virtual) destructor. It deletes the object `objA` and removes a reference to `objB`. Consequently, a virtual function is invoked on both objects `objA` and `objB`. In case `Z::~Z()` is invoked on an adversary-controlled counterfeit object, the adversary effectively controls the pointers `*objA` and `*objB`. The adversary can make these pointers point to injected counterfeit objects.

Accordingly, `Z::~Z()` can be misused by an adversary to make two consecutive COOP-style vfgadget invocations. This, in turn, effectively enables the adversary to invoke an *arbitrary number* of vfgadgets, if the counterfeit object `objB` is shaped such that `Z::~Z()` is recursively invoked. The left side of Figure 2.18 schematically depicts the counterfeit object layouts that are required for this: for each regular counterfeit object, one additional *auxiliary counterfeit object* is required that resembles an object of class `Z`. Each auxiliary counterfeit object's `*objB` points to the next auxiliary counterfeit object (pointers ❷ and

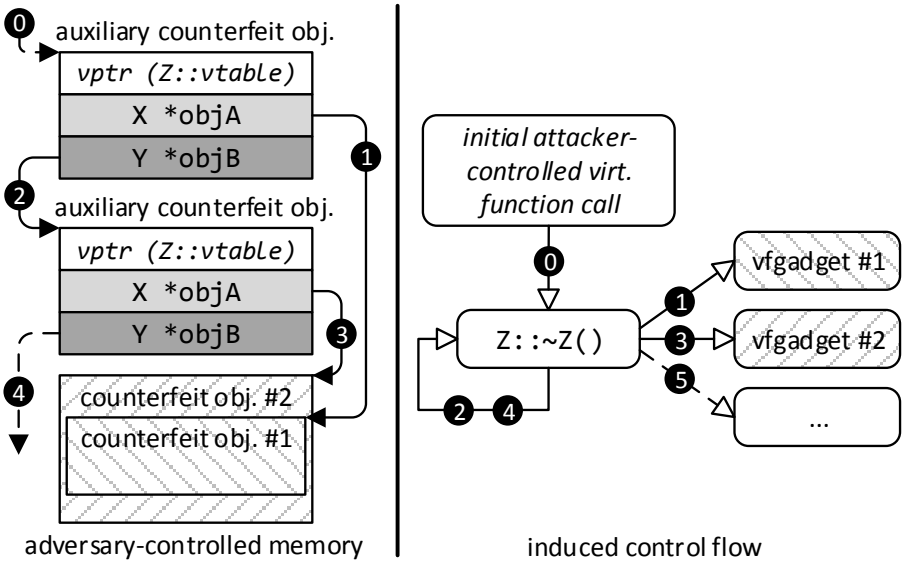


Figure 2.18: Schematic layout of adversary-controlled memory with pointers (left) and control-flow transitions (right) in a recursion-based COOP attack using `Z::~Z()` as REC-G

❹ Figure 2.18), whereas each `*objA` points to a regular counterfeit object that corresponds to a certain vfgadget (pointers ❶ and ❸). The right side of Figure 2.18 shows the resulting adversary-induced control flow.

We remark that not only destructors but any virtual function may qualify as REC-G. The required abstract structure of a REC-G is shown on the right side of Figure 2.17: as discussed, at least two invocations of virtual functions on distinct and adversary-controlled object pointers are required (parts ② and ④); the code before these invocations (parts ① and ③) must not write to registers that are required for passing arguments to vfgadgets. As per definition C++ destructors do not receive any explicit arguments, parts ① and ③ are particularly likely to not write to argument registers if the parts ② and ④ are both comprised of a `delete` statement.

**Unrolled COOP** Given a virtual function with not only *two* consecutive virtual function invocations (like a REC-G) but *many*, it is also possible to mount an *unrolled* COOP attack that does not rely on a loop or recursion. This COOP variant is detailed in a paper[4] by Crane et al. [59].

### 2.5.3 A Framework for Counterfeit Object-oriented Programming

Implementing a COOP attack against a given application is a three step process: *(i)* identification of vfgadgets, *(ii)* implementation of attack semantics using the identified vfgadgets, and *(iii)* arrangement of possibly overlapping counterfeit objects in a buffer.

[4]The author of this dissertation is also one of the co-authors of this paper.

Since the individual steps are cumbersome and hard to perform by hand, we created a framework in the Python scripting language that automates steps *(i)* and *(iii)*. This framework greatly facilitated the development of our example exploits for Internet Explorer and Chromium (see Section 2.5.4). In the following, we provide an overview of our implementation.

#### 2.5.3.1 Finding Vfgadgets Using Basic Symbolic Execution

For the identification of useful vfgadgets in an application, our *vfgadget searcher* relies on binary code only and optionally debug symbols. Binary x86-32 C++ modules are disassembled using the popular *Interactive Disassembler* (IDA) version 6.5. Each virtual function in a C++ module is considered a potential vfgadget. The searcher statically identifies all vtables in a C++ module using debug symbols or, if these are not available, a set of simple but effective heuristics is applied. Akin to other work [163, 235], our heuristics consider each address-taken array of function pointers a potential vtable. The searcher examines all identified virtual functions whose number of basic blocks does not exceed a certain limit. In practice, we found it sufficient and convenient to generally only consider virtual functions with one or three basic blocks as potential vfgadgets; the only exception being ML-Gs and ML-ARG-Gs, which due to the required loop often consist of more basic blocks. Using short vfgadgets is favorable as their semantics are easier to evaluate automatically and they typically exhibit fewer unwanted side effects. Including long vfgadgtes can, however, be necessary to fool heuristics-based code-reuse attack detection approaches (see Section 2.5.5).

The searcher summarizes the semantics of each basic block in a vfgadget in *single static assignment* (SSA) form. These summaries reflect the I/O behavior of a basic block in a compact and easy to analyze form. The searcher relies for this on the *backtracking* feature of the METASM binary code analysis toolkit [93], which performs symbolic execution on the basic-block level. An example of a basic block summary as used by our searcher was already provided in the listed semantics for the second basic block of `Exam::getWeightedScore()` in Section 2.5.1.4. To identify useful vfgadgets, the searcher applies filters on the SSA representation of the potential vfgadgets' basic blocks. For example, the filter: "*left side of assignment must dereference any argument register; right side must dereference the this-ptr*" is useful for identifying 64-bit W-Gs; the filter: "*indirect call independent of [this]*" is useful for finding INV-Gs; and the filter: "*looped basic block with an indirect call dependent on [this] and a non-constant write to [esp-4]*" can in turn be used to find 32-bit ML-ARG-Gs.

#### 2.5.3.2 Aligning Overlapping Objects Using an SMT Solver

Each COOP "program" is defined by the order and positioning of its counterfeit objects of which each corresponds to a certain vfgadget. As described in Section 2.5.1.3, the overlapping of counterfeit objects is an integral concept of COOP; it enables immediate data flows between vfgadgets through fields of counterfeit objects. Manually obtaining the alignment of overlapping counterfeit objects right on the binary level is a time-consuming and error-prone task. Hence, we created a COOP *programming environment* that auto-

matically, if possible, correctly aligns all given counterfeit objects in a fixed-size buffer. In our programming environment, the "programmer" defines counterfeit objects and labels. A label may be assigned to any byte within a counterfeit object. When bytes within different objects are assigned the same label, the programming environment takes care that these bytes are mapped to the same location in the final buffer, while assuring that bytes with different labels are mapped to distinct locations. Fields without labels are in turn guaranteed to never overlap. These constraints are often satisfiable, as actual data within counterfeit objects is typically sparse.

For example, the counterfeit object $A$ may only contain its vptr (at relative offset +0), an integer at the relative offset +16 and have the label $X$ for its relative offset +136; the counterfeit object $B$ may only contain its vptr and have the same label $X$ for its relative offset +8. Here, the object $B$ fits comfortably and without conflicts inside $A$ such that $B$ +8 maps to the same byte as $A$ +136.

Our programming environment relies on the Z3 SMT solver [67] to determine the alignment of all counterfeit objects within the fixed-size buffer such that, if possible, all label-related constraints are satisfied. At the baseline, we model the fixed-size buffer as an *array* mapping integers indexes to integers in Z3. To prevent unwanted overlaps, for each byte in each field, we add a *select* constraint [68] in Z3 of the form

$$select(\textit{offset-obj} + \textit{reloffset-byte}) = \textit{id-field}$$

where *offset-obj* is an integer variable to be determined by Z3 and *reloffset-byte* and *id-field* are constant integers that together uniquely identify each byte. For each desired overlap (e. g., between objects $A$ and $B$ using label $X$), we add a constraint of the form

$$\textit{offset-objA} + \textit{reloffset(A,X)} = \textit{offset-objB} + \textit{reloffset(B,X)}$$

where *offset-objA* and *offset-objB* are integers to be determined by Z3 and $\textit{reloffset(A,X)} = 136$ and $\textit{reloffset(B,X)} = 8$ are constants.

In the programming environment, for convenience, symbolic pointers to labels can be added to counterfeit objects. Symbolic pointers are automatically replaced with concrete values once the offsets of all labels are determined by Z3. This way, multiple levels of indirection can be implemented conveniently.

An example of a vfgadget that reads attacker-controlled data through multiple levels of indirection was provided in the W-SA-G `Student2::getLatestExam()` whose semantics are given in Section 2.5.1.4. The programming environment also contains templates for common object pointer container formats used in ML-Gs. For these common formats, the counterfeit object pointer container can be created automatically. The programming environment outputs a buffer that contains all counterfeit objects and is ready to be injected in a COOP attack.

### 2.5.4 Proof of Concept Exploits

To demonstrate the practical viability of our approach, we implemented exemplary COOP attacks for Microsoft Internet Explorer 10 (32-bit and 64-bit) and Google Chromium 41 for Linux x86-64. In the following, we discuss different aspects of our attack codes that we

find interesting. We used our framework described in Section 2.5.3 for the development of all three attack codes. Each of them fits into 1024 bytes or less.

For our Internet Explorer 10 examples, we used a publicly documented vulnerability related to an integer signedness error in Internet Explorer 10 [111] as foundation. The vulnerability allows a malicious website to perform arbitrary reads at any address and arbitrary writes within a range of approximately 64 pages on the respective heap using JavaScript code. This gives the attacker many options for hijacking C++ objects residing on the heap and injecting her own buffer of counterfeit objects; it also enables the attacker to gain extensive knowledge on the respective address space layout. We successfully tested our COOP-based exploits for Internet Explorer 10 32-bit and 64-bit on Windows 7. Note that our choice of Windows 7 as target platform is only for practical reasons; the described techniques also apply to Windows 8. To demonstrate the flexibility of COOP, we implemented different attack codes for 32-bit and 64-bit. Both attack codes could be ported to the respective other environment without restrictions.

#### 2.5.4.1 Internet Explorer 10 64-bit

Our COOP attack code for 64-bit only relies on vfgadgets contained in mshtml.dll that can be found in every Internet Explorer process; it implements the following functionality:

- read pointer to kernel32.dll from IAT.
- calculate pointer to `WinExec()` in kernel32.dll.
- read the current tick count from the `KUSER_SHARED_DATA` data structure.
- if tick count is odd, launch calc.exe using `WinExec()`;
- else, execute alternate execution path and launch mspaint.exe.

The attack code consists of 17 counterfeit objects with counterfeit vptrs and four counterfeit objects that are pure data containers. Overall eight different vfgadgets are used; including one LOAD-R64-G for loading `rdx` through the dereferencing of a field that is used five times. The attack code is based on a ML-G similar to our exemplary one given in Figure 2.9 that iterates over a plain array of object pointers. With four basic blocks, the ML-G is the largest of the eight vfgadgets. The conditional branch depending on the current tick count is implemented by overwriting the next-in-line object pointer such that the ML-G is recursively invoked for an alternate array of counterfeit object pointers. In summary, the attack code contains eight overlapping counterfeit objects and we used 15 different labels to create it in our programming environment. All vfgadgets used in this attack code are listed in Table 2.7.

**Attack Variant Using only Vptrs Pointing to the Beginning of Vtables** The described 64-bit attack code relies on counterfeit vptrs (see Section 2.5.1.3) that do not necessarily point to the beginning of existing vtables but to positive or negative offset from them. As a proof of concept, we developed a stealthier variant of the attack code above that *only* uses vptrs that point to the beginning of existing vtables. Accordingly, at each vcall

| Symbol name of vfgadget | # | Type | Purpose |
|---|---|---|---|
| `CExtendedTagNamespace::Passivate` | 1, 9b | ML-G | array-based main loop |
| `CCircularPositionFormatField-Iterator::Next` | 2, 5, 7, 9a, 10b | LOAD-R64-G | load `rdx` from dereferenced field |
| `XHDC::SetHighQualityScaling-Allowed` | 3 | ARITH-G | store `rdx`&1 |
| `CWigglyShape::OffsetShape` | 4 | LOAD-R64-G | load `r9` from field |
| `CStyleSheetArrayVarEnumerator::-MoveNextInternal` | 6 | LOAD-R64-G | load `r8` from field |
| `CDataCache<class CBoxShadow>::-InitData` | 8 | W-COND-G | write `r8` to `[rdx]` if `r9` is not zero |
| `CRectShape::OffsetShape` | 10a, 11b | ARITH-G | add `[rdx]` to field |
| `Ptls6::CLsBlockObject::Display` | 11a, 12b | INV-G | invoke field as function pointer |

Table 2.7: Vfgadgets in mshtml.dll *10.0.9200.16521* used in Internet Explorer 10 64-bit exploit; execution splits into paths *a* and *b* after index 8.

| Symbol name of vfgadget | # | Type | Purpose |
|---|---|---|---|
| `CExtendedTagNamespace::Passivate` | 1 | ML-G | array-based main loop |
| `CMarkupPageLayout::IsTopLayout-Dirty` | 2, 4 | LOAD-R64-G | load `edx` from field |
| `HtmlLayout::GridBoxTrack-Collection::GetRangeTrackNumber` | 3 | ARITH-G | `r8` = 2 · `rdx` |
| `CAnimatedCacheEntryTyped<float>-::UpdateValue` | 4 | INV-G | invoke field from argument as function pointer |

Table 2.8: Vfgadgets in mshtml.dll *10.0.9200.16521* used in exemplary Internet Explorer 10 64-bit exploit that only uses vptrs pointing to the beginning of existing vtables

site, we were restricted to the set of virtual functions compatible with the respective fixed vtable index. Under this constraint, our exploit for the given vulnerability is still able to launch calc.exe through an invocation of `WinExec()`. The attack code consists of only five counterfeit objects, corresponding to four different vfgadgets (including the main ML-G) from mshtml.dll. Corresponding to the given vulnerability, the used main ML-G can be found as fourth entry in an existing vtable whereas, corresponding to the vcall site of the ML-G, the other three vfgadgets can be found as third entries in existing vtables. The task of calculating the address of `WinExec` is done in JavaScript code beforehand. All vfgadgets used in this attack code are listed in Table 2.8.

```
mov     edi, edi
push    ebp
mov     ebp, esp
push    ecx
push    ecx
push    esi
mov     esi, ecx
lea     eax, [esi+3ACh]
; -- inlined constructor of iterator --
mov     [ebp+iterator.end], eax
mov     [ebp+iterator.current], eax
; --

loop:
lea     ecx, [ebp+iterator]
call    SListBase::Iterator::Next()
test    al, al
jnz     end

mov     eax, [ebp+iterator.current]
push    [esi+140h] ; push argument field
mov     ecx, [eax+4] ; read object pointer from iterator
mov     eax, [ecx]
call    [eax+4] ; call 2nd virtual function
jmp     loop

end:
pop     esi
mov     esp, ebp
pop     ebp
ret
```

Listing 2.8: Assembly code of ML-ARG-G in jscrip9.dll version *10.0.9200.16521* used in exemplary Internet Explorer 10 32-bit exploit: a linked list of object pointers is traversed; a virtual function with one argument is invoked on each object.

#### 2.5.4.2 Internet Explorer 10 32-bit

Our 32-bit attack code implements the following functionality: *(1)* read pointer to kernel32.dll from IAT; *(2)* calculate pointer to `WinExec()` in kernel32.dll; *(3)* enter loop that launches calc.exe using `WinExec()` $n$ times; *(4)* finally, enter an infinite waiting loop such that the browser does not crash.

The attack code does not rely on an array-based ML-ARG-G (recall that in 32-bit ML-ARG-Gs are used instead of ML-Gs); instead, it uses a more complex ML-ARG-G that traverses a linked list of object pointers using a C++ iterator. We discovered this ML-ARG-G in jscript9.dll that is available in every Internet Explorer process. The ML-ARG-G consists of four basic blocks and invokes the function `SListBase::Iterator::Next()` to get the next object pointer from a linked list in a loop. The assembly code of the ML-ARG-G is given in Listing 2.8.

Figure 2.19 depicts the layout of the linked list: each item in the linked list consists of one pointer to the next item and another pointer to the actual object. This layout allows for the low-overhead implementation of conditional branches and loops. For example, to implement the loop in our attack code, we simply made parts of the linked list circular

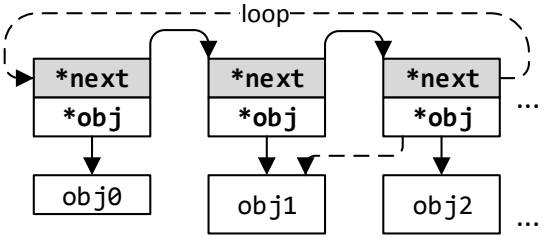


Figure 2.19: Schematic layout of the linked list of object pointers the ML-ARG-G traverses in the Internet Explorer 10 32-bit exploit; dashed arrows are examples for dynamic pointer rewrites for the implementation of conditional branches.

```
size_t SkComposeShader::contextSize() const {
    return sizeof(ComposeShaderContext)
            + fShaderA->contextSize() + fShaderB->contextSize();
}
```

Listing 2.9: Example of a REC-G in Chromium 41 (C++)

as shown in Figure 2.19. Inside the loop in our attack code, a counter within a counterfeit object is incremented for each iteration. Once the counter overflows, a W-COND-G rewrites the *backward* pointer such that the loop is left and execution proceeds along another linked list. Our attack code consists of 11 counterfeit objects, and 11 linked list items of which two point to the same counterfeit object. Four counterfeit objects overlap and one counterfeit object overlaps with a linked list item to implement the conditional rewriting of a *next* pointer. The actual vfgadgets used in our attack code are listed in Table 2.9. This example highlights how powerful linked list-based ML-Gs/ML-ARG-Gs are in general.

#### 2.5.4.3 Chromium 41 for Linux x86-64

To demonstrate the wide applicability of COOP, we also created an attack code for a modified version of Chromium 41 for Linux x86-64. This specific version was compiled with LLVM and was altered to contain the critical vulnerability CVE-2014-3176 [5], which had been identified and patched in an earlier version of Chromium. Our COOP attack code here reads a pointer to libc.so from the *global offset table* (GOT) and calculates the address of `system()` from that in order to finally invoke `system("/bin/sh")`.

The attack code is comprised of six counterfeit objects (of which two overlap) corresponding to six different vfgadgets from Chromium's main executable module. The vfgadgets are listed in detail in Table 2.10.

We also created a *loopless* variant of this COOP attack code that, instead of an ML-G, uses a REC-G from Chromium 41 which is depicted in Listing 2.9.

In this REC-G, `fShaderA->contextSize()` constitutes part ② and `fShaderB->contextSize()` part ④ as depicted in Figure 2.17 in Section 2.5.2.

[5] See `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-3176`

| Symbol name of vfgadget | # | Type | Purpose |
|---|---|---|---|
| `jscript9!ThreadContext-::ResolveExternalWeak-ReferencedObjects` | 1 | ML-ARG-G | linked list-based main loop |
| `CDataTransfer::Proxy` | 2 | W-SA-G | write deref. field to scratch area |
| `CDCompSwapChainLayer::Set-DesiredSize` | 3 | R-G | load field from scratch area |
| `CDCompSurfaceTargetSurface-::GetOrigin` | 4 | ARITH-G / W-SA-G | write summation of two fields to scratch area |
| `CDCompLayerManager::Set-AnimationCurveToken` | 5 | R-G | load field from scratch area |
| `HtmlLayout::SvgBoxBuilder::-PrepareBoxForDisplay` | *loop_entry:* 6, 11 | W-G | rewrite *argument field* |
| `CDXTargetSurface::OnEndDraw` | 7, 8 | MOVE-SP-G | move stack pointer up |
| `ieframe!Microsoft::WRL::-Callback::ComObject::Invoke` | 9 | INV-G | invoke function pointer with 2 arguments |
| `CMarkupPageLayout::Add-LayoutTaskOwnerRef` | 10 | ARITH-G | increment field |
| `Ptls6::CLsDnodeNonText-Object::SetDurFmtCore` | 12 | W-COND-G | conditionally write argument to field; rewrites linked list; resumes at *loop_entry* or *loop_exit* |
| `CDispRecalcContext::-OnBeforeDestroyInitial-IntersectionEntry` | *loop_exit* | NOP | nop; loops to self |

Table 2.9: Vfgadgets used in Internet Explorer 10 32-bit exploit; vfgadgets taken from mshtml.dll (if not marked differently), jscript9.dll, or ieframe.dll version *10.0.9200.16521*.

### 2.5.5 Discussion

We now analyze the properties of COOP, discuss different defense concepts against it, and review our design goals **G-1**–**G-4** from Section 2.5.1.1. The effectiveness against COOP of several existing defenses is discussed afterwards in Section 2.5.6.

#### 2.5.5.1 Preventing COOP

We observe that the characteristics **C-1**–**C-5** of existing code-reuse attack approaches cannot be relied on to defend against COOP (goal **G-1**): in COOP, control flow is only dispatched to existing and address-taken functions within an application through existing indirect calls. In addition, COOP does neither inject new nor alter existing return addresses as well as other code pointers directly. Instead, only existing vptrs (i. e., pointers to code pointers) are manipulated or injected. Technically, depending on the choice of vfgadgets, a COOP attack may however execute a high ratio of indirect branches and thus exhibit characteristic **C-3**. But we note that ML-Gs (which are used in each COOP attack

| Symbol name of vfgadget | # | Type | Purpose |
|---|---|---|---|
| `icu_52::PatternMap::~PatternMap` | 1 | ML-G | array-based main loop |
| `SkBlockMemoryStream::rewind` | 2 | R-G / W-G | read pointer to libc and write it to field |
| `TraceBufferRingBuffer::ClonedTraceBuffer::NextChunk` | 3 | LOAD-R64-G | load `rsi` with offset of `system()` |
| `net::AeadBaseEncrypter::GetCiphertextSize` | 4 | ARITH-G | add field to `rsi` |
| `TtsControllerImpl::SetPlatformImpl` | 5 | W-G | store `rsi` |
| `browser_sync::AddDBThreadObserverTask::RunOnDBThread` | 6 | INV-G | invoke function pointer from field and pass field as argument |

Table 2.10: Vfgadgets used in Chromium 41 64-bit Linux exploit

as central dispatchers) are legitimate C++ virtual functions whose original purpose is to invoke many (different) virtual functions in a loop. Any heuristics attempting to detect COOP based on the frequency of indirect calls will thus inevitably face the problem of high numbers of false positive detections. Furthermore, similar to existing attacks against behavioral-based heuristics [65,91], it is straightforward to mix-in long "dummy" vfgadget to decrease the ratio of indirect branches.

As a result, COOP cannot be effectively prevented by *(i)* CFI that does not consider C++ semantics or *(ii)* detection heuristics relying on the frequency of executed indirect branches and is unaffected by *(iii)* shadow call stacks that prevent rogue returns and *(iv)* the plain protection of code pointers.

On the other hand, a COOP attack can only be mounted under the preconditions given in Section 2.5.1.2. Accordingly, COOP is conceptually thwarted by defense techniques that prevent the hijacking or injection of C++ objects or conceal necessary information from the attacker, e. g., by applying ASLR and preventing information leaks.

#### 2.5.5.2 Generic Defense Techniques

We now discuss the effectiveness of several other possible defensive approaches against COOP that do not require knowledge of precise C++ semantics and can thus likely be deployed without analyzing an application's source code or recompiling it.

**Restricting the Set of Legitimate API Invocation Sites** A straightforward approach to tame COOP attacks is to restrict the set of code locations that may invoke certain sensitive library functions. For example, by means of binary rewriting it is possible to ensure that certain WinAPI functions may only be invoked through constant indirect branches that read from a module's IAT (see *CCFIR* [236]). In the best case, this approach could effectively prevent the API calling techniques **W-2** and **W-3** described in Section 2.5.1.5. However, it is also common for benign code to invoke repeatedly used or dynamically resolved WinAPI functions through non-constant indirect branches like `call`

`rsi`. Accordingly, in practice, it can be difficult to precisely identify the set of a module's legitimate invocation sites for a given WinAPI function. We also remark that even without immediate access to WinAPI functions or systems calls COOP is still potentially dangerous, because, for example, it could be used to manipulate or leak critical data.

**Monitoring of the Stack Pointer** In 64-bit COOP, the stack pointer is virtually never moved in an irregular or unusual manner. For the 32-bit *thiscall* calling convention though, this can be hard to avoid as long as not only vfgadgets with the same fixed number of arguments are invoked. This is a potential weakness that can reveal a COOP attack on Windows x86-32 to a C++-unaware defender that closely observes the stack pointer. However, we note that it may be difficult to always distinguish this behavior from the benign invocation of functions in the *cdecl* calling convention.

#### 2.5.5.3 Fine-grained Code Randomization

COOP is conceptually resilient against the fine-grained randomization of locations of binary code, e. g., on function, basic block, or instruction level. This is because in a COOP attack, other than for example in a ROP attack, knowing the exact locations of certain instruction sequences is not necessary but rather only the locations of certain vtables. Moreover, in COOP, the attacker mostly misuses the *actual* high-level semantics of existing code. Most vfgadget types, other than ROP gadgets, are thus likely to be unaffected by semantics-preserving rewriting of binary code. Only LOAD-R64-Gs that are used to load x86-64 argument registers could be broken by such means. However, the attacker could probably oftentimes fall back to x86-32-style ML-ARG-G-based COOP in such a case.

**C++ Semantics-aware Defense Techniques** We observe that the control flow and data flow in a COOP attack are similar to those of benign C++ code (goal **G-2**). However, there are certain deviations that can be observed by C++-aware defenders. We now discuss several corresponding defenses.

**Verification of Vptrs** In basic COOP, vptrs of counterfeit objects point to existing vtables but not necessarily to their beginning. This allows for the implementation of viable defenses against COOP when all legitimate vcall sites and vtables in an application are known and accordingly each vptr access can be augmented with sanity checks. Such a defense can be implemented without access to source code by means of static binary code rewriting as concurrently shown by Prakash et al. [163]. While such a defense significantly shrinks the available vfgadget space, our exploit code from Section 2.5.4.1 demonstrates that COOP-based attacks are still possible, at least for large C++ target applications.

Ultimately, a defender needs to know the set of allowed vtables for each vcall site in an application to reliably prevent malicious COOP control flow (or at least needs to arrive at an approximation that sufficiently shrinks the vfgadget space). For this, the defender needs *(i)* to infer the global hierarchy of C++ classes with virtual functions and *(ii)* to determine the C++ class (within that hierarchy) that corresponds to each vcall site. Both can easily be achieved when source code is available. Without source code, given only

binary code and possibly debug symbols or RTTI metadata[6], the former can be achieved with reasonable precision while, to the best of our knowledge, the latter is generally considered to be hard for larger applications by means of static analysis [72, 80, 85, 163].

**Monitoring of Data Flow** COOP also exhibits a range of data-flow patterns that can be revealing when C++ semantics are considered. Probably foremost, in basic COOP, vfgadgtes with varying number of arguments are invoked from the same vcall site. This can be detected when the number of arguments expected by each virtual function in an application is known. While trivial with source code, deriving this information from binary code can be challenging [163]. An even stronger (but also likely costlier) protection could be created by considering the actual types of arguments.

In a COOP attack, counterfeit objects are not created and initialized by legitimate C++ constructors, but are injected by the attacker. Further, the concept of overlapping objects creates unusual data flows. To detect this, the defender needs to be aware of the life-cycle of C++ objects in an application. This requires knowledge of the whereabouts of (possibly inlined) constructors and destructors of classes with virtual functions.

**Fine-grained Randomization of C++ Data Structures** In COOP, the layout of each counterfeit object needs to be byte-compatible with the semantics of its vfgadget. Accordingly, randomizing C++ object layouts on application start-up, e. g., by inserting randomly sized paddings between the fields of C++ objects, can hamper COOP. Also, the fine-grained randomization of the positions or structures of vtables is a viable defense against COOP. In fact, we pursued this approach in a joint paper with Crane et al. [59] as is described in more detail in Section 2.5.6.4.

We conclude that COOP can be mitigated by a range of means that do not require knowledge of C++ semantics. But we regard it as vital to consider and to enforce C++ semantics to reliably prevent COOP. Doing so by means of static binary analysis and rewriting only is challenging as the compilation of C++ code is in most cases a *lossy* process. For example, in binary code, distinguishing the invocation of a virtual function from the invocation of a C-style function pointer that happens to be stored in a read-only table can be difficult. Hence, unambiguously recovering essential high-level C++ semantics afterwards can be hard or even impossible. In fact, as we discuss in more detail in Section 2.5.6, we know of no binary-only CFI solution that considers C++ semantics precisely enough to fully protect against COOP.

#### 2.5.5.4 Applicability and Turing Completeness

We have shown that COOP is applicable to popular C++ applications on different operating systems and hardware architectures (goal **G-3**). Naturally, a COOP attack can only be mounted in case at least a minimum set of vfgadgets is available. We did not conduct a quantitative analysis on the general frequency of usable vfgadgets in C++ applications: determining the actual usefulness of potential vfgadgets in an automated way

[6] *Runtime Type Information* (RTTI) metadata is often linked into C++ applications for various purposes. RTTI includes the literal names of classes and the precise class hierarchy.

is challenging and we leave this for future work. In general, we could choose from many useful vfgadgets in the libraries mshtml.dll (around 20 MB) and libxul.so (around 60 MB) and found the basic vfgadget types ARITH-G, W-G, R-G, LOAD-R64-G, and W-SA-G to be common even in smaller binaries.

The availability of central dispatcher vfgadgets such as ML-Gs/ML-ARG-Gs or REC-Gs is vital to every COOP attack. While especially ML-Gs/ML-ARG-Gs are generally sparser than the more basic types, we found well-usable dispatcher vfgadgets gadgets, e. g., in Microsoft's standard C/C++ runtime libraries msvcr120.dll and msvcp120.dll (both smaller than 1 MB; dynamically linked to many C and C++ applications on Windows): the virtual function `SchedulerBase::CancelAllContexts()` with five basic blocks in msvcr120.dll is a linked list-based ML-G. In msvcr120.dll, we also found the INV-G `Cancellation-TokenRegistration_TaskProc::_Exec()` that consists of one basic block and is suitable for x86-32 and x86-64 COOP.

The virtual function `propagator_block::unlink_sources()` with eight basic blocks in msvcp120.dll is an array-based ML-ARG-G. Interestingly, this particular ML-ARG-G is also defined in Visual Studio's standard header file agents.h. The virtual destructor of the class `Concurrency::_Order_node_base<enum Concurrency::agent_status>` with seven basic blocks in msvcp120.dll is a REC-G.

Given the vfgadget types defined in Table 2.6, COOP has the same expressiveness as unrestricted ROP [188]. Hence, it allows for the implementation of a Turing machine (goal **G-4**) based on memory load/store, arithmetic, and branches. In particular, the COOP examples in Section 2.5.4 show that complex semantics like loops can be implemented under realistic conditions.

### 2.5.6 Security Assessment of Existing Defenses

Based on the discussions in Section 2.5.5, we now assess a selection of contemporary defenses against code-reuse attacks and discuss whether they are vulnerable to COOP in our adversary model. A summary of our assessment is given in Table 2.11.

#### 2.5.6.1 Generic CFI

We first discuss CFI approaches that do not consider C++ semantics for the derivation of the CFG that should be enforced. We observe that all of them are vulnerable to COOP.

The basic implementation of the original CFI work by Abadi et al. [1] instruments binary code such that indirect calls may only go to address-taken functions (imprecise CFI). As already discussed in Section 2.2.5.2, this scheme and a closely related one [238] have recently been shown to be vulnerable to advanced ROP-based attacks [65,90]. Abadi et al. also proposed to combine their basic implementation with a shadow call stack that prevents call/return mismatches. This extension effectively mitigates these advanced ROP-based attacks while, as discussed in Section 2.5.5, it does not prevent COOP. The same applies in general also to the recently proposed Lockdown system [156]. However, besides a shadow call stack and standard imprecise CFI policies, Lockdown additionally enforces that across modules only mutually *imported/exported* functions may be invoked indirectly. Accordingly, an COOP attack would for instance be limited to those functions

| Category | Scheme | Realization | Effective? |
|---|---|---|---|
| Generic CFI | Original CFI + shadow call stack [1] | Binary + debug symbols | ✗ |
| | Lockdown [156] | Binary + debug symbols | ✗ |
| | CFI for COT [238] | Binary | ✗ |
| | CCFIR [236] | Binary | ✗ |
| | O-CFI [216] | Binary | ✗ |
| | MIP [145] | Source code | ✗ |
| | SW-HW Co-Design [64] | Source code + CPU features | ✗ |
| | Windows 10 CFG [206] | Source code | ✗ |
| | LLVM IFCC [211] | Source code | ? |
| C++-aware CFI | **various** [8, 110, 211] | Source code | ✓✓✓ |
| | T-VIP [85] | Binary | ✗ |
| | VTint [235] | Binary | ✗ |
| | vfGuard [163] | Binary | ? |
| Heuristics-based detection | **various** [54, 152, 228, 239] | Binary + CPU features | ✗✗✗ |
| | Microsoft EMET 5 [83, 134] | Binary | ✗ |
| Code hiding, shuffling, or rewriting | STIR [221] | Binary | ✗ |
| | G-Free [148] | Source code | ✗ |
| | Readactor [58] | Source code + CPU features | ✗ |
| | XnR [22] | Binary/source code + CPU features | ? |
| | Readactor++ [59] | Source code + CPU features | ✓ |
| Memory safety | **various** [7–9, 52, 143, 187] | Mostly source code | (✓✓✓) |
| | CPI/CPS [122] | Source code | ✓/✗ |

Table 2.11: Overview of the effectiveness of a selection of code-reuse defenses and memory safety techniques (below double line) against COOP; ✓ indicates effective protection and ✗ indicates vulnerability; **?** indicates at least partial protection.

from kernel32.dll or libc that are actually used by the target application. We remark that this import/export policy probably cannot generally be applied to C++ virtual functions without the risk of high rates of false positives. This is because it is not uncommon for a C++ module to unknowingly access a vtable defined in another module when dynamically dispatching a virtual function call. In such a case, a virtual function that is neither *exported* nor *imported* is legitimately invoked across module boundaries.

Davi et al. described a hardware-assisted CFI solution for embedded systems that incorporates a shadow call stack and a certain set of runtime heuristics [64]. However, the indirect call policy only validates whether an indirect call targets a valid function start. As COOP only invokes entire functions, it can bypass this hardware-based CFI mechanism.

CCFIR [236], a CFI approach for Windows x86-32 binaries, uses a randomly arranged "springboard" to dispatch all indirect branches within a code module. On the baseline, CCFIR allows indirect calls and jumps to target all address-taken locations in a binary and restricts returns to certain call-preceded locations. One of CCFIR's core assumptions is that the attacker is unable to "[...] selectively reveal [s]pringboard stub addresses of their choice" [236]. Göktaş et al. recently showed that ROP-based bypasses for CCFIR are possible given an up-front information leak from the springboard [90]. In contrast,

COOP breaks CCFIR without violating its assumptions: the springboard technique is ineffective against COOP as we do not inject code pointers but only vptrs (pointers to code pointers). CCFIR though also ensures that sensitive WinAPI functions (e. g., `CreateFile()` or `WinExec()`) can only be invoked through constant indirect branches. However, as examined in Section 2.5.5.2, this measure does not prevent dangerous attacks and can probably also be sidestepped in practice. In any case, COOP can be used in the first stage of an attack to selectively readout the springboard.

In *Monitor Integrity Protection* (MIP) [145], applications are compiled such that they are composed of variable-sized chunks: single instructions, basic blocks, or functions that do not include calls (*leaf functions*). Indirect branches are instrumented in such a way that they can only lead to the beginning of chunks. It is claimed that MIP can "[...] prevent arbitrary code execution" for an attacker that is able to read/write arbitrary data in an application's address space but cannot directly write to the processor's registers. Since COOP only invokes legitimate virtual functions it will never trigger an alarm in MIP.

Many system modules in the Microsoft Windows 10 Technical Preview are compiled with *Control Flow Guard* (CFG) [206], a simple form of CFI. In summary, Microsoft CFG ensures that protected indirect calls may only go to a certain set of targets. This set is specified in a module's PE header [171]. If multiple CFG-enabled modules reside in a process, their sets are merged. At least all functions contained in a DLL's EAT are contained in the set. For C++ modules like `mshtml.dll`, additionally, all virtual functions are contained in the set and can thus be invoked from any indirect call site. Accordingly, Microsoft CFG in its current form does not prevent COOP, but also likely not advanced ROP-based attacks like the one by Göktaş et al.

Tice et al. recently described two variants of *Forward-Edge CFI* for the GCC and LLVM compiler suites [211] that solely aim at constraining indirect calls and jumps but not returns. As such, taken for itself, forward-edge CFI does not prevent ROP in any way. One of the proposed variants is the C++-aware *virtual table verification* (VTV) technique for GCC. It tightly restricts the targets of each vcall site according to the C++ class hierarchy and thus prevents COOP. VTV is available in mainline GCC since version *4.9.0*. However, the variant for LLVM called *indirect function-call checks* (IFCC) "[...] does not depend on the details of C++ or other high-level languages" [211]. Instead, each indirect call site is associated with a set of valid target functions. A target is valid if *(i)* it is address-taken and *(ii)* its signature is *compatible* with the call site. Tice et al. discuss two definitions for the *compatibility* of function signatures for IFCC: *(i)* all signatures are compatible or *(ii)* signatures with the same number of arguments are compatible. We observe that the former configuration does not prevent COOP, whereas the latter can still allow for powerful COOP-based attacks in practice as discussed in Section 2.5.5.3.

#### 2.5.6.2 C++-aware CFI

As discussed in Section 2.5.5, COOP's control flow can be reliably prevented when precise C++ semantics are considered from source code. Accordingly, various source code-based CFI solutions exist that prevent COOP, e. g., GCC VTV as described above, Safedispatch [110], or WIT [8].

Recently, three C++-aware CFI approaches for legacy binary code were proposed: T-VIP [85], vfGuard [163], and VTint [235]. They follow a similar basic approach:

1. identification of vcall sites and vtables (only vfGuard and VTint) using heuristics and static data-flow analysis
2. instrumentation of vcall sites to restrict the set of allowed vtables

T-VIP ensures at each instrumented vcall site that the vptr points to read-only memory. Optionally, it also checks if a random entry in the respective vtable points to read-only memory. Similarly, VTint copies all identified vtables into a new read-only section and instruments each vcall site to check if the vptr points into that section. Both effectively prevent attacks based on the injection of fake vtables, but as in a COOP attack only actual vtables are referenced, they do not prevent COOP. VfGuard instruments vcall sites to check if the vptr points to the *beginning* of any known vtable. As discussed Section 2.5.5.3, such a policy restricts the set of available vfgadgets significantly, but still cannot reliably prevent COOP. VfGuard also checks the compatibility of calling conventions and consistency of the this-ptr at vcall sites, but this does not affect COOP. Nonetheless, we consider vfGuard to be one of the strongest available binary-only defenses against COOP. VfGuard significantly constraints attackers and we expect it to be a reliable defense in at least some attack scenarios, e. g., for small to medium-sized x86-32 applications that are considerably smaller than Internet Explorer.

#### 2.5.6.3 Heuristics-based Detection

Microsoft EMET [134] is probably the most widely deployed exploit mitigation tool. Among others, it implements different heuristics-based strategies for the detection of ROP [83]. Additionally, several related heuristics-based defenses have been proposed that utilize certain debugging features available in modern x86-64 processors [54, 152, 228]. All of these defenses have recently been shown to be unable to detect more advanced ROP-based attacks [47, 65, 91, 180]. Similarly, the HDROP [239] defense utilizes the *performance monitoring counters* of modern x86-64 processors to detect ROP-based attacks. The approach relies on the observation that a processor's internal branch prediction typically fails in abnormal ways during the execution of common code-reuse attacks.

As discussed in Section 2.5.5.1, such heuristics are unlikely to be practically applicable to COOP and we can in fact confirm that our Internet Explorer exploits (Section 2.5.4.1 and Section 2.5.4.2) are not detected by EMET version 5.

#### 2.5.6.4 Code Hiding, Shuffling, or Rewriting

STIR [221] is a binary-only defense approach that randomly reorders basic blocks in an application on each start-up to make the whereabouts of gadgets unknown to an attacker—even if she has access to the exact same binary. As discussed in Section 2.5.5.2, approaches like this do conceptually not affect our attack, as COOP only uses entire functions as vfgadgets and only knowledge on the whereabouts of vtables is required. This applies also to the recently proposed O-CFI approach [216] that combines the STIR concept with coarse-grained CFI.

G-Free [148] is an extension to the GCC compiler. G-Free produces x86-32 native code that (largely) does not contain *unaligned* indirect branches. Additionally, it aims to prevent attackers from misusing *aligned* indirect branches: return addresses on the stack are encrypted/decrypted on a function's entry/exit and a "cookie" mechanism is used to ensure that indirect jump/call instructions may only be reached through their respective function's entry. While effective even against many advanced ROP-based attacks [47, 65, 90, 91, 180], G-Free does not affect COOP.

The Execute-no-Read (XnR) concept [22] prevents an application's code pages from being read at runtime in order geared to hamper so-called *JIT-ROP* [194] attacks. We note that, depending on the concrete scenario, a corresponding JIT-COOP attack could not always be thwarted by such measures as it can suffice to readout vtables and possibly RTTI metadata (which contains the literal names of classes) from data sections and apply pattern matching to identify the addresses of the vtables of interest. XnR is meant to be implemented in hardware as a complementary feature to the execute-disable/NX bit already available in modern processors.

The Readactor system [58] leverages the *Extended Page Tables* (EPT) [107] feature of modern x86-64 processors to place an application's code in *execute-only* memory [7] at runtime. In Readactor, a C/C++ application is compiled such that *(i)* all its actual code pointers are hard-coded inside *trampolines* in execute-only memory, *(ii)* only pointers to those trampolines but no actual code pointers—including return addresses—are stored in readable memory at runtime, and *(iii)* the binary code layout is randomized in a fine-grained manner. Consequently, the whereabouts of all an application's code—except for trampolines—are concealed at runtime even from attackers that can read the entire address space with respect to page permissions. Crane et al. claim that Readactor "[...] provides protection against all known variants of ROP attacks [...]" [58]. However, we observe that the Readactor concept does conceptually not hinder COOP, because Readactor neither hides vtables in any special way nor randomizes their layouts. Vptrs also receive no special treatment from Readactor.

Finally, Readactor++ [8] [59] is an extension of the Readactor concept that was specifically designed to tackle COOP and RILC. Readactor++ applies all the defensive measures of Readactor and also pseudo-randomly changes the structure layout of vtables (and also other function pointer tables) as outlined in Section 2.5.5.3. In order to exacerbate attempts at guessing useful vtable entries, Readactor++ also adds so-called "booby trap" entries to randomized vtables that on execution terminate the protected application. It is argued that a minimal COOP attack, which requires at least the execution of three vfgadgets from distinct vtables, would be hindered by Readactor++ with a chance of at least 99.97%. Readactor++'s average overhead is low and it can be considered one of the most cost-effective strong defenses against COOP.

[7] Across different contemporary processor architectures, if memory is *executable*, then typically it is implicitly also *readable*.

[8] The author of this dissertation is a co-author of the paper on Readactor++.

#### 2.5.6.5 Memory Safety

Systems that provide forms of memory safety for C/C++ applications [7–9,52,122,143,187] can constitute strong defenses against control-flow hijacking attacks in general. As our adversary model explicitly foresees an initial memory corruption and information leak (see Section 2.5.1.2), we do not explore the defensive strengths of these systems in detail. Instead, we exemplarily discuss two recent approaches in the following.

Kuznetsov et al. proposed *Code-Pointer Integrity* (CPI) [122] as a low-overhead control-flow hijacking protection for C/C++. On the baseline, CPI guarantees the spatial and temporal integrity of code pointers and, recursively, that of pointers to code pointers. As in C++ applications typically many pointers to code pointers exist (essentially each object's vptr), CPI can still impose a significant overhead there. As a consequence, Kuznetsov et al. also proposed *Code-Pointer Separation* (CPS) as a less expensive variant of CPI that specifically targets C++. In CPS, sensitive pointers are not protected recursively, but it is still enforced that "[...] (i) code pointers can only be stored to or modified in memory by code pointer store instructions, and (ii) code pointers can only be loaded by code pointer load instructions from memory locations to which previously a code pointer store instruction stored a value" [122] where *code pointer load/store instructions* are fixed at compile time. Kuznetsov et al. argue that the protection offered by CPS could be sufficient in practice as it conceptually prevents recent advanced ROP-based attacks [47, 65, 91]. We observe that CPS does not prevent our attack, because COOP does not require the injection or manipulation of code pointers. In the presence of CPS, it is though likely hard to invoke library functions not imported by an application. But we note that almost all applications import critical functions. The invocation of library functions through an INV-G could also be complicated or impossible in the presence of CPS. This is however not a hurdle, because, as CPS does not consider C++ semantics, imported library functions can always easily be called without taking the detour through an INV-G as described in Section 2.5.1.5 in approach **W-2**.

CPS can straightforwardly be made resilient to COOP by extending the protection of immediate code pointers to C++ vptrs. In fact, recent implementations of CPS incorporate this tweak [121].

## 2.6 Conclusion

In this chapter, we gave an overview of the ongoing battles fought around the exploitation and the mitigation of memory errors. Subsequently, we presented our own contributions to the arms race in the form various novel code-reuse attacks that break with common assumptions and in the consequence bypass different contemporary defenses in realistic adversarial settings.

First, we examined the practical effectiveness of three heuristics-based defenses against ROP—namely kBouncer, ROPGuard, and ROPecker. We discussed how all of them can reliably detect and prevent legacy exploits and demonstrated in turn how they can still be bypassed in generic ways with little effort. Our results show how heuristics-based detections are conceptually vulnerable to attackers who are aware of their presence and who can adapt their strategy accordingly. On a side note, an interesting insight is that

kBouncer and ROPecker rely on a custom kernel driver and employ complicated detection techniques build upon the LBR feature of modern processors. They though fall short to supply significantly higher protection levels than the much simpler ROPGuard. Our experimental results also hint at kBouncer and ROPecker being more prone to false positive attack detections than ROPGuard.

Next, we described *counterfeit object-oriented programming* (COOP), a completely new form of code-reuse attack. We discussed the technical details behind COOP and sketched and evaluated possible defenses against it. We also performed a security assessment of a broad range of existing defenses: COOP bypasses almost all imprecise CFI solutions and also defenses from other categories that do not consider object-oriented C++ semantics. Our most important is insight that higher-level programming language-specific semantics need to be taken into account. This is a valuable guide for the design and implementation of future defenses. In particular, our results with COOP demand for a rethinking in the assessment of defenses that rely solely on the instrumentation of binary code.

Overall, we believe that our results contribute to the ongoing research on designing practical and secure defenses against control-flow hijacking and code-reuse attacks. In particular, they show that many defenses that are believed to be “good enough” are in fact not.

Chapter 3

# Towards the Mitigation of Backdoors in Software

Backdoors in software probably exist since the very first access control mechanisms were implemented and they are a well-known security problem. Despite a wave of public discoveries of backdoors in the recent past, this threat has only rarely been tackled so far.

This chapter explores the Backdoor adversarial setting as introduced in Chapter 1. It presents an approach to reduce the attack surface for backdoors and strives for an automated identification and elimination of these in (stripped) binary server applications. At its core, this approach applies variations of the *delta debugging* technique and relies on a set of heuristics for the identification of those regions in binary applications that backdoors are typically installed in, i. e., authentication and command processing functions. The practical feasibility of the approach is demonstrated on real-world backdoors found in modified versions of the server applications ProFTPD and OpenSSH and also on software running on embedded devices powered by a MIPS32 processor.

We first introduce the adversarial setting considered in this chapter (Section 3.1) and give a further motivation for our research on backdoors and summarize our results (Section 3.2). We detail our approach and specifically describe the essential A-WEASEL algorithm in Section 3.3. Building upon this, our implementation of the approach in form of the tool Weasel is described in Section 3.4 and evaluated on real and artificial backdoors in Section 3.4. This chapter closes with a review of related work (Section 3.6) and a final conclusion (Section 3.7).

## 3.1 Adversarial Setting

In this section, we introduce the Backdoor adversarial setting that is considered in the remainder of the chapter. The setting is similar to the Classic setting examined in Chapter 2 inasmuch as that a remote attacker is considered that interacts with the external interface of a server application. However, a different focus is applied here: we leave aside memory error vulnerabilities, which are usually the result of careless programming.

Instead, we concern with *backdoors* that were purposely installed in software. Intuitively, we define the term backdoor as follows:

**Definition:** *A backdoor is a hidden, undocumented, and unwanted program modification that on certain external triggers performs unwanted or malicious actions.*

Naturally there are unlimited ways how an attacker can implement a backdoor in a given software. As such, the line between a backdoor and a conventional vulnerability/bug can be blurry. For instance, the infamous *Heartbleed* bug [157], in essence a reliably exploitable spatial memory error, was by some believed to have been purposely and not carelessly added to the source code of OpenSSL. (The term *bugdoor* is sometimes used to refer to such perceived bug/backdoor hybrids.)

In general, the problem of detecting software backdoors is undecidable; among others, it encompasses the problem of detecting memory error vulnerabilities. Thus we limit the focus for remainder of this chapter on two specific types of backdoors:

**B-1** flawed authentication routines

**B-2** hidden commands and features

In the simplest cases, a flawed authentication routine (**B-1**) may accept a *magic* password and a hidden command (**B-2**) may give system access to unauthenticated users. Whereas, for example, a more elaborate combination of type **B-1** and type **B-2** backdoors may record the credentials of legitimate users and leak them upon external request over a covert channel. We assume a realistic and powerful attacker. The attacker aims at server application software running on trusted hosts and proceeds in two steps.

**Step 1: Installation of a Backdoor** In the *first step*, the attacker installs one or multiple backdoors of types **B-1** or **B-2** in a server application. For example, the attacker may be able to gain write-access to the online code repository of a software, for instance by remotely exploiting a memory corruption vulnerability in the repository software (see Chapter 2). If the attacker's malicious addition goes unnoticed by the company or the community that owns and maintains the respective software, then chances are that the backdoor is at one point deployed in production environments [185]. In a similar scenario, an attacker may compromise a system and install a backdoor in a local server application such that the she may stealthily return to the system later on [76].

Another common scenario is the case of a malicious insider, e. g., a programmer on the payroll of a software company, who adds a backdoor alongside her regular changes to the code. In such a case, the intent for adding a backdoor is not necessarily of malicious nature; instead, the backdoor could for example be an undocumented debugging interface that remains activated in the final release build of the software [139]. Similarly, backdoors may also manifest in the form of purposely installed and possibly well-intentioned hidden entrances [95, 184, 192, 233], e. g., for the purpose of convenient remote customer support.

**Step 2: Triggering of a Backdoor** Typically, backdoors are designed to stay dormant until, in the *second step* of an attack, they are triggered by certain events. The nature of these triggers can be diverse. In the simplest cases, backdoors of types **B-1** and **B-2** are

```
if (strcmp(target,"ACIDBITCHEZ") == 0)
{
  setuid(0);
  setgid(0);
  system("/bin/sh;/sbin/sh");
}
```

Listing 3.1: Backdoor in ProFTPD server

triggered by an attacker supplying a *magic* password or command respectively. However, more sophisticated trigger mechanisms are of course possible. For example, a backdoor may only activate at a certain time of day, for a certain remote IP address, after a certain port knocking sequence, or in case a certain process ID was assigned to the modified server application. This diversity of triggers is what makes the problem of detecting and containing backdoors especially challenging.

### 3.1.1 Running Example

As a running example, to further concretize the discussion, we illustrate a backdoor that attackers added to the ProFTPD FTP server software. This example highlights the challenges we face and explains some of the issues we have to deal with. Note that the example is in C code, while we perform our analysis on the binary level.

At the end of November 2010, the distribution server of the ProFTPD project was compromised and a snippet of code was added to one of the source files of ProFTPD 1.3.3c [185]. In essence, a function responsible for the processing of the standard FTP command HELP was modified in such a way that passing the argument `ACIDBITCHEZ` would result in immediate privileged access to the corresponding system (a *root shell*) for an unauthenticated user. The actual malicious code introduced by the attacker is shown in Listing 3.1. The backdoor was (likely manually) detected about three days later and then eliminated by removing the changes.

## 3.2 Research Motivation and Contributions

For users of software—consumers and corporations alike—the threat of backdoors is real and troubling as they typically lack the expertise or the resources to thoroughly review the software they employ. Furthermore, software is often distributed in binary form and even if source code is available, backdoors may still only materialize in the compiled binary version of a program; e. g., even a provably harmless software modification on source level may, as the result of a bug in a compiler, morph into a functional backdoor on binary code level [25]. Hence, automated approaches towards the mitigation of backdoors in binary software are very desirable. However, maybe surprisingly, the area of software backdoors has received relatively little attention from the research community so far, whereas the detection (and also the design) of malicious circuits in hardware has been addressed in quite a few works in recent years (e. g., [5, 27, 99, 106, 116, 166, 219, 220]).

In this chapter, we address the problem of software backdoors in server applications and introduce an automated way to detect and to disable backdoors of types **B-1** and **B-2** in a given binary server application. We stress that we do not aim to catch *all* backdoors, as this is impossible. Instead, our goal is to automate the process of dismantling certain notorious forms of backdoors that are today typically discovered by accident or through tedious manual analysis of binary code.

### 3.2.1 Approach Overview

Our novel approach comprises three phases: In the *first phase*, we identify the specific regions in a given binary that are especially prone to attacks, i. e., the authentication routines and code related to command dispatching and command handling functionality. These are the two components with which a remote attacker can interact and thus we view them as the most relevant attack target. We closely monitor the server application while it processes inputs automatically generated according to a protocol specification and analyze the resulting control-flow traces. This way, we can spot the program parts of interest in a precise and automated way. We leverage the idea of *delta debugging/differential analysis* [79, 234] and introduce an algorithm to identify the relevant regions of a given binary application.

In the *second phase*, once the initial components of interest are identified, we use this knowledge to determine suspicious components in an application, e. g., hidden command handlers or edges not taken in the CFG of an authentication routine. To this end, we introduce several heuristics that enable us to determine which code regions are suspicious. Furthermore, we aggregate information that can serve as a starting point for further automated (or manual) investigations.

In an optional *third phase*, the results from the previous phases can be used to modify or instrument the server application in an automated way such that program parts identified as suspicious are monitored or disabled at runtime. In particular, the analysis results often can be used to selectively reduce the functionality of a server application. This can for example be useful in cases where certain commands are suspected or known to be unsafe.

### 3.2.2 Results

We have implemented our approach to detect and to disable software backdoors in server applications in a tool called Weasel [1]. For the recording of traces and other runtime analyses, Weasel employs GDB, the standard debugger for the GNU software system. GDB is available for a large number of operating systems and hardware architectures and Weasel currently contains adapter code for x86-32, x86-64 and MIPS32 processor architectures running Linux operating systems. This enables the analysis of a wide range of platforms.

We have successfully tested Weasel with seven different server applications on different platforms including a widespread corporate VoIP telephone and a popular consumer router. In all cases, we were able to precisely and automatically identify the key program parts involved in the authentication process or the dispatching and handling of commands.

[1] The source code of our tool is available online: `https://github.com/flxflx/weasel`

This demonstrates our ability to not only analyze common instruction set architectures such as x86-32/x86-64, but also on commercial off-the-shelf embedded devices powered by a MIPS32 processor. We were able to detect known real-world backdoors contained in certain versions of ProFTPD and OpenSSH [76, 185]. Furthermore, as a case study, graduate students were tasked with the implementation of eleven different kinds of backdoors for ProFTPD. Our tool can be used to detect or disable the majority of these artificial backdoors as well. This demonstrates the practical feasibility of the approach to reduce the attack surface for software backdoors, but also highlights cases of software backdoors where additional research is needed.

To demonstrate a potential mitigation approach, WEASEL is also capable of transforming a given binary application to reduce the set of available commands. This is implemented by precisely identifying the command dispatcher functions leveraging automated data structure identification methods [123, 193]. Once we have found the specific data structures, we modify them such that certain commands are inaccessible.

## 3.3 Approach

We now describe our approach in detail before we present implementation details in the next section. Throughout the following, we will refer to our running example: the code snippet introduced in Listing 3.1 in Section 3.1.1. This code snippet was added as a malicious backdoor to the code base of the ProFPTD server software for the FTP protocol. The central goal of our approach is to automatically identify such harmful extensions in binary code.

### 3.3.1 Identifying Backdoors in Binary Code

Malicious additions to binary software can often be reliably detected by means of static analysis when a trusted benign version of the software is available: typically, the deployment of a simple backdoor introduces a handful of additional basic blocks to the CFG of a function that handles external inputs. Figure 3.1 depicts the malicious addition to the CFG of the function `pr_help_add_response()` in ProFTPD that constitutes the backdoor of our running example. In such a case, binary software comparison approaches [79, 84, 157, 158] are likely capable to detect these suspicious code regions. Unfortunately, in practice we typically do not have access to such a trusted benign version for most software systems. This is why our approach does not rely on the existence of such a version. Instead, we examine the behavior of a single version of binary software at runtime and apply techniques that extend the idea of Zeller's *delta debugging* approach [234] on binary level as explained in the following.

#### 3.3.1.1 Basic Analysis Approach

We argue that there are four parts of a server application that are generically prone to the two backdoor classes in our focus. These parts are:

- authentication validation code


Figure 3.1: The two additional basic blocks in the CFG of the function *pr_help_add_response()* in ProFTPD that implement the `ACIDBITCHEZ` backdoor

- specific authentication validation result handling code (e. g., code that terminates a session in case of invalid credentials)
- command parsing code
- specific command handling code

To identify these parts in a given server application, we record runtime traces for multiple different inputs and compare them. The intuition behind this approach is the following: consider for example the authentication mechanism of a server application such as ProFTPD. By definition, the purpose of each authentication mechanism is to decide whether or not a user sufficiently proved its identity to qualify for elevation of privilege.

The process is similar in case of command dispatching: different operations are performed for different commands and arguments. In order to behave differently for different inputs, a server application in general needs at one point during runtime to leave the common execution path and follow an exclusive execution path accordingly[2]. By comparing control flow traces for various inputs, it is possible to determine *common* execution paths and those that are *exclusive* to a certain group of inputs. In the next step, it is often possible to determine *deciders* and *handlers* on function or on basic-block level. In the case of the authentication process, *deciders* perform the actual authentication validation,

[2]One could probably draw scenarios where the different features of a server application are entirely implemented by differences in the data flow only and not in the control flow. Due to the lack of real-world relevance of such scenarios, they are not considered further here.

while *handlers* process the validation result. In the case of the command dispatching process, *deciders* parse and dispatch commands, while *handlers* implement their specific functionality.

In general, the following possibilities exist for a server application to implement exclusive execution paths for different inputs:

C-1 through exclusive function invocations

C-2 through exclusive paths inside commonly invoked functions (i. e., exclusive basic blocks)

C-3 without exclusive program parts, but through an exclusive execution order of common functions and basic blocks

In our empirical studies of different server applications we found that the two cases C-1 and C-2 are by far the most common in practice. Case C-3 is not entirely unlikely to be encountered, though: consider for example a server application implementing the available commands in an internal scripting language. Runtime traces both on function and on basic-block level for different inputs of such a server application would differ, but possibly not contain any exclusive program parts.

Cases C-1 and C-2 will receive the most attention for the rest of this chapter. An intuitive and straightforward approach for the identification of *handlers* and *deciders* in these cases is the following: Given two traces $T_0$ and $T_1$ for different inputs (e. g., valid password and invalid password), *handlers* can be identified by determining the set of exclusive functions/basic blocks for each of the two traces: $S_{T_0,ex} = S_{T_0} \backslash S_{T_1}$ and $S_{T_1,ex} = S_{T_1} \backslash S_{T_0}$. In turn, the corresponding *deciders* are necessarily in $S_{common} = S_{T_0} \cap S_{T_1}$ and are likely parents of exclusive functions/basic blocks in $S_{ex} = S_{T_0,ex} \cup S_{T_1,ex}$.

Built upon this basic idea for the identification of *deciders* and *handlers*, we have developed the algorithm A-WEASEL which is described in the following.

### 3.3.2 The A-WEASEL Algorithm

The A-WEASEL algorithm is an integral part of our approach. Before describing it in detail, we provide a high-level overview. The algorithm starts working on function level traces only since they can be collected on any platform where GDB is available in a straightforward and efficient way. Basic-block level traces are collected as needed.

Given a set of traces of a server application on function level for different *protocol runs* (e. g., traces for different FTP commands), we recursively compute a combined *decision tree* composed of *deciders* and top-level *handlers* as depicted in Figure 3.2. Handlers are initially identified by simply determining the set $S_{T,ex} = S_T \backslash S_{common}$ of exclusive functions for each trace and can be shared between multiple traces. For each identified decider (or handler shared between multiple traces) on function level, traces on basic-block level are recorded dynamically for all corresponding protocol runs[3]. Given these basic-block traces, we compute the *internal decision tree* of the respective function. For example in Figure 3.2,

[3]We record basic-block level traces of functions in a call stack sensitive manner. This way we can ensure that only the certain invocations of interest of a function are examined on basic-block level.

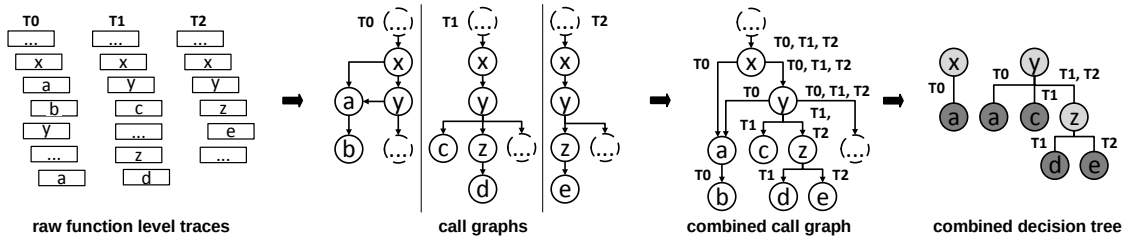


Figure 3.2: Schematic derivation of the *decision tree* (right) from the exemplary function level traces $T1$, $T2$ and $T3$ (left). The functions a, b, c, d, e and z are not contained in all traces and are thus *exclusive* to certain traces. The functions x, y and z are *deciders*, while a, c, d and e are top-level *handlers*.

the internal decision tree of the decider function **z** is generated from corresponding basic-block level traces for protocol runs 1 and 2. A-WEASEL is recursively invoked with the applying set of function level sub-traces for each identified decider function.

#### 3.3.2.1 Detailed Description of A-WEASEL

Given a set of $n$ traces on function level the A-WEASEL algorithm recursively performs the following steps:

1 Determine the set of functions present in all $n$ traces:

$$S_{common,funcs} = S_{T_0,funcs} \cap S_{T_1,funcs} \cap ... \cap S_{T_{n-1},funcs}$$

2 For each trace, determine the set of exclusive functions:

$$S_{T_i,ex,funcs} = S_{T_i,funcs} \backslash S_{common,funcs}$$

3 For each exclusive function in each set $S_{T_i,ex,funcs}$, determine the minimum number of call stack levels needed to distinguish between all of its invocations in the call graph (CG) of $T_i$. We use the term *signature call-stack* to refer to the call stack that unambiguously identifies an invocation. A new set $S_{T_i,ex,funcs,callstack}$ containing all invocations with different signature call-stacks of all functions in $S_{T_i,ex,funcs}$ is created:

$$S_{T_i,ex,funcs,callstack} = \delta(S_{T_i,ex,funcs})$$

4 For each set $S_{T_i,ex,funcs,callstack}$, remove those invocations of exclusive functions from the set that are *dominated* by other exclusive functions in the CG of the corresponding trace $T_i$:

$$S_{T_i,ex,funcs,top} = \varphi(S_{T_i,ex,funcs,callstack})$$

Thus only top-level invocations of exclusive functions of $T_i$ are contained in the set.

5 Group all remaining invocations in all sets $S_{T_i,ex,funcs,top}$ according to their signature call-stacks. Invocations with compatible signature call-stacks are grouped together. Two signature call-stacks are compatible if both are equal or are equal up to the end of one of the two call stacks. Note how each group only corresponds to one specific exclusive function and can at most contain one specific invocation from each trace.

6 The immediate parent function in the common call stack of a group's exclusive function is necessarily in set $S_{common,funcs}$ and is added as *decider* to the *decision tree*. Note that several groups can also share a common decider function. In case a group consists of only a single invocation from a single trace, the corresponding exclusive function is added as *handler* to the decision tree. Recursion ends in this case.

7 For each group, dynamically trace the corresponding *decider* function for the group's common signature call-stack for all applicable protocol runs on basic-block level. From the recorded basic-block traces, the internal *decision tree* of the decider function for the signature call-stack is generated by a similar but simpler algorithm.

8 For each group recursively execute A-WEASEL. For each invocation belonging to the group, a self-contained and minimal sub-trace $T_i'$ is cut from the original trace $T_i$ that starts with the corresponding signature call-stack. A-WEASEL is executed on the set of all such sub-traces corresponding to the group. I.e., A-WEASEL is executed on the sub-CGs of the *decider* identified for the group for all applying traces.

9 In case a *decider* function is found to be a leaf in the resulting decision tree and does not exhibit any control flow differences on basic-block level for applicable protocol runs, it is transformed into a common *handler* function.

### 3.3.3 Refining the Output of A-WEASEL

A-WEASEL reliably determines the decision tree for a given set of traces. However, it is only applicable for the aforementioned server application implementation case C-1 or a combination of cases C-1 and C-2. This is due to the algorithm depending on the identification of exclusive handlers: if there are no functions that are exclusive to a certain subset of all traces, no handlers can be identified (and as a consequence, no deciders as well). In order to cope with server applications that are implemented strictly according to cases C-2 and C-3 and also to improve overall results, we implemented a set of additional algorithms in WEASEL that are outlined in the following.

#### 3.3.3.1 Differential Return Value Comparison

A major drawback of A-WEASEL is that it fails to identify *decider* and *handler* functions when decisions made at runtime only manifest in differences in the control flow on basic-block level. To cope with this problem, WEASEL records exactly two traces for each protocol run. Functions with different return values in both traces are added to the set of functions to ignore. This way, irrelevant functions in our context such as `malloc()` or

`time()` are filtered out. Functions with the same return value in both traces, but different return values between different protocol runs, are treated as handlers by A-WEASEL. By applying this technique, A-WEASEL for example correctly identifies the function `sys_auth_passwd()` as decider in the authentication process of OpenSSH (see Section 3.5.1.1), which it would have not otherwise. Though there are numerous ways a function can signal its outcome to its caller, the described technique only takes immediate return values of functions into account.

#### 3.3.3.2 Scoring Heuristics

WEASEL contains a set of simple heuristic scoring algorithms that aim at identifying *deciders* and estimating their importance by comparing the structure of the call graphs of a given set of traces. The algorithms are used to rank the importance of *deciders* identified by A-WEASEL. Besides, they can partly serve as a fall back when server applications are encountered that do not implement exclusive functionality through exclusive function invocations (cases C-2 and C-3 as discussed above). All algorithms have in common that they require a *reference trace* which the other traces are compared to. When examining the authentication process, the trace for valid credentials serves as reference trace, while in the case of the command dispatching process a specially recorded trace for a knowingly invalid command serves this purpose.

For instance, one algorithm attempts to remove loops from all traces, determines the longest common subsequence of calls between all traces and the reference trace and assigns scores to functions according to their positions in the common sub-sequences (scores increase towards the end as those functions are believed to be more likely to be responsible for the decision to split execution paths). An even simpler algorithm assigns scores to deciders linear to the amount of exclusive children. The scores assigned to functions by the different algorithms are summed up.

### 3.3.4 Application of Analysis Results

Once the *deciders* as well as the *handlers* are known for the authentication or the command dispatching process of a server application, further analysis can be conducted. The goal is to identify possible backdoors and to enable the hardening of legacy binary applications.

#### 3.3.4.1 Discovering Suspicious Program Paths

When the functions or basic blocks handling a successful authentication or a certain command are known, we can apply existing methods of static and dynamic binary analysis for the detection of backdoors. A straightforward approach which we apply here is the static enumeration and comparison of all system calls and external library calls reachable in the static call graph from identified handlers (we utilize the third party tool *IDA Pro* for this). For example, even invocations of `socket()` or `send()` should be considered suspicious when they are only referenced from one of multiple handlers. In the case of our running example, the installed backdoor in the HELP command can be identified this way (see Section 3.5.2 for a detailed discussion).

Moreover, identified deciders and handlers can be used as starting points for more complicated analysis techniques such as *symbolic execution* [115]. Starting symbolic execution at the entry-point of identified handler code should in many cases—and especially for complex software such as ProFTPD—deliver better and more cost-effective results than approaches examining an entire application. In order to be able to apply techniques of symbolic execution, one of course always needs to declare certain memory as symbolic. Identifying memory regions that are worthwhile to declare as symbolic poses a challenge when examining single functions. To tackle this problem, we have implemented an analysis module for Weasel that compares the arguments of identified deciders in different traces and heuristically searches for differences (see Section 3.4.2.2). In the case of a typical password validation function it is then for example possible to determine that certain arguments are pointers to memory regions with varying contents (e. g., username and password). In the next step, these memory regions could be marked as symbolic when analyzing the respective function with symbolic execution techniques.

#### 3.3.4.2 Disabling Functionality

One can very well think of scenarios where it is desirable to disable certain functionality of a legacy server application. For example it might be known that certain commands are vulnerable to attacks. Instead of shutting the whole service down or applying error-prone filtering on the network level, our approach allows for the disabling of single commands by means of binary instrumentation or binary rewriting. In the case of our running example, effective protection can already be achieved by simply writing an illegal instruction at the start of the handler for the HELP command, causing the respective fork of the server to crash and exit when the command is issued.

**Cutting Cold Edges** Backdoors in the authentication process, like for example hard-coded credentials, often manifest in additional edges and nodes in the CFG of one of the involved decider functions. These additional edges and nodes are usually not contained in any recorded basic-block trace for legitimate input. We call such edges "cold". For complex software, cold edges and nodes are only a rather weak indication for the presence of a backdoor, as there are usually many benign basic blocks that are only visited under rare conditions. Nevertheless, knowledge of cold edges in decider functions of the authentication process can be used to increase the protection level of applications: techniques for binary instrumentation or rewriting can for example be used to log access to edges identified as *cold* during runtime over a longer period. In case an edge is taken for the first time, an alert can be triggered and the incident can be investigated. In practice, we suggest the utilization of a training phase during which additional benign paths are discovered and successively enabled before the final rewriting/instrumenting takes place. Entirely disabling cold edges in decider functions might severely weaken the security of an application, e. g., the protection against password brute-forcing could be rendered non-functional. In the following, we use the term of "cutting an edge" in order to refer to the monitoring or disabling of an edge.

**Elimination of Undocumented Commands** Another application of our approach is the identification and elimination of undocumented commands. Command deciders of server applications of classes C-1 or C-2 dispatch recognized commands either through conditional and direct (e. g., `jz <offset>`) or indirect (e. g., `call eax`) branches to their designated handlers. The latter is the case for our running example: when a command is recognized in ProFTPD, a C structure describing it is loaded from a static table. Each such structure contains a pointer to the handler function for the corresponding command, which is called by the decider/dispatcher through a `call eax` instruction. For server applications built in a similar way, two interesting measures become possible on top of our basic approach:

- Once several command handlers are known, a likely location and size of the table(s) holding the command-describing structures in memory can be determined. In the next step, it is possible to identify all available commands and unwanted commands can easily be eliminated using techniques of binary instrumentation or rewriting.
- Once the point of execution is known where the control flow is dynamically transferred to a command handler, techniques of binary instrumentation or rewriting can be used to prevent the execution of unknown or unwanted command handlers.

We have developed a module for our analysis framework that heuristically checks for tables containing command descriptors given a set of pointers to command handlers (see Section 3.4.2).

#### 3.3.4.3 Enforcing Authentication

When deciders and handlers of the authentication process of an application can be linked to certain authentication levels, it becomes possible to determine the authentication level of an active session by examining the control flow of the corresponding thread at runtime. Combined with the knowledge of whereabouts of command handlers, fine-grained access control or defense mechanisms such as *shadow authentication* [62] can be realized. In our running example, it would be possible to limit the availability of the HELP command to those threads that were observed successfully authenticating before.

## 3.4 Implementation

The core of the analysis framework Weasel is our library PyGdb. It is written for Python 2.7 and implements a client for the *GDB Remote Serial Protocol* [197] and thus needs to be connected to a remote gdbserver instance. PyGdb supports all basic debugging tasks from setting breakpoints to following forks. Currently our software supports environments running Linux on x86-32, x86-64, or MIPS32 platforms.

The tracing engine is built on top of PyGdb and supports tracing on function as well as on basic-block level. As PyGdb is designed to not include comprehensive disassemblers for its supported platforms, basic blocks are initially identified by stepping single instructions. (We remark that on x86 the problem of statically producing a fully correct disassembly is generally undecidable [222].)

The tracer is aware of *implicit edges* in the CFG induced by conditional instructions and records *virtual basic blocks* accordingly. To understand the need for this consider the x86-32 conditional move instruction `cmovz`: data is only moved from the source to the destination operand in case the *zero-flag* is set (e. g., as the result of a compare operation). Compilers use such instructions to efficiently translate simple *if-then* constructs. Taking this into account can be crucial for successfully detecting and disabling backdoors (see Sections 3.5.1.1 and 3.5.2).

### 3.4.1 Protocol Player

In order to fully automate the analysis process, we developed a system for the specification and playback of protocols. Similar to existing work in the realm of fuzz testing of software [6, 11], we describe protocols in a block-based manner according to their specifications. The blocks describing a protocol are ordered in *levels* and are grouped by *strings* and *privilege levels*. Our description of the FTP protocol according to RFC 959 [162] for example possesses nine levels, the strings CmdArg0, CmdArg1 and CmdArg2 and the privilege levels NoAuth, Anonymous and Auth.

Before traces of a certain server application are recorded, the corresponding *protocol description* is compiled to a set of specific *protocol scripts*. Compiled scripts are solely built of the atoms PUSH_DATA, SEND, RECV and WAIT, and the virtual atoms START_RECORDING and STOP_RECORDING. The last two are automatically inserted by the *protocol compiler* before and after the atoms of interest. When the *protocol processor* encounters one of them while playing the protocol, it activates/deactivates the tracer. We thus ensure that as little noise as possible is recorded. In order to be able to describe interactive elements in protocols (such as for example the PING/PONG messages in the IRC protocol), each atom can dynamically yield new atoms in reaction to the state of the protocol run.

Figure 3.3 schematically shows the description of two commands of the FTP protocol: the command HELP accepts none or one argument. Accordingly, it belongs to the strings CmdArg0 and CmdArg1. As the command is available in any session – unauthenticated as well as authenticated – it belongs to all three privilege levels. In contrast, the command MKD belongs to the privilege levels Anonymous and Authenticated, and solely the string CmdArg1 as it expects one argument and is only available in authenticated sessions.

### 3.4.2 Analysis Modules

We implemented two analysis modules that work on the results delivered by the A-WEASEL analysis algorithms described in Section 3.3.

#### 3.4.2.1 Function Pointer Table Identifier

Many server applications written in C/C++ store command descriptors including pointers to *handler* functions in central data structures such as arrays. We have implemented an analysis module that scans the memory of a server application at runtime for pointers to previously identified handler functions. When the distance in memory between several identified pointers to handler functions is of equal size, we assume that a table of

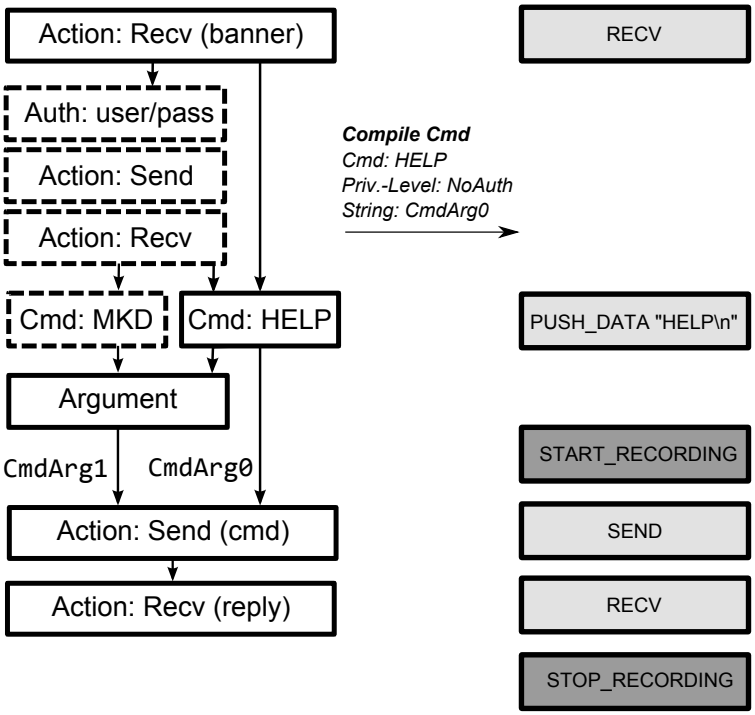


Figure 3.3: Scheme of the description of the FTP protocol (two commands) and the correspondingly compiled script of the command HELP; blocks not available for privilege level NoAuth are dashed. Atoms are gray. Virtual atoms are dark-gray.

command descriptors was found (compare [123, 193]). In the next step, we attempt to heuristically determine the beginning and end of the respective table. Once such a table is identified it is possible to check for pointers to unknown command handlers and thus identify undocumented commands.

#### 3.4.2.2 Differential Function Arguments Identifier

There are several scenarios in which it might be desirable to identify those arguments of a *decider* function that are protocol run-specific (see Section 3.3.4.1). A simple example is a password validation function that expects (among other not session-specific arguments) pointers to both the username and the password entered by a user. We implemented an analysis module that—given the list of decider functions of a server application—tries to heuristically identify such arguments. The module replays the different protocol runs and examines the stack at the entrance to the given decider functions in a differential manner. The module thus only works for calling conventions passing arguments on the stack. The module distinguishes between *data* and *pointers* and tries to identify pointers by dereferencing values in memory and checks if the resulting address resides in the same type of memory for each protocol run. Thereby, the module differentiates between the following types of memory: stack, heap, and binary image. If a value in memory is found

to be pointing to the same type of accessible memory for each protocol run, it is assumed that it is in fact a pointer. Otherwise it is assumed to be plain data. In this case, the data is compared between all protocol runs. In case of any difference, the argument is marked as protocol run-specific. The analysis module follows suspected pointers up to a certain level of depth. This way it is possible to identify protocol run-specific arguments passed inside of nested data structures.

## 3.5 Evaluation

To demonstrate the practicality of our approach, we evaluated Weasel with several open and closed source server applications for different protocols and platforms. The results are summarized in Table 3.1. All applications were tested in a standard configuration. For the sake of simplicity, Weasel was limited to only consider traces of login attempts for the following cases: Valid username/valid password (*valid-pw*) and valid username/invalid password (*invalid-pw*). In its default configuration, Weasel also considers the third case of an invalid username. This can for example be useful for the detection of backdoors triggering on certain usernames. The amount of function call events in a single trace ranged from 3 (BusyBox Telnetd authentication) to 12,335 (ProFTPD command dispatching). In the following, we discuss the test results of three server applications in more detail.

For the other four applications, we remark that the test results were satisfactory to the extent that Weasel correctly identified handlers and deciders of both the authentication and the command dispatching process with little to no noise.

### 3.5.1 Detailed Analysis of SSH Servers

We first examine software backdoors in SSH server implementations. The description of our SSH protocol is limited to the *SSH Authentication Protocol* (SSH-AUTH) according to RFC 4252 [231]. Other aspects of the SSH protocol, such as the transport layer, are not considered for our purposes. RFC 4252 specifies the following authentication methods for SSH-AUTH: *password*, *publickey*, *hostbased*, and *none*. The corresponding protocol specification in Weasel treats these four methods as commands.

#### 3.5.1.1 OpenSSH (x86-64)

For OpenSSH we chose a version that was reported by an antivirus vendor to have been found in the wild containing the following backdoors [76]:

X-1 On startup, the server sends the hostname and port on which it is listening to remote web hosts assumingly controlled by the attackers.

X-2 A master password enables logins under arbitrary accounts without knowledge of the actual corresponding passwords (*password* and *keyboard-interactive* authentication).

X-3 A master public key enables logins under arbitrary accounts with knowledge of the corresponding private key (*publickey* authentication).

| **Server** | **Platform** | **Protocol** | **Decision tree (*Cmd, Auth*)** | **Remarks** |
|---|---|---|---|---|
| BusyBox Telnetd | MIPS32 | Telnet | 1/0, 0/0 | Does not support standard IAC commands. |
| Dropbear SSH | MIPS32 | SSH-AUTH | 3/9, 1/2 | See Section 3.5.1.2 |
| OpenSSH | x86-64 | SSH-AUTH | 2/2, 4/7 (monitor process)<br>2/3, 3/3 (slave process) | Identified command descriptor table; detected backdoors |
| ProFTPD | MIPS32, x86-32, x86-64 | FTP | 6/60, 5/37 (x86-64) | See Section 3.5.2 |
| Pure-FTPd | x86-64 | FTP | 2/29, 1/11 | Results similar to ProFTPD |
| NcFTPD | x86-32, x86-64 | FTP | 4/40, 2/11 (x86-64) | Results similar to ProFTPD |
| Dancer-IRCD | x86-64 | IRC | 2/28, 2/12 | Identified command descriptor table. |

Table 3.1: Overview of evaluation results; the *decision tree* column describes the calculated command dispatching and authentication decision trees in the form *<number of decider functions>/<number of handler functions>*.

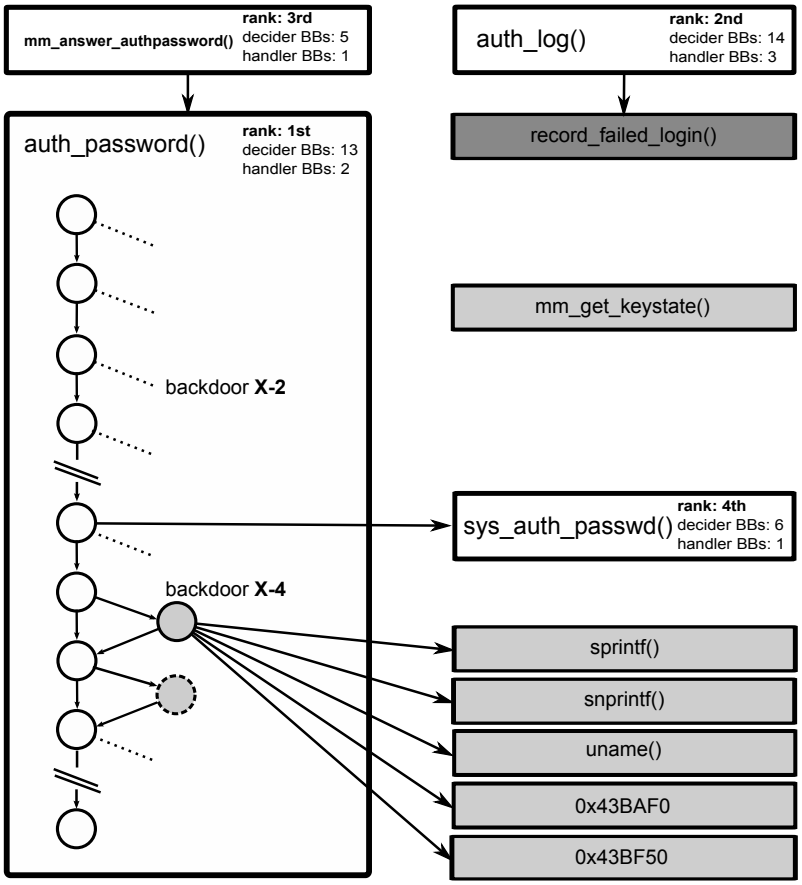


Figure 3.4: Decision tree for the password authentication in the monitor process of the malicious version of OpenSSH; functions are rectangles, basic blocks are circles. Deciders are white, handlers for *valid-pw* are gray, handlers for *invalid-pw* are dark-gray. Cold edges are dotted.

X-4 Credentials used in any successful login attempts are sent to the remote web hosts (*password* and *keyboard-interactive* authentication).

As no source code is publicly available for this malicious version of OpenSSH, we had to limit our evaluation to the x86-64 platform. Due to the privilege separating architecture of OpenSSH [164], Weasel automatically generates decision trees for two processes: a privileged process called *monitor* and an unprivileged process called *slave*.

**Authentication** Backdoor X-4 can be easily spotted from the decision tree of the monitor process for the SSH *password* authentication as depicted in Figure 3.4. The decision tree only contains the decider functions located at virtual addresses `40B440h` (`auth_password()`), `420E20h` (`mm_answer_authpassword()`), `412EB0h` (`auth_log()`), and `40-`

`B390h` (`sys_auth_passwd()`) in the binary file[4]. The scoring algorithms of WEASEL rank the decider `auth_password()` as most important. It leads to five exclusive handlers for *valid-pw* that are all called from the same handler basic block and implement backdoor X-4.

Of these exclusive handlers, the one at addresses `43BF50h` can automatically be identified as highly suspicious, as it (among others) statically calls the functions `socket()`, `connect()` and `write()`. Manual analysis reveals that this handler function attempts to send data to remote web hosts. Correspondingly, the handler function at address `43BAF0h` implements URL encoding of strings.

The basic-block level decision tree of `auth_password()` contains 13 deciders and two exclusive handlers for the *valid-pw* protocol run (see Figure 3.4). While one of the handlers contains backdoor X-4 as described above, the other handler is a legitimate virtual basic block induced by the conditional assembly instruction `setnz dl`, which sets the return value of the function according to the validity of the password. Of the 13 deciders in the basic-block level decision tree, eleven are cold.

Most importantly, these cold edges are related to the optional PAM authentication[5], password expiration handling, and backdoor X-2 (*master password*). The attacker implemented this backdoor by adding a short piece of code at the beginning of `auth_password()`: each password to check is compared to a predefined one. In case of a match, the function returns, falsely indicating a successful authentication to its caller. The backdoor is automatically rendered inoperative by *cutting* (see Section 3.3.4.2) the cold edges in `auth_password()`.

As WEASEL's protocol description of SSH-AUTH does not cover the *publickey* authentication method, we cannot find the backdoor X-3. Note that this is only a limitation of the current protocol description as the implementations of backdoor X-3 and backdoor X-2 are very similar on assembler level. WEASEL cannot be used to identify backdoor X-1 (*notification of remote web hosts on startup*), because it is designed to only examine the authentication and command dispatching processes of server applications.

The decision tree of the slave process is not depicted. It consists of three deciders, two handlers for *invalid-pw* and a single handler for *valid-pw*. As static analysis hints at nothing suspicious in these functions, the slave process is not further discussed here.

**Command Dispatching** As for the authentication, command dispatching in OpenSSH stretches over a monitor and a slave process. For the slave process, WEASEL identifies only the function at `414960h` (`input_userauth_request()`) as decider. Also, the decision tree of the slave process only contains exclusive handler functions for the protocol runs of the commands *password*, *publickey* and *none*: `41BF00h` (`userauth_passwd()`), `41CA50h`(`userauth_pubkey()`) and `41BE20h` (`userauth_none()`), respectively. These results suggests that the *hostbased* authentication method is disabled – a circumstance that can be verified by manual analysis.

[4]The actual malicious binary file does not contain debugging symbols and thus names of function cannot be obtained directly. For reasons of clearness, the names of functions of interest were manually resolved by comparing assembly code and OpenSSH source code.

[5]Code path related to PAM were not taken during testing, as PAM was not enabled in our employed default configuration.

Based on this results, WEASEL's analysis module that heuristically scans for function pointer tables as described in Section 3.4.2 automatically and unambiguously identifies the correct address and size of the command descriptor table in the `.data` section of the binary file, spanning from virtual address `674678h` to `6746f8h`. The table, which is defined in the OpenSSH source code under the identifier `authmethods`, contains simple structures of three members (*name*, *handler function*, *enabled flag*) describing all available authentication methods of the server. Interestingly, the analysis module identifies two handler functions not contained in any of the collected traces: `41BD40h`(`userauth_kbdint()`) and `41B9F0h` (`userauth_hostbased()`). While the latter belongs to the disabled authentication method *hostbased*, the former belongs to the well-known authentication method *keyboard-interactive* which is not described in RFC 4252 (and suffers as well from backdoors X-2 and X-4). This demonstrates the ability of WEASEL to identify handlers for unknown commands. The decision tree of the monitor process contains only the monitor-side handler function for the *password* authentication method. We do not discuss it further due to space restrictions.

#### 3.5.1.2 Dropbear SSH (MIPS32)

We examined a binary-only version of Dropbear SSH server shipped as part of the firmware of the Siemens VoIP desk telephone *OpenStage 40*. Function and basic-block traces were recorded remotely on the embedded hardware. We exploited a memory corruption vulnerability (see Section 2.2) in the device's firmware to gain root privilege and upload a cross-compiled version of gdbserver [6]. We were able to automatically and unambiguously identify important deciders and handlers in both the authentication and the command dispatching process of the SSH server.

**Authentication** The rather compact decision tree computed from the *valid-pw* and *invalid-pw* protocol runs is depicted in Figure 3.5.

The only identified decider `svr_auth_password()` evaluates the correctness of a password by a simple string comparison and, depending on the outcome, subsequently calls one of the two identified handlers. No suspicious external functions are reachable from either handler.

**Command Dispatching** The decision tree computed from the protocol runs corresponding to the four authentication methods of the SSH-AUTH protocol contains three deciders, with `recv_msg_userauth_request()` ranking first. From this decider, the only exclusive handlers `svr_auth_password()` and `svr_auth_pubkey()` are called, which belong to the *password* and the *publickey* authentication method, respectively. The application does not contain exclusive handlers for the other authentication methods. As Dropbear SSH in general does not dispatch commands via function pointer tables, WEASEL does in this case correctly not recognize any function pointer tables of interest. The two identified exclusive handler functions were not found to lead to any suspicious calls.

[6] The vulnerability was reported by us and acknowledged and fixed by the vendor in March 2013.

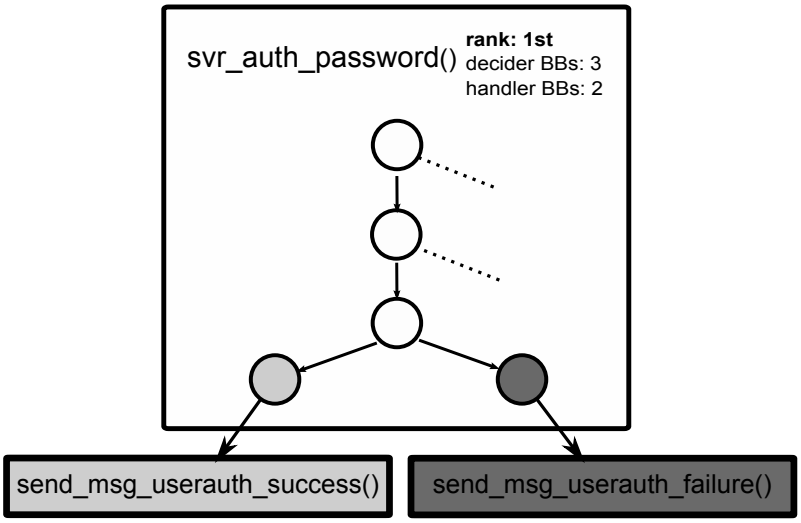


Figure 3.5: Decision tree of the authentication process of Dropbear SSH

### 3.5.2 ProFTPD (x86, x86-64, MIPS32)

Graduate students were asked to implement an arbitrary set of backdoors for ProFTPD in addition to the real-world backdoor of our running example. We have chosen ProFTPD since it is a complex program (e. g., FTP is a non-trivial protocol, it contains different software modules, and there were real-world attacks against this program) but still has a manageable code base. Altogether, eleven different backdoors were developed in a not supervised manner.

Seven out of these eleven backdoors interfere with the authentication or command dispatching process of ProFTPD and can thus theoretically be identified using Weasel. The other four backdoors implement malicious functionality like a covert out-of-band interactive shell that cannot be found by applying the described detection approach. We thus evaluate Weasel on the following set of backdoors containing the seven artificial backdoors and our running example:

Y-1 *Acidbitchez*: our real-world running example.

Y-2 *Happy hour*: at a certain time of day all passwords are accepted by the server.

Y-3 *Blessed one*: for a certain client IP address all passwords are accepted.

Y-4 *File access*: hidden commands for unauthorized file access bypassing the authentication process.

Y-5 *Credentials stealing 1*: validated credentials are stored and made available via a hidden command.

Y-6 *Credentials stealing 2*: validated credentials are sent via DNS requests to a remote server.

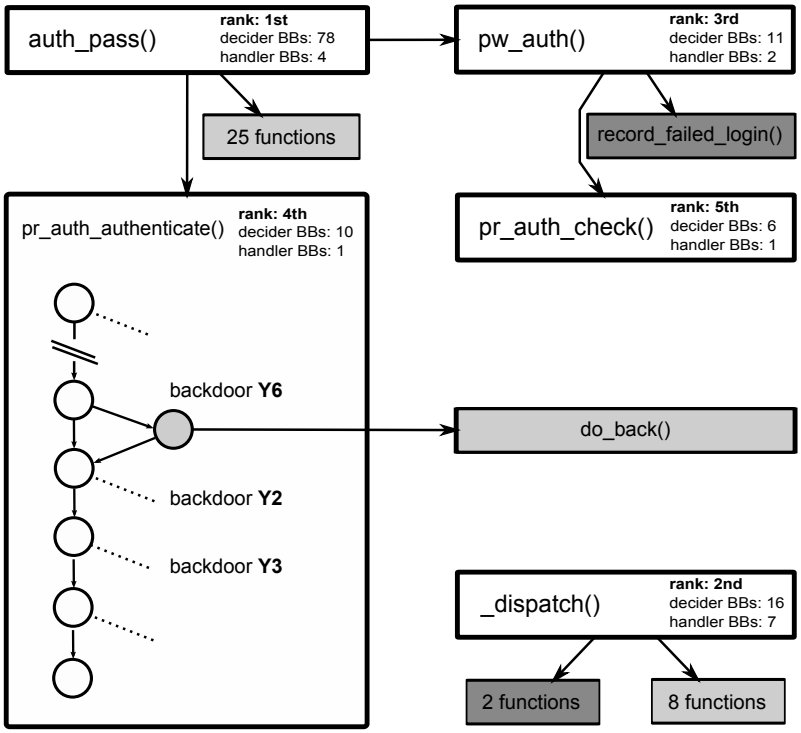


Figure 3.6: Decision tree of the authentication process of ProFTPD

Y-7 *Self-modifying authentication*: in case of a certain process ID, the central authentication function is rewritten on startup of the application to accept arbitrary passwords.

Y-8 *Authentication module*: a malicious auth. module.

We analyzed a version of ProFTPD containing this backdoors on Weasel's supported platforms x86-32, x86-64 and MIPS32. Since the high-level results were identical for all platforms, we discuss them together (concrete numbers apply at least to the x86-64 version). We demonstrate that it is possible to reliably detect or to disable backdoors Y-1–Y-6, substantiating our approach to reduce the attack surface.

We cannot cope with backdoors Y-7 and Y-8: Backdoor Y-7 can only be detected when during testing by chance a case is encountered where the authentication function is actually being overwritten (i. e., the backdoor triggers on a certain process ID). Backdoor Y-8 cannot be identified as Weasel currently does not evaluate dynamically loaded modules.

**Authentication** The decision tree computed from the *valid-pw* and *invalid-pw* protocol runs is depicted in Figure 3.6. On the function level, four deciders are identified, with `auth_pass()` with 78 deciders on basic-block level, 25 handler functions for *valid-pw* and the 1st scoring rank being clearly dominant. Though, three of the eleven backdoors are located in the decider function `pr_auth_authenticate()`, which only leads to a single handler function (`do_back()`) for *valid-pw*. This handler function belongs to backdoor Y-6 and stores credentials once they are successfully validated. The function can automatically

be identified as highly suspicious, as it (among other activities) statically calls the standard functions `mmap()`, `shm_open()`, `socket()`, and `sendto()`. Two of the nine cold decider basic blocks in `pr_auth_authenticate()` lead to the implementations of backdoors Y-2 and Y-3, respectively. Thus cutting cold edges in the identified deciders would effectively render these backdoors non-functional.

In the x86-32 version of our modified ProFTPD server the code implementing the "happy hour" backdoor (Y-2) in `pr_auth_authenticate()` uses the conditional instruction `CMOVZ` to manipulate the outcome of the function. This underlines the need to consider *implicit edges* when examining a function's CFG since we would otherwise overlook this case.

**Command Dispatching** WEASEL's protocol description of FTP was modelled according to RFC 959 [162] and contains 34 commands (e. g., `HELP` and `MKD`). The function level decision tree consists of six deciders (of which `pr_cmd_dispatch_phase()` ranks 1st) and 60 handlers. Out of those handlers, 43 are exclusive to a single protocol run. By manual analysis it can be verified that these exclusive handlers indeed implement each of the 34 commands. For the majority of commands there exists exactly one exclusive handler. Subsequently, it is easily possible to automatically identify the backdoor of our running example (Y-1): among the external functions reachable from `HELP` command's only exclusive handler `core_help()` in the static CFG are the aforementioned in this context highly suspicious ones `setegid()`, `seteuid()` and `system()`. Accordingly, the corresponding `HELP` command can be identified as suspicious and further defensive measures can be applied.

WEASEL automatically and correctly identifies the addresses and sizes of exactly five function pointer tables in the address space of the respective ProFTPD process: `core_cmdtab`, `xfer_cmdtab`, `auth_cmdtab`, `ls_cmdtab`, and `delay_cmdtab`. The first one is the largest and contains 35 entries describing the core set of the commands supported by ProFTPD. Eight entries in `core_cmdtab` contain function pointers that are not contained in any of the recorded traces. By examining the respective entries in the table in the binary program, they can already be identified to be corresponding to the following commands: `EPRT`, `EPSV`, `MDTM`, `SIZE`, `DOWNLOAD`, `UPLOAD`, `GSM` and `RSLV`. While the first four are known benign FTP commands that are simply not defined in RFC 959, the last four belong to the backdoors Y-4, Y-5 and Y-6. The other four identified function pointer tables also partly contain pointers to functions that were not encountered during testing. These functions correspond either to known commands (e. g., `xfer_log_stor()`) or the benign FTP command `PROT`, which is not defined in RFC 959 as well.

## 3.6 Related Work

Backdoors in computer systems are a well-known security problem. Over the years it has been considered in different contexts and from different perspectives.

One line of work is the detection or implementation of backdoors in integrated circuits [5, 27, 99, 106, 116, 166, 219, 220]. Furthermore, several backdoors in different components of a computer such as network cards [198] or directly in the CPU [74] were proposed. Our approach is orthogonal to such work since we focus on the detection of backdoors in binary software, an area that has received little attention so far.

Wysopal et al. [227] presented a pattern-based, static analysis approach to identify backdoors in software. The main limitation is that patterns need to be specified in advance, which implies that potential backdoors need to be known before the analysis can be carried out. Costin et al. [56] recently analyzed a large set of embedded devices' firmwares for backdoors by means of static pattern matching akin to the techniques proposed by Wysopal et al. [227]. Their system dynamically expands the set of suspicious patterns and they report on having found previously unknown backdoors (and vulnerabilities in general) in a large number of different firmware images this way. However, the backdoors identified by their approach all correspond to hardcoded login credentials.

Two recent works [157, 158] by Pewny et al.[7] concern with the static identification of variants of known bugs and backdoors in binary software across different compilers, operating systems, and (partly) also across different processor architectures. On the baseline, these works extract the semantics of a certain backdoor from a known vulnerable software using techniques from the realm of symbolic execution and search for similar semantics in other software. Among others, the authors demonstrate how the Heartbleed bug can be found in different embedded firmwares using their techniques [157].

The Firmalice system [189] aims at statically identifying authentication backdoors in the binary code of embedded devices' firmwares. To this end, Firmalice applies symbolic execution and program slicing techniques to identify program paths that reach "privileged program points" without being authenticated. This way, Firmalice detected backdoors that bypass authentication in the firmwares of two different devices.

Geneiatakis et al. [86] proposed a similar (although not backdoor-related) technique to ours to identify "authentication points" in server applications using Intel's Pin [128]. However, our approach can be applied to a broader range of platforms and environments and is fully automated.

A basic insight for backdoor detection is that some kind of trigger needs to be present such that an attacker can activate the backdoor. As a result, work on automated identification or silencing of triggers is also related to our work. Brumley et al. [45] demonstrated how trigger-based behavior in malware can be identified and Dai et al. introduced an approach to eliminate backdoors from *response-computable authentication* routines, i. e., the typical challenge-response based authentication [61]. While such approaches reduce the attack surface, an attacker can still implement a backdoor and bypass the approach (e. g., by adding additional command handlers or completely bypassing the authentication process). Our approach complements such approaches and helps to reduce the attack surface even further.

Somewhat related to our approach is the idea of *privilege separation*, i. e., the process of splitting an application into different isolated trust domains [33, 46, 113, 141, 164, 178, 230, 232]. Note that such approaches do not completely mitigate the risk of backdoors since an attacker can often still install certain types of backdoors within sufficiently privileged components. The backdoors for SSH servers analyzed in Section 3.5 demonstrate that this is indeed a problem in practice.

Two advanced approaches for the stealthy implementation of backdoors in software have recently been proposed: Andriesse and Bos [15] presented a technique for the stealthy

[7] The author of this dissertation has co-authored the earlier one of these two works.

implementation of backdoors in software. On the baseline, they propose to implement the actual backdoor functionality in *unaligned* x86 assembly instructions (see also Section 2.2.4.2). When deployed, the attacker makes an application's control flow reach these instructions by exploiting a purposely planted control-flow hijacking vulnerability. Bosmann and Bos [37] described *sigreturn oriented programming* (SROP, already introduced in Section 2.2.4.2 in Chapter 2) a code-reuse attack approach that borrows basic principles from ROP. In SROP, malicious computations are implemented through repeated invocations of the UNIX system call `sigreturn`. Bosmann and Bos discuss SROP as a potential technique for the implementation of stealthy backdoors. While our approach is unlikely to be applicable to backdoors of this kind, it can still be used to selectively disable or restrict corresponding suspicious functionality of a binary server application. We emphasize that the aim of our work is not to prevent *all* kinds of backdoors (which is impossible) but rather the most notorious ones.

## 3.7 Conclusion

In this chapter, we presented an approach towards the automatic detection and disabling of certain types of backdoors in server applications by carefully examining runtime traces for different protocol runs. Our implementation of the approach in the form of a tool called Weasel automatically captures these traces by repeatedly invoking a server application under test according to a formal, block-based specification of the respective protocol. As Weasel only relies on gdbserver for the recording of traces, it is widely applicable to a variety of platforms and we discussed several empirical analysis results that demonstrate how Weasel can be used to precisely detect relevant code parts and data structures within a given binary application.

Our approach makes heavy use of different heuristics. We showed in Section 2.4 in Chapter 2 that heuristics-based ROP detection systems can be bypassed by aware attackers with little effort. The same most probably also applies to our approach and the Weasel tool in the context of backdoors. That is, in case an attacker is aware of our heuristics it becomes simple to evade them. However, we emphasize that the goal of our work here is not to provide a strong generic defense against backdoors but rather to demonstrate automated techniques that can mitigate some of the most notorious ones. We hope that our results encourage further research on how to mitigate the real but often neglected threat of backdoors in server applications.

Chapter 4

# Trustworthy Data Analytics in the Cloud using SGX

Without doubt has cloud computing been one of the big trends in IT in recent years. Despite this, a range of common concerns about security in cloud computing exists (see e. g., [112, 202]), which still largely demand to be addressed in a verifiable and dependable, yet practical manner. In this chapter, the CLOUD setting, the third and final one of the adversarial settings defined in Chapter 1, is tackled: the *VC3* (short for *Verifiable Confidential Cloud Computing*) system is presented, which guarantees the confidentiality and integrity of code and data for C++ MapReduce applications executed in the untrusted cloud. In particular, *VC3* also allows users to verify the overall integrity of distributed MapReduce computations, i. e., that a computation ran to completion and its results were not tampered with. *VC3*, as will be shown, provides these strong confidentiality and verifiability guarantees even if the cloud operating system or hypervisor are compromised or operated by a malicious administrator. *VC3* relies on the Intel's SGX [12, 100, 133] technology. As processor chips equipped with this technology are not available at the time of this writing, a practical implementation of *VC3* based on an emulator is described. Our experimental results show that *VC3* will, given that actual performance of SGX will be as expected, enable general-purpose secure cloud computation with almost negligible performance overhead in many cases.

Like the previous chapters, this one begins with an introduction to the considered adversarial setting (Section 4.1) and a motivation for the conducted research (Section 4.2). Next, in Section 4.3, important background on Intel SGX and MapReduce is given and our cryptographic assumptions are defined. Section 4.4 describes the high-level architecture of *VC3*. The cryptographic protocols *VC3* relies on for its *key exchange* and the *job execution* are specified in sections 4.5 and 4.6. Subsequently, *VC3*'s resilience against certain practical (side channel) attacks is discussed (Section 4.7) and formal proof for the security of our protocols is provided (Section 4.8). Our implementation of *VC3* is described in Section 4.9 and the results of its experimental evaluation are discussed in Section 4.10. The chapter closes with a brief description of possible future extensions to *VC3* (Section 4.11), an overview of related work (Section 4.12), and a conclusion (Section 4.13).

## 4.1 Adversarial Setting

In the CLOUD setting (see Chapter 1), from a security perspective, cloud providers are today usually only able to offer forms of encrypted storage. This is sufficient in cases where a cloud provider's service is limited to the storing of data. However, many use cases of cloud computing also involve the processing of data. For example, MapReduce [16, 69] is a popular framework for the distributed processing (i. e., on multiple computing nodes in parallel) of data in the cloud. To process data efficiently, cloud providers typically need to access their users' data *and* code in plain form at one point in time. Once a user's data and code float decrypted in a cloud provider's data centers, the confidentiality and integrity of both are at risk. Of concern are not only external intruders but also the cloud provider's organization. For example, a corrupt administrator may leak data or prevent users' code from running to completion in order to misuse the (paid for) resources.

### 4.1.1 Attacker Model

Precisely, for the remainder of this chapter, we assume a powerful attacker that may control the entire software stack (including hypervisor and operating system) of systems in a cloud provider's infrastructure and may arbitrarily record and replay network packets. As stated, *VC3* inherently relies on SGX-enabled Intel processors (further introduced in Section 4.3) as TCB. As such, regarding hardware attacks *VC3* is naturally bound to the defensive strength of SGX. Hence, we consider attackers that try to read or modify protected data after it (temporarily) left the processor by attaching electronic probes, through *direct memory access* (DMA), or similar techniques. However, in the model considered in this chapter, the attacker is *not* able to physically open and extract secrets from certified SGX processors in the cloud provider's data centers. Such an attack would probably require access to a processor in a sophisticated laboratory environment for an extended period of time and we assume that the cloud provider has effective measures in place to prevent the removal of intact processor chips from its data centers.

We also neglect the possibility of fault-injection attacks against SGX processors, e. g., through power spikes, and other side channels that undermine the general security properties of the SGX technology. Denial-of-service and network traffic-analysis attacks are outside our scope. Furthermore, for now, we also do not consider the two previous adversarial settings CLASSIC (e. g., control-flow hijacking attacks) and BACKDOOR for the trusted code parts of *VC3*.

In essence, we assume parts of the staff of the cloud provider to be corrupt or its infrastructure being compromised by an expert and persistent remote attacker.

## 4.2 Research Motivation and Contributions

In general, cloud users hope for the following security guarantees:

I Confidentiality and integrity for *data* and *code*; i. e., the guarantee that the code and data are not changed by attackers and that they remain secret.

II Verifiability of the execution of the *code* over the *data*; i. e., the guarantee that the distributed computation ran to completion and was not tampered with.

In theory, multiparty computation techniques could address some of them. For instance, data confidentiality can be achieved using *fully homomorphic encryption* (FHE), which enables cloud processing to be carried out on encrypted data [87]. However, FHE is not efficient for most computations [88]. In line with this, Van Dijk and Juels argued in 2010 "[...] that cryptography alone can't enforce the privacy demanded by common cloud computing services, even with such powerful tools as FHE" [215].

A computation can also be shared between independent parties while guaranteeing confidentiality for individual inputs (using e. g., garbled circuits [103]) and providing protection against corrupted parties (see e. g., SPDZ [63]). In some cases, one of the parties may have access to the data in the clear, while the others only have to verify the result, using zero-knowledge proofs (see e. g., Pinocchio [154], Pantry [40], and ZQL [81]). Other systems use specific types of computation and do not use strong encryption for all code and data (see e. g., CryptDB [159] and Cipherbase [19]). Still, our goals cannot currently be achieved for distributed general-purpose computations using these techniques without a serious impact on performance.

The *VC3* MapReduce framework described in this chapter provides the security guarantees I and II formulated above. These guarantees even hold if the cloud operating system or hypervisor are compromised, or operated by a malicious administrator. The considered attacker model (already given in Section 4.1.1) accounts for powerful adversaries who potentially control the whole cloud provider's infrastructure, except for the certified SGX-enabled processors that are involved in the computation.

### 4.2.1 Approach Overview

In *VC3*, users upload encrypted code and data to the cloud. On each involved computing node, the cloud operating system loads the encrypted code into a *secure region* within the address space of a process and leverages the hardware security mechanisms of Intel SGX to make this region inaccessible to the operating system and the hypervisor. Subsequently, in *VC3*, the code inside the secure region decrypts itself and runs the distributed computation that processes the data. *VC3* uses a form of tamper-evident logging [174] to ensure the integrity of the distributed computation as a whole. The computing nodes produce summaries of their work and aggregate the summaries they receive from their peers. By verifying the summaries included in the final results of the computation, the user can unambiguously check that the results are correct for the given code and data. While this mechanism protects the user from malicious or faulty behavior by the cloud provider, the cloud provider can still freely schedule and balance the computation between the nodes in *VC3*.

While the focus of this chapter mainly lies on MapReduce applications with a single user, *VC3* also enables secure multi-user computations in the cloud. A typical multi-user scenario would for example be the case of competitors wishing to compute statistics over their combined annual revenue and disclose only aggregate results, using the cloud as a neutral ground.

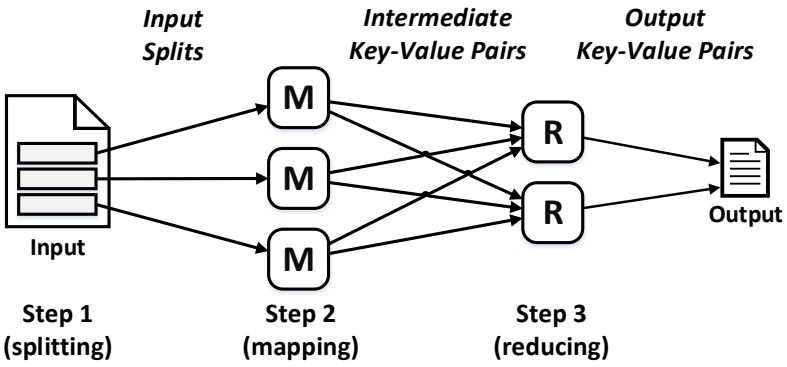


Figure 4.1: The steps of a MapReduce job as discussed in this work with mappers (**M**) and reducers (**R**)

We implemented *VC3* for the popular MapReduce platform *Hadoop* on the Windows operating system. Users simply write the usual map and reduce functions in C++, and *VC3* compiles them into an executable that implements the *Hadoop* streaming interface [18]. Experimentally, our implementation is based on the specification [109] of the new hardware security mechanisms of Intel SGX, but it could in principle target other hardware-based secure computing technologies [20, 38, 131, 149]. Benchmarks show that *VC3* provides general-purpose secure cloud computation with negligible performance overhead across a diverse set of MapReduce applications.

## 4.3 Background

### 4.3.1 MapReduce

MapReduce [69] is a popular programming model for processing large data sets: users write *map* and *reduce* functions, and the execution of both functions is automatically parallelized and distributed.

The abstract data flow of a parallel MapReduce job is depicted in Figure 4.1. Each job is a series of three steps: *splitting*, *mapping*, and *reducing*. In the splitting step, the framework breaks raw input data into so called *input splits*. Input splits are then distributed between mappers. Each mapper node parses its splits into *input key-value pairs*, and calls the map function on each of them to produce *intermediate key-value pairs*. The framework groups these pairs by key and distributes them between reducers (also referred to as *partitioning* and *shuffling*). Each reducer node calls the reduce function on sets of all the values with the same key to produce *output key-value pairs*.

Probably the most popular framework for the execution and deployment of MapReduce jobs is *Hadoop* [16]. Hence, we chose it as our default execution environment.

### 4.3.2 Intel SGX

SGX is an upcoming extension to the x86-64 instruction set architecture (ISA). All in all, SGX introduces 17 new instructions to x86-64. With the SGX technology, Intel aims to "[...] enable [software] developers to develop and deploy secure applications on open platforms" [133]. There are some advantages of SGX in the light of our approach which to our knowledge no other competing technology offers: SGX is likely going to be widely available within the next years as part of professional and consumer commodity x86-64 processors—other than, e. g., OASIS-enabled processors [149]. Further, SGX is designed to work well with existing virtualization technologies [109] and has little impact on the overall runtime behavior/responsiveness of a system—other than, e. g., Flicker [131] which is based on the older *AMD Secure Virtual Machine* (SVM) x86-64 ISA extension [4]. The last aspect is especially important in the context of cloud computing.

**Creation and Measurement of Enclaves** The core feature of SGX is the creation of *enclaves* within conventional user mode processes. An enclave is a continuous memory region that contains code and data of a sensitive application. Enclaves can be created by kernel mode code in any user mode process using new x86-64 instructions such as `ECREATE` or `EINIT`. The creation of an enclave is *measured* by the processor into a secure log [12], which is called an enclave's *measurement*. The measurement reflects, among others, an enclave's contents and memory layout. It has the form of a cryptographic hash (digest). The creation of an enclave always ends with kernel mode code (i. e., the operating system or a driver) executing the `EINIT` instruction for it. After that, the enclave's measurement is fixed and user mode code can enter the enclave at fixed entry points using the `EENTER` instruction. A processor core is bound to an enclave until *(i)* the in-enclave code invokes the `EEXIT` instruction or *(ii)* an exception or an (external) interrupt is triggered. The latter case is also referred to as *asynchronous enclave exit* (AEX) [109]. In the event of an AEX, the respective core's state (defined by the core's registers' values) is securely stored within the enclave before being wiped. In-enclave code has not only access to memory pages included in the enclave, but also to the entire address space of its host process. This is a powerful feature of SGX that allows for high-performance interaction between code inside and outside of an enclave. However, a running enclave is read/write protected from the entire rest of the system. This accounts for the operating system and other software as well as for any other hardware component than the processor core that runs the enclave. Also, system calls are conceptually not available within enclaves [109]. Hence, enclave code needs to be largely self-sufficient; e. g., it cannot rely on the operating system to initialize and manage stacks and heaps. Consequently, most existing software, including essential libraries such as `kernel32.dll` or common libc implementations, cannot be used within enclaves without complex modifications.

By design, the operating system manages an enclave's memory pages and schedules its execution time. The operating system may thus refuse to create an enclave, block the execution of an enclave, or discard its pages from memory (*denial of service*) but it can never read from or write to a running enclave. Any misbehavior from an untrusted component during the setup of an enclave inevitably leads to a corrupted measurement.

**Key Derivation and Remote Attestation** SGX provides *sealed storage* and *attestation* for enclaves [12]. These features have the same basic purpose as sealed storage and attestation in other trusted computing hardware, e. g., TPMs. In-enclave code can use the new `EGETKEY` instruction to derive various types of symmetric keys in a deterministic, yet to the outside opaque way. The only key types of immediate interest to our approach are the *sealing key* and the *report key*. Sealing keys can be used to *seal* data, e. g., to a local untrusted hard disk drive. Report keys are a concept specific to SGX. They are used for *local attestation* between enclaves and thus enable the establishment of secure communication channels between local enclaves. To enable *remote attestation* on top of the local attestation mechanism, each SGX processor is provisioned with a unique asymmetric private key for the EPID group signature scheme [42–44] that can be accessed only by a special *quoting enclave* (QE) [12]. We refer to this special QE as SGX QE. The SGX QE signs measurements of local enclaves together with digests of data produced by them, creating so called *quotes*. A quote proves to a remote entity that certain data came from a specific enclave running on a genuine SGX processor.

### 4.3.3 Cryptographic Assumptions

We now introduce standard notations and security assumptions for the cryptography we use. We write $m \mid n$ for the tagged concatenation of two messages $m$ and $n$. (That is, $m_0 \mid n_0 = m_1 \mid n_1$ implies both $m_0 = m_1$ and $n_0 = n_1$.)

**Cryptographic Hash, PRF, and Enclave Digest** We rely on a keyed pseudo-random function, written $\mathsf{PRF}_k(text)$ and a collision-resistant cryptographic hash function, written $\mathsf{H}(text)$. Our implementation uses HMAC and SHA-256.

We write $\mathsf{EDigest}(C)$ for the SGX measurement of an enclave's initial content $C$. We refer to $C$ as the *code identity* of an enclave. Intuitively, $\mathsf{EDigest}$ provides collision resistance; the SGX specification [109] details its construction.

**Public-key Cryptography** We use both public-key encryption and remote attestation for key establishment. A public-key pair $pk, sk$ is generated using an algorithm $\mathsf{PKGen}()$. We write $\mathsf{PKEnc}_{pk}\{text\}$ for the encryption of $text$ under $pk$. In every session, the user is identified and authenticated by a public-key $pk_u$. We assume the public-key encryption scheme to be at least *IND-CPA* [28]: without the decryption key, and given the ciphertexts for any chosen plaintexts, it is computationally hard to extract any information from those ciphertexts. Our implementation uses an *IND-CCA2* [28] RSA encryption scheme.

We write $\mathsf{ESig}_P[C]\{text\}$ for a quote from a QE with identity $P$ that jointly signs $\mathsf{H}(text)$ and the $\mathsf{EDigest}(C)$ on behalf of an enclave with code identity $C$. We assume that this quoting scheme is *unforgeable under chosen message attacks* (UF-CMA). This assumption follows from collision-resistance for $\mathsf{H}$ and $\mathsf{EDigest}$ and UF-CMA for the EPID group signature scheme [42]. Furthermore, we assume that Intel's quoting protocol implemented by QEs is secure [12]: only an enclave with code identity $C$ may request a quote of the form $\mathsf{ESig}_P[C]\{text\}$.

```
#include "SGXLibc.h"
#include "MapRed.h"

/* ... */

void Mapper::map(
  char* k, char* v, Context<String*, String*>& c) {

  // Parse input for words.
  char *cur = v;
  while(*cur != '\0') {
    char *word = cur;
    cur = AdvanceToNextWord(word);
    if (!WordIsValid(word))
      continue;

    // Write <word:'1'> out as intermediate KV.
    c.write(new String(word), new String(1));
  }
}

/* implementation of reduce function */
void Reducer::reduce(
  char* w, StringList* l, Context<String*, String*>& c) {

  unsigned long count = 0;

  // Accumulate occurrences of word.
  for(char *v = l->begin(); v != l->end(); v = l->next())
    count++;

  // Write <word:count> as output KV.
  c.write(new String(w), new String(count));
}
```

Listing 4.1: WordCount for *VC3* (C++)

**Authenticated Encryption** For bulk encryption, we rely on a scheme that provides *authenticated encryption with associated data* (AEAD). We write $\mathsf{Enc}_k(\mathit{text}, \mathit{ad})$ for the encryption of *text* with associated data *ad*, and $\mathsf{Dec}_k(\mathit{cipher}, \mathit{ad})$ for the decryption of *cipher* with associated data *ad*. The associated data is authenticated, but not included in the ciphertext. When this data is communicated with the ciphertext, we use an abbreviation, writing $\mathsf{Enc}_k[\mathit{ad}]\{\mathit{text}\}$ for $\mathit{ad} \mid \mathsf{Enc}_k(\mathit{text}, \mathit{ad})$. (Conversely, any IV or authentication tag used to implement AEAD is implicitly included in the ciphertext.) We assume that our scheme is both *IND-CPA* [29] (explained above) and *INT-CTXT* [29]: without the secret key, and given the ciphertexts for any chosen plaintexts and associated data, it is hard to forge any other pair of ciphertext and associated data accepted by $\mathsf{Dec}_k$. Our implementation uses AES-GCM [132], a high-performance AEAD scheme.

## 4.4 Architecture

In *VC3*, users implement MapReduce jobs in the usual and familiar way. The choice of programming languages is though limited to C/C++ for now. The integration with *VC3* is straightforward: *VC3* provides an abstract C++ class interface and, in the simplest case, the user only implements the (virtual) functions `Mapper::map()` and `Reducer::reduce()`. As an example, the source code of the *VC3* implementation of the classic introductory MapReduce job *WordCount*[1] is given in Listing 4.1.

[1] See `http://wiki.apache.org/hadoop/WordCount` (accessed 06/10/2015)

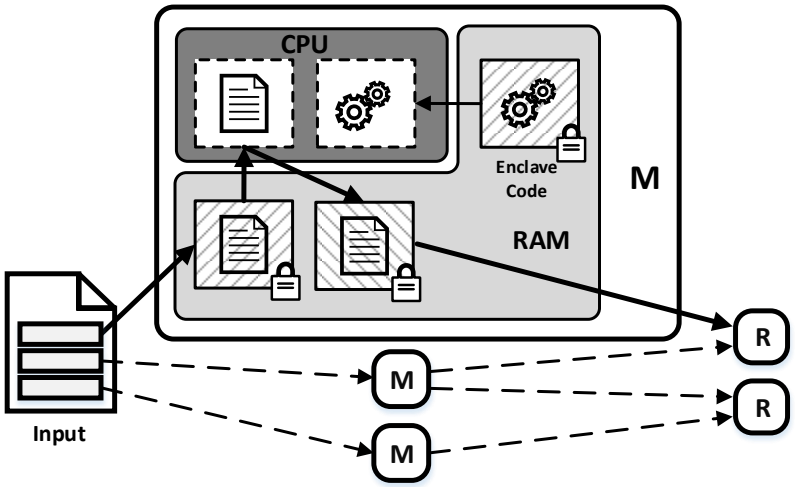


Figure 4.2: High-level concept of a *VC3* enhanced MapReduce job: code and data are always kept encrypted when outside the processor chip.

In *VC3*, the user-provided code constitutes the *private enclave code* $E^-$. $E^-$ and the user's data are distributed exclusively in encrypted form and are ever only decrypted inside enclaves as depicted in Figure 4.2.

$E^-$ is linked against *VC3*'s generic *public enclave code* $E^+$, which implements *VC3*'s central *key exchange* and *job execution* protocols (detailed in sections 4.5 and 4.6). Together with $E^-$ and $E^+$, *VC3*'s public and untrusted *framework code* $F$ is deployed to all worker nodes that participate in a MapReduce job. $F$ comprises a user mode component and a kernel driver. On each worker node, $F$ initializes an enclave within the process address space of its user mode component and loads $E^-$ and $E^+$ into this enclave. Inside the enclave, $E^+$ initializes and manages the enclave's stack and heap. Figure 4.3 depicts (left) the memory layout of a user mode process containing the described components and (right) outlines their dependencies and distinguishes between *trusted* and *untrusted* components. Abstractly, *VC3*'s software TCB only comprises $E^-$ and $E^+$.

Once it has securely received the corresponding cryptographic key, $E^+$ decrypts $E^-$ inside the enclave. This concept is similar to that of "software packers", which have long been used as means to enforce digital rights management (DRM) policies and to complicate reverse engineering attempts. Such software packers can often be effectively attacked by reading the concealed code parts from memory once they are decrypted, e. g., using a standard debugger. Note how in our approach such an attack is not feasible as enclave memory pages are generally not readable from the outside.

**Interaction with Hadoop** $F$, running outside the enclave, provides a set of functions to $E^+$, running inside the enclave, over an interface system similar to the well-known *ioctl* calls. In this system, data is passed from/to the enclave over a shared memory region outside the enclave. Essentially, $F$ provides to $E^-$ the functions `readKeyValuePair()` and `writeKeyValuePair()` for reading and writing key-value pairs from and to Hadoop.

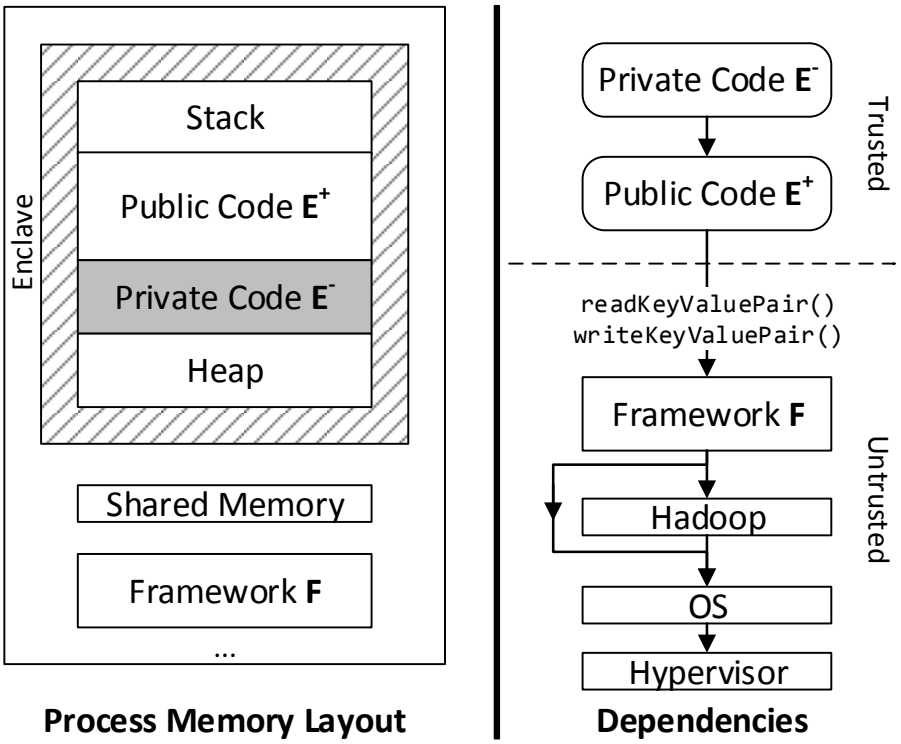


Figure 4.3: **Left:** Memory layout of process containing SGX enclave and framework code; **Right:** Dependencies between the involved components.

$F$ mediates invocations of these functions to unmodified Hadoop deployments via the standard *Hadoop streaming interface* [18]. Thus, *VC3* worker nodes appear as conventional worker nodes to Hadoop. Consequently, Hadoop can use its normal scheduling and fault-tolerance mechanisms to manage all data flows, including performance mechanisms for load balancing and straggler mitigation. All while Hadoop, the operating system, and the hypervisor are kept out of the TCB.

**Trusted Computing Base** On the hardware side, *VC3*'s TCB is only the SGX-enabled processor. This is a considerably smaller TCB than in systems based on TXT [108] or a TPM (large parts of the motherboard). A small hardware TCB is especially important in the Cloud setting where the hardware is largely controlled by the cloud provider.

Conceptually, instead of SGX, *VC3* could also be built on top of a trusted hypervisor [53, 101, 124, 130, 200] that provides isolation, sealing, and attestation functionality similar to SGX. However, establishing trust in a hypervisor is generally difficult for users in the Cloud setting; even when remote attestation of the integrity of the hypervisor is possible: hypervisors are large privileged pieces of software under the control of the cloud provider and are typically subject to periodic software updates. Any trusted hypervisor—in its entirety—necessarily becomes part of an application's software TCB in the Cloud setting. As such, any hypervisor-based *VC3* implementation would have a considerably

larger software TCB than the SGX-based implementation described in this chapter. For the latter, an application's software TCB is only composed of $E^+$ and $E^-$, which are both entirely chosen and compiled by the user. Whereas a trusted hypervisor would be chosen and compiled by the cloud provider. Naturally, a hypervisor-based *VC3* implementation would also be susceptible to common hardware-based attacks such as DMA.

## 4.5 Job Deployment

After preparing the code of their *VC3* application, users deploy it to the cloud provider. The code is then loaded into enclaves in worker nodes and it runs our key exchange protocol to get cryptographic keys to decrypt the *map* and *reduce* functions ($E^-$). After this, the worker nodes run our job execution and verification protocol. This section and the next present our cryptographic protocols for the exchange of keys and the actual MapReduce job execution, respectively. Before describing these protocols in detail, we first discuss the concept of *cloud attestation* used in *VC3*.

### 4.5.1 Cloud Attestation

As described above, in SGX, remote attestation for enclaves is achieved via *quotes* issued by QEs. The default SGX QE only certifies that the code is running on *some* genuine SGX processor, but it does not guarantee that the processor is actually located in the cloud provider's data centers. This may be exploited through variants of the *cuckoo attack* described by Parno [153]: an attacker could, for example, buy *any* SGX-enabled processor and conduct a long term physical attack to extract its master secret. If no countermeasures were taken, the attacker would then be in a position to impersonate any SGX-enabled processor in the provider's data centers. Note that the attacker model (see Section 4.1.1) in the Cloud setting excludes physical attacks only on the processors inside the cloud provider's data centers.

To defend against such attacks, *VC3* relies on auxiliary *Cloud QEs*. A Cloud QE is installed by the cloud provider (or a trusted third party) on all its SGX-enabled worker nodes before they enter operation. During the installation process, a Cloud QE generates a public/private key pair (e. g., for the EPID group signature scheme), outputs the public key, and seals the private key, which never leaves the Cloud QE in plain form.

The purpose of the Cloud QE is to complement quotes by the generic SGX QE, stating thereby that an enclave not only runs on a genuine SGX-enabled processor but also inside a certain data center or within certain geographical, jurisdictional, or other boundaries of interest to users. To protect against corrupted cloud providers, in *VC3*, quotes from the Cloud QE are always only used in conjunction with quotes from the SGX QE. (In the considered attacker model the cloud provider cannot fake quotes from the SGX QE since physical attacks on the processors inside its data centers are excluded.)

In the following, two fixed signing identities for SGX and for the cloud are assumed, we write $\mathsf{ESig}_{SGX}[C]\{text\}$ and $\mathsf{ESig}_{Cloud}[C]\{text\}$ for quotes by the SGX QE and the Cloud QE, respectively; for their concatenation $\mathsf{ESig}_{SGX}[C]\{text\} \mid \mathsf{ESig}_{Cloud}[C]\{text\}$ we write $\mathsf{ESig}_{SGX,Cloud}[C]\{text\}$.

### 4.5.2 Key Exchange

To execute the MapReduce job, enclaves first need to get keys to decrypt the code and the data, and to encrypt the results. This section describes the key exchange protocol used in *VC3* for this. This key exchange protocol is carefully designed such that it can be implemented using a conventional MapReduce job that works well with existing installations of the Hadoop MapReduce framework. In the following, the protocol is first described using generic messages. Subsequently, it is shown how to integrate the protocol with Hadoop.

Recall that the user is identified by her cryptographic public key $pk_u$, the public and private parts of the enclave code provided by her are denoted with $E^+$ and $E^-$, and each SGX-enabled processor runs a pair of SGX and Cloud QEs.

Before running the protocol itself, the user negotiates with the cloud provider an allocation of worker nodes for running a series of jobs. Setting up a new job involves three messages between the user and each worker node:

1. The user chooses a fresh job identifier $j$ and generates a fresh symmetric key $k_{code}$ to encrypt $E^-$ before sending to every node the packaged code of the *job enclave*:

$$C_{j,u} = E^+ \mid \mathsf{Enc}_{k_{code}}[]\{E^-\} \mid j \mid pk_u.$$

2. Each node $w$ starts an enclave with code identity $C_{j,u}$. Within the enclave, $E^+$ derives a symmetric key $k_w$ [2] and encrypts it under the user's public key:

$$m_w = \mathsf{PKEnc}_{pk_u}\{k_w\}.$$

   The enclave then requests quotes from the SGX and Cloud QEs for $m_w$, thereby linking $m_w$ to its code identity $C_{j,u}$ (and thus also to the job-specific $j$ and $pk_u$). The message $m_w$ and the quotes are sent back to the user:

$$p_w = m_w \mid \mathsf{ESig}_{SGX,Cloud}[C_{j,u}]\{m_w\}.$$

3. The user processes the message $p_w$ from each node $w$, as follows: the user verifies that both quotes attest that the message $m_w$ originates from an enclave with code identity $C_{j,u}$. Subsequently, the user decrypts $m_w$ and responds with the *job credentials* encrypted under the resulting node key $k_w$:

$$JC_w = \mathsf{Enc}_{k_w}[]\{k_{code} \mid \mathbf{k}\}$$

   where $k_{code}$ is the key that protects the code $E^-$ and

$$\mathbf{k} = k_{job} \mid k_{in} \mid k_{inter} \mid k_{out} \mid k_{prf}$$

   is the set of authenticated-encryption keys used in the actual job execution protocol (see Section 4.6). Specifically, $k_{job}$ is used to protect verification messages, $k_{prf}$ is used for keying the pseudo-random function $\mathsf{PRF}$, and $k_{in}$, $k_{inter}$, and $k_{out}$ are used to protect input splits, intermediate key-value pairs, and output key-value pairs, respectively.

[2] This can be the enclave's sealing key obtained through the instruction `EGETKEY` or a key generated using the random output of the standard x86-64 instruction `RDRAND`.

4. Each node resumes $E^+$ within the job enclave, which decrypts the job credentials $JC_w$ using $k_w$, decrypts its private code segment $E^-$ using $k_{code}$, and runs $E^-$.

On completion, the user knows that any enclave that contributes to the job runs the correct code, and that she shares the keys for the job with (at most) those enclaves.

For readability, an outline of the security theorem for the key exchange is given below, while the formal theorem statement, auxiliary definitions, and proof are given in Section 4.8.

**Theorem 1.** *Enclave and Job Attestation (Informally)*

1. *If a node completes the exchange with user public key $pk_u$ and job identifier $j$, then the user completed the protocol with those parameters; and*
2. *all parties that complete the protocol with $(pk_u, j)$ share the same job code $E^+$, $E^-$ and job keys in $\mathbf{k}$; and*
3. *the adversary learns only the encrypted size of $E^-$, and nothing about the job keys in $\mathbf{k}$.*

The protocol provides a coarse form of forward secrecy, inasmuch as neither the user nor the nodes need to maintain long-term private keys; the user may simply generate a fresh $pk_u$ in every session. The protocol can also easily be adapted to implement a Diffie-Hellmann key agreement, but this would complicate the integration with Hadoop described in the following.

#### 4.5.2.1 Integrating Key Exchange with Hadoop

Hadoop does not foresee online connections between nodes and the user, hence another mechanism is needed to implement the key exchange in practice. This section describes the *in-band* variant of key exchange, which is compatible with unmodified Hadoop installations and is implemented in our *VC3* prototype.

The *in-band* variant of key exchange is designed as a lightweight *key-exchange job* that is executed before the *actual job*. The online communication channels between nodes and user are replaced by the existing communication channels in a MapReduce job: *user* $\rightarrow$ *mapper* $\rightarrow$ *reducer* $\rightarrow$ *user*. By design, our in-band key exchange also does not require nodes to locally maintain state between invocations. (Per default, Hadoop does not foresee applications to store files permanently on nodes.) This is achieved by diverting the enclaves' unique and secret *sealing keys* from their common use. The exact procedure follows:

1. The user creates $C_{j,u}$ and the accompanying parameters for the *actual job* as described. The user then deploys this exact $C_{j,u}$ first for the special *key-exchange job*. It is necessary that the same $C_{j,u}$ is executed on the same nodes for both jobs.
2. When launched on a mapper or reducer node, $E^+$ obtains the enclave's unique *sealing key* (unique to the processor and digest of $C_{j,u}$, see Section 4.3.2) and uses it as its

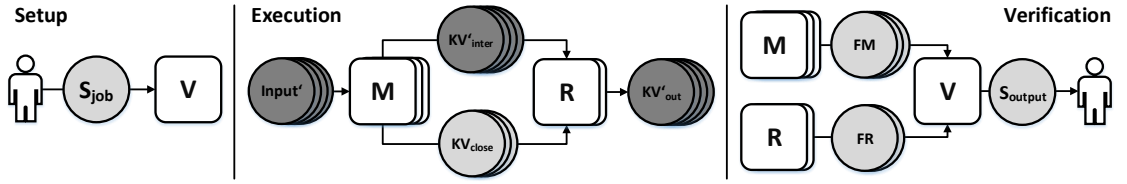


Figure 4.4: Schematic overview of the job execution protocol; the verifier (**V**), mappers (**M**), and reducers (**R**) are depicted as squares. A light-gray circle displays a message/key-value pair that is sent *once* by an entity; a dark-gray circle one that is sent multiple times. The user is depicted at both far ends.

node key $k_w$. Each node outputs the corresponding $p_w$ in the form of a MapReduce key-value pair. Mapper nodes immediately terminate themselves subsequently, while reducer nodes remain active until having forwarded all intermediate key-value pairs containing the mappers' $p_w$. $E^-$ is not (and cannot be) decrypted during the *key-exchange job*.

3. The user obtains all $p_w$ from the final outputs of the *key-exchange job*. The user creates the job credentials $JC_w$ for each node as described. Finally, the user writes $JC_w$ for all nodes to a file $D$ and deploys it together with $C_{j,u}$ for the *actual job*.

4. During the *actual job*, $E^+$ derives the unique sealing key (equivalent to $k_w$) on each node again and uses it to decrypt the corresponding entry in $D$, obtaining $k_{code}$ and $\mathbf{k}$. Afterward, $E^-$ is decrypted and the execution of the job proceeds as normal.

Note how it is essential to use the exact same $C_{j,u}$ in both jobs. Otherwise, the sealing keys used in the *key-exchange job* could not be re-obtained during the execution of the *actual job*. Thus, $E^+$ needs to implement the required functionality for both jobs.

## 4.6 Job Execution and Verification

After obtaining keys to decrypt the secret code and data, worker nodes need to run the distributed MapReduce computation. A naïve approach for protecting the computation would be to simply encrypt and authenticate all the key-value pairs exchanged between the nodes. A hostile cloud environment would though still be in the position to arbitrarily drop or duplicate data. This would allow for the manipulation of outputs. A dishonest cloud provider might also simply be tempted to drop data in order to reduce the complexity of jobs and thus to save on resources. Finally, even in the absence of bad intent, bugs or reliability issues in the cloud provider's software stack, e. g., in the employed MapReduce framework, could unnoticedly affect the correctness of a job's results. Furthermore, care has to be taken when encrypting data in a MapReduce job in order not to negatively impact the load-balancing and scheduling capabilities of Hadoop or the correctness of results. This section presents *VC3* job execution protocol, which tackles these problems and guarantees the overall integrity of a job and the confidentiality of data. As before, the protocol is described using generic messages before its integration with Hadoop is shown.

For now, the description of the protocol relies on a not further specified *verifier* that can communicate securely with the user and is trusted by the user. In practice, the verifier can run on a user's local machine or in an enclave.

Our implementation uses a distinct tag for each type of message; these tags are omitted below for simplicity. The entire protocol is implemented in $E^+$. Figure 4.4 gives a schematic overview of the message flows in the protocol.

### Step 1: Setup

As a preliminary step, the user uploads chunks of AEAD-encrypted data as *input splits* to the cloud provider. Each encrypted input split *Input* is cryptographically bound to a fresh, unique identifier (ID) $\ell_{in}$:

$$Input' = \mathsf{Enc}_{k_{in}}[\ell_{in}]\{Input\}$$

Our *VC3* implementation, uses the 128-bit MAC of the AES-GCM encryption as ID. Book keeping can though be simpler for incremental IDs. All encrypted input splits $Input'$ are stored by the cloud provider. The user decides on a subset of all available input splits as input for the job: $B_{in} = \{\ell_{in,0}, \ell_{in,1}, \ldots, \ell_{in,n-1}\}$; chooses a number of logical reducers for the job: $R$; and passes the job specification $S_{job} = j \mid k_{job} \mid R \mid B_{in}$ securely to the verifier. The number of mapper instances is not fixed *a priori* as Hadoop dynamically creates and terminates mappers while executing a job. We write $m \in \mathbf{m}$ for the mapper indexes used for the job. Note that this set of indexes is *a priori* untrusted; one goal of the protocol is to ensure that all reducers agree on it.

### Step 2: Mapping

Hadoop distributes input splits to running mapper instances. As input splits are encrypted, Hadoop cannot parse them for key-value pairs. Hence, the parsing of input splits is undertaken by $E^+$. Mappers keep track of the IDs of the input splits they process, and they refuse to process any input split more than once.

**Intermediate Key-Value Pairs** Mappers produce *intermediate* key-value pairs $KV_{inter} = \langle K_{inter} : V_{inter} \rangle$ from the input splits they receive. Hadoop assigns these to reducers for final processing according to each pair's key (the *shuffling* step). For the functional correctness of a job, it is essential that key-value pairs with identical keys are processed by the same reducer; otherwise the job's final output could be fragmented. However, the user typically has a strong interest in keeping not only the value $V_{inter}$ but also the key $K_{inter}$ of an intermediate key-value pair secret. Thus, our mappers wrap plaintext intermediate key-value pairs in encrypted intermediate key-value pairs $KV'_{inter}$ of the following form:

$$\begin{aligned} K'_{inter} &= r \equiv \mathsf{PRF}_{k_{prf}}(K_{inter}) \bmod R \\ V'_{inter} &= \mathsf{Enc}_{k_{inter}}[j \mid \ell_m \mid r \mid i_{m,r}]\{\langle K_{inter} : V_{inter} \rangle\} \\ KV'_{inter} &= \langle K'_{inter} : V'_{inter} \rangle \end{aligned}$$

By construction, it is $K'_{inter} \in 0..R-1$ and all intermediate key-value pairs $KV$ with the same key are assigned to the same logical reducer. The derivation of $K'_{inter}$ is similar to the standard *partitioning* step performed by Hadoop [16].

In the associated authenticated data above, $\ell_m$ is a secure unique job-specific ID randomly chosen by the mapper $m \in \mathbf{m}$ for itself (our implementation uses the x86-64 instruction `RDRAND` inside enclaves); $r$ is the reducer index for the key; and $i_{m,r}$ is the number of key-value pairs sent from this mapper to this reducer so far. Thus, $(\ell_m, r, i_{m,r})$ uniquely identifies each intermediate key-value pair within a job. Note that, in practice, many plaintext $KV_{inter}$ from one mapper to one reducer may be batched into a single $KV'_{inter}$.

**Mapper Verification** For verification purposes, after having processed all their inputs, our mappers also produce a special *closing intermediate key-value pair* for each $r \in R$:

$$KV_{close} = \langle r : \mathsf{Enc}_{k_{inter}}[j \mid \ell_m \mid r \mid i_{m,r}]\{\}\rangle$$

This authenticated message ensures that each reducer knows the total number $i_{m,r}$ of intermediate key-value pairs (zero or more) to expect from each mapper. In case a reducer does not receive exactly this number of key-value pairs, or receives duplicate key-value pairs, it terminates itself without outputting its final verification message (see next step).

Furthermore, each mapper sends the following final verification message to the verifier:

$$FM = \mathsf{Enc}_{k_{job}}[j \mid \ell_m \mid B_{in,m}]\{\}$$

where $B_{in,m}$ is the set of IDs of all input splits the mapper $m \in \mathbf{m}$ processed. This authenticated message lets the verifier aggregate information about the distribution of input splits.

## Step 3: Reducing

Assuming for now that Hadoop correctly distributes all intermediate key-value pairs $KV'_{inter}$ and $KV_{close}$, reducers produce and encrypt the final *output key-value pairs* for the job:

$$KV'_{out} = \langle \ell_{out} : \mathsf{Enc}_{k_{out}}[\ell_{out}]\{KV_{out}\}\rangle$$

where $KV_{out}$ is a split of final output key-value pairs, with secure unique ID $\ell_{out}$. Again, our implementation uses the MAC of the AES-GCM encryption as unique ID. By design, the format of $V'_{out}$ is compatible with the format of encrypted input splits, allowing the outputs of a job to be immediate inputs to a subsequent one.

**Reducer Verification** After having successfully processed and verified all key-value pairs $KV'_{inter}$ and $KV_{close}$ received from mappers, each reducer sends a final verification message to the verifier:

$$FR = j \mid r \mid B_{out,r} \mid \mathsf{Enc}_k(j \mid r \mid B_{out,r} \mid P_r, \{\})$$
$$P_r \subseteq (\ell_m)_{m \in \mathbf{m}}$$

The authenticated message $FR$ carries the set $B_{out,r}$ of IDs $\ell_{out}$ for all outputs produced by the reducer with index $r \in R$. It also authenticates a sorted list $P_r$ of mapper IDs, one for each closing intermediate key-value pair it has received. (To save bandwidth, $P_r$ is authenticated in $FR$ but not transmitted.)

### Step 4: Verification

The verifier receives a set of $FM$ messages from mappers and a set of $FR$ messages from reducers. To verify the global integrity of the job, the verifier proceeds as follows:

1. The verifier checks that it received exactly one $FR$ for every $r \in 0..R-1$.
2. The verifier collects and sorts the mapper IDs $P_{verifier} \subseteq (\ell_m)_{m \in \mathbf{m}}$ from all received $FM$ messages, and it checks that $P_{verifier} = P_r$ for all received $FR$ messages, thereby ensuring that all reducers agree with $P_{verifier}$ (i. e., the verifier checks that all reducers spoke to the same mappers).
3. The verifier checks that the sets $B_{in,m}$ received from the mappers form a partition of the input split IDs of the job specification, thereby guaranteeing that every input split has been processed once.
4. Finally, the verifier accepts the union of the sets received from the reducers, $B_{out} = \bigcup_{r \in 0..R-1} B_{out,r}$, as the IDs of the encrypted job output.

The user may download and decrypt this output, and may also use $B_{out}$ in turn as the input specification for another job (setting the new $k_{in}$ to the previous $k_{out}$).

## 4.6.1 Security Discussion

We outline below our security theorem for the job execution and subsequently discuss the protocol informally. The formal theorem statement, auxiliary definitions, and proof are given in Section 4.8.

**Theorem 2.** *Job Execution (Informally)*

1. *If the verifier completes with a set of output IDs, then the decryptions of key-value pairs with these IDs (if they succeed) yield the correct and complete job output.*
2. *Code and data remains secret up to traffic analysis: The adversary learns at most (i) encrypted sizes for code, input splits, intermediate key-value pairs, and output key-value pairs; and (ii) key-repetition patterns in intermediate key-value pairs.*

Observe how if the verifier completes with a set of output IDs, then the decryptions of key-value pairs with these IDs (if they succeed) yield the correct and complete job output. For each cryptographic data key, AEAD encryption guarantees the integrity of all messages exchanged by the job execution protocol; it also guarantees that any tampering or truncation of input splits will be detected. Each message between mappers, reducers,

and verifier ($KV'_{inter}$, $KV_{close}$, $FM$, and $FR$) includes the job-specific ID $j$, so any message replay between different jobs is also excluded.

Thus, the adversary may at most attempt to duplicate or drop some messages within the same job. Any such attempt is eventually detected as well: if the verifier does not receive the complete set of messages it expects, verification fails. Otherwise, given the $FM$ messages from the set $\mathbf{m}'$ of mappers, it can verify that the mappers with distinct IDs $(\ell_m)_{m \in \mathbf{m}'}$ together processed the correct input splits. Otherwise, if any inputs splits are missing, verification fails. Furthermore, given one $FR$ message for each $r \in 0..R-1$, the verifier can verify that every reducer communicated with every mapper. Given $R$, the verifier can also trivially verify that it communicated with all reducers that contributed to the output.

Reducers do not know which mappers are supposed to send them key-value pairs. Reducers though know from the $KV_{close}$ messages how many key-value pairs to expect from mappers they know of. Accordingly, every reducer is able to locally verify the integrity of all its communication with every mapper. Although the adversary can remove or replicate entire streams of mapper/reducer communications without being detected by the reducer, this would lead to an incomplete set $P_r$ of mapper IDs at the reducer, eventually detected by the verifier.

### 4.6.2 Analysis of Verification Cost

The cost for the verification of a job with $M$ mappers and $R$ reducers is now analyzed. *VC3*'s full runtime cost is experimentally assessed in §4.10.

There are $M + R$ verification messages that mappers and reducers send to the verifier. These messages most significantly contain for each mapper the set $B_{in,m}$ of processed input split IDs and for each reducers the set $B_{out,r}$ of IDs of produced outputs. Each ID has a size of 128 bits. Typically, input splits have a size of 64 MB or larger in practice. Hence, mappers need to *securely* transport only 16 bytes to the verifier per 64+ MB of input. As reducers should batch many output key-value pairs into one $KV'_{out}$, a similarly small overhead is possible for reducer/verifier communication. There are $M \times R$ verification messages sent from mappers to reducers. These messages are small: they contain only four integers.

The computational cost of *verification* amounts to the creation and verification of the MACs for all $M + R + M \times R$ verification messages. Additionally, book keeping has to be done by all entities. We consider the cost for *verification* to be small.

### 4.6.3 Integrating the Verifier with Hadoop

For the job execution protocol it is again desirable to avoid online connections between the involved entities. Hence, a variant of the protocol that implements an *in-band* verifier as a simple MapReduce job is described in the following. Our *VC3* prototype implements this variant of the protocol.

In the refined variant of the protocol, mappers send their $FM$ messages in the form of key-value pairs to reducers (instead of sending them to a central verifier). Reducers output all $FM$ key-value pairs they receive from mappers unaltered and also output their

own *FR* messages in the form of key-value pairs (again, instead of sending them to a central verifier). After the actual job terminated, a subsequent *verification job* is executed.

The verification job is given $S_{job}$ of the *actual job* and is invoked on the entire corresponding outputs. The mappers of the verification job parse input splits for *FM* and *FR* messages and forward them to exactly one *verification reducer* by wrapping them into key-value pairs with a predefined key $K'_{inter}$. On success, the verification reducer outputs exactly one key-value pair certifying $B_{out}$ as valid output for $S_{job}$. This key-value pair can finally easily be verified by the user. In practice, the verification job can be bundled with a regular job that already processes the outputs to be verified while parsing for verification messages. In such a case, one of the regular reducers also acts as verification reducer (the *VC3* prototype assigns this task to the reducer with $r = 0$). The bundled job in turn creates its own verification messages *FM* and *FR*. This way, it is possible to chain an arbitrary number of secure MapReduce jobs, each verifying the integrity of its immediate successor with low overhead.

## 4.7 Discussion

In this section, several attack scenarios on *VC3* are discussed, which are partly outside the attacker model from Section 4.1.1.

### 4.7.1 Information Leak through the Distribution of Intermediate Key-Value Pairs

One basic principle of MapReduce is that all intermediate key-value pairs with the same intermediate key have to be processed by the same reducer. An attacker able to observe network connections between worker nodes can count the number of pairs delivered to each reducer. As soon as there is more than one reducer, the attacker thus directly learns the distribution of intermediate keys. In the following, the extent of this information leak is discussed in more detail:

For the whole job, each key $K_{inter}$ is mapped to a fixed, uniformly-sampled value $K'_{inter} \in 0..R-1$, where $R$ is the number of reducers for the job chosen by the user (see Section 4.6). For each intermediate key-value pair, the attacker may observe the mapper, the reducer, and $K'_{inter}$. Intuitively, the smaller the overall number of unique intermediate keys $K_{inter}$ in relation to $R$, the more the attacker may learn on the actual distribution of intermediate keys. For example, in the case of a presidential election vote count, there are only two possible intermediate keys (the names of both candidates). If $R > 1$, then the attacker easily learns the distribution of the votes but not necessarily the name of the successful candidate. Conversely, if there are many intermediate keys (each with a small number of corresponding intermediate key-value pairs) relative to $R$, then leaking the total number of pairs dispatched to each reducer leaks relatively little information. In particular, when all intermediate keys are unique, no information is leaked.

Attackers may also use more advanced traffic-analysis techniques against *VC3* [50, 195, 225]. For example, by observing traffic, an attacker may correlate intermediate key-value pairs and output key-value pairs to input splits; over many runs of different jobs this may reveal substantial information about the secret contents of input splits. However, the

circumstance that in *VC3* the private enclave code $E^-$, which implements the map and reduce functions, remains unknown to the attacker could complicate attacks in practice.

At the time of this writing, Ohrimenko et al. were about to release a paper [146] which explores *VC3*'s vulnerability to traffic-analysis attacks in more detail and proposes two different extensions to *VC3* that tackle such attacks at the cost of runtime performance.

### 4.7.2 Replay Attacks

The attacker could try to profit in various ways from fully or partially replaying a past MapReduce job. Such replay attacks are generally prevented in case the *online* key exchange (Section 4.5.2) is employed, as the user can simply refuse to give $JC_w$ a second time to any enclave. This is different for the *in-band* version of our approach (Section 4.5.2.1): an enclave is not able to tell if it ran on a set of input data before as it cannot securely keep state between two invocations. (The adversary can always revert a sealed file and reset the system clock.) Given $C_{j,u}$ and $JC_w$ corresponding to a certain processor under their control, the attacker is in the position to arbitrarily replay parts of a job that the processor participated in before or even invoke a new job on any input splits encrypted under $k_{in}$ contained in $JC_w$. This allows the attacker to repeatedly examine the runtime behavior of $E^-$ from outside the enclave and thus to amplify other side-channel attacks against *confidentiality*.

The resilience of *VC3* against such attacks can be enhanced by hardcoding a job's specification into mappers to restrict the input splits they should accept to process. Also, Strackx et al. proposed an extension to SGX that provides *state continuity* for enclaves [199] and, if adopted, could be leveraged in *VC3* to largely prevent replay attacks.

### 4.7.3 Vulnerabilities in Enclave Code

Like other software written in C++, the enclave code of a *VC3* job may suffer from classic software vulnerabilities such as *unsafe memory accesses* or may even contain *backdoors*. Hence, the adversarial settings Classic and Backdoor, which were discussed in chapters 2 and 3, are considered for *VC3*'s enclave code package $C_{j,u}$ (including $E^+$ and $E^-$) in the following.

**Passive Attacker** By design, code within an enclave can access the entire address space of its host process. This enables the implementation of efficient communication channels with the outside world as discussed in Section 4.3.2, but also broadens the attack surface of enclaves. If enclave code, due to a memory access-related programming error, ever dereferences a corrupted pointer to untrusted memory outside the enclave, compromise of different forms becomes possible: the untrusted runtime environment (including the operating system etc.) can catch and handle the corresponding exception without the enclave noticing. In case an erroneous dereference is made in a reading operation, the untrusted environment is in the position to inject arbitrary data into the enclave. Such a data injection may in the simplest case affect the correctness of computations, but may also, depending on the nature of the corrupted pointer, pave the way for a classic control-

flow hijacking attack (see Section 2.2) against the enclave. Conversely, if a corrupted pointer is dereferenced in a writing operation, data immediately leaks outside the enclave.

Other than what is usually the case in the CLASSIC setting, *null pointer dereferences* are of special concern in the context of SGX: uninitialized or invalid pointers are commonly set to `0` in C/C++ and consequently, null pointer dereferences at runtime are a common error. To prevent exploitation, modern operating systems often disallow allocating virtual memory around address `0` in user mode processes. Accordingly, null pointer dereferences are often treated as minor annoyance rather than a threat for most application software today. This is different for SGX enclaves, as an adversarial operating system may very well map memory pages at and around address `0`. Hence, null pointer dereferences from within enclaves are a serious threat.

**Active Attacker** The attacker may also actively misuse the communication channel between $E^+$ and the external component $F$ to trigger and exploit a memory access error, e. g., a buffer overflow, in the enclave. (Recall that $F$ outside the enclave, among others, makes the functions `readKeyValuePair()` and `writeKeyValuePair()` available to $E^+$ inside the enclave.) This scenario resembles very much the CLASSIC setting: technically, from the attacker's perspective, exploiting a vulnerability in the input handling code of a local *VC3* enclave is similar to exploiting such a vulnerability in a remote server application. As the NX bit (see Section 2.2.3) is available for enclave pages [3], we expect meaningful control-flow hijacking attacks against enclave code to leverage code-reuse techniques such as ROP (see Section 2.2.4.2) or COOP (see Section 2.5). In general, code-reuse attacks work the same in the context of enclaves as they do for other software.

In line with the BACKDOOR setting, the attacker may also attempt to trigger a mischievously installed backdoor in $E^+$ or $E^-$; either through the explicit communication channel between $E^+$ and $F$ or a hidden, implicit channel. In the case of enclaves, many possible hidden channels exist from and to the outside world. For example, a backdoor in enclave code could stay dormant until a certain sequence of enclave page evictions (performed by the operating system) is observed. A backdoor in enclave code could for example manipulate or leak sensitive data directly—such as cryptographic key—or allow the attacker to inject her own malicious code into the enclave.

**Countermeasures** To prevent the exploitation of erroneous memory accesses made by enclave code, effective forms of memory safety should be enforced inside *VC3* enclaves. However, existing memory safety solutions for C/C++ typically incur high runtime performance overhead (see Section 2.2.3) and may also be less effective within the enclave environment. For example, the CPI approach (see Section 2.5.6.5) requires metadata associated with sensitive pointers to be stored in a so called *safe region*. To prevent an attacker from manipulating this sensitive metadata, on x86-64, CPI maps the safe region at a randomly-chosen secret location in the virtual address space of a protected application's process. Whereas the virtual address space of a process on x86-64 is effectively $2^{48}$

[3] In our implementation of *VC3*, all data pages in the enclave are marked as non-executable.

bytes large, the size of SGX enclaves is typically dramatically smaller [4], which would likely considerably facilitate guessing or deducing the secret location of a CPI safe region within an enclave for an attacker.

To tackle erroneous memory accesses effectively, we extended a recent version of the Microsoft C++ compiler to optionally enforce either one of the following two invariants for enclave code:

- *Region-write-integrity:* writes through pointers can only go to *(i)* address-taken[5] local variables on the stack, *(ii)* address-taken global variables, or *(iii)* allocations from the enclave heap. To detect buffer overflows/underflows from one writable memory region to another, non-writable (dummy) regions are placed between writable ones. To guarantee that compiler inserted checks can never be skipped, a coarse CFI policy (see Section 2.2.5.2) is enforced guaranteeing that indirect branches (calls or jumps) may only reach address-taken code locations.

- *Region-read-write-integrity:* the same guarantees as for *region-write-integrity* are given; additionally, memory from outside the enclave cannot be read.

Whenever, at compile time, an indirect writing/reading operation or indirect control-flow transfer cannot be proven to be always safe, the extended compiler adds a lightweight check before the write, read, or branch instruction in question. These checks examine data and code pointers at runtime and enforce the given invariant. Whenever a violation is encountered, the enclave is terminated immediately. In particular, the write checks prevent the corruption of all compiler-generated data including return addresses stored on the stack. This property together with the enforced CFI policy prevents the enclave's control flow from ever skipping the installed write/read checks in the event of a code pointer corruption. However, this guarantee only holds as long as the stack pointer is not corrupted; otherwise, fake return addresses may divert the control flow to arbitrary code locations. (Note that a corruption of the stack pointer is unlikely to occur, because control data such as saved stack frames is always write-protected and techniques to move the stack pointer like the one described in Section 2.5.1.4 for 32-bit COOP do not work for all common x86-64 calling conventions.)

The implementation of the runtime checks is lightweight and shares some ideas with the WIT technique [8] by Akritidis et al.: two bitmaps are maintained at runtime to keep track of memory locations inside the enclave that are legitimate targets for indirect writes or for indirect control-flow transfers, respectively. Hence, corresponding runtime checks have the form of a simple bitmap look up. The two bitmaps map each 8-byte/16-byte chunk in the address space of the enclave to a single bit flag (indicating if the chunk is a legitimate target for an indirect write or branch, respectively). Accordingly, the "write" bitmap occupies $\frac{1}{64}$th of an enclave's memory and the "branch" bitmap $\frac{1}{128}$th. The checks on reads are implemented in an even simpler manner: a static bitmask is used to determine if the target address lies within the enclave or not. As such, it may surprise that in practice

[4] At the time of this writing, the maximum size of SGX enclaves was still unknown. Our *VC3* prototype uses enclaves with a size of 512 MB ($2^{29}$ bytes).

[5] We use the term "address-taken" to refer to a variable to which an implicit or explicit pointer exists in the source code. For example, every C-style array is address-taken.

the runtime checks on reads generally have a bigger impact on performance than the checks on writes. This is due to the circumstance that programs in general perform significantly more reads than writes through pointers. The *region-write-integrity* and *region-read-write-integrity* options and their implementations are described in more detail in the original *VC3* publication [176] alongside an experimental evaluation of their performance.

Among others, the *region-write-integrity* option prevents enclave code from writing to memory outside the enclave and *region-read-write-integrity* additionally prevents reading from the outside. This makes accidental data leakage and passive data injection attacks as described above impossible. However, for the communication channel between $E^+$ and $F$ to work, certain code parts of $E^+$ must of course still be able to read and write data from and to outside memory. Accordingly, when one of the two options is chosen by the user, $E^+$ is compiled such that a single function (`copyAcrossBorder()`) is not augmented with write or read checks. This function is used by other functions in $E^+$ to *explicitly* copy data from and to the enclave. In essence, `copyAcrossBorder()` is used in `readKeyValuePair()` and `writeKeyValuePair()` in $E^+$. We remark that `copyAcrossBorder()` is not address-taken and is thus not a valid target for indirect branches; accordingly, control flow cannot accidentally reach it through a corrupted code pointer. (In the consequence, data leakage or injection cannot accidentally occur through `copyAcrossBorder()`.)

With *region-read-write-integrity* in place, the `readKeyValuePair()`/`writeKeyValuePair()` interface between $E^+$ and $F$ remains as the sole avenue for attacks from the outside apart from side channels. By interacting with this interface, the attacker can still attempt to exploit vulnerabilities that are not prevented by the *region-write-integrity* invariant such as most temporal memory errors (see Section 2.2) or certain non-sequential spatial memory errors, e. g., an array access where the index is under attacker control. In the next step, the attacker could attempt to launch a code-reuse attack. Given the enforced CFI policy and the protection of return addresses, the resilience of *region-write-integrity* against code-reuse attacks can be considered roughly the same as of the original two-label CFI implementation by Abadi et al. [1] in combination with a shadow call stack (see sections 2.2.5.2 and 2.5.6.1). Hence, ROP-based attacks are unlikely to succeed against protected enclave code, whereas COOP remains largely unaffected by *region-write-integrity*. Nonetheless, even in a COOP-style attack, the guarantees provided by *region-write-integrity* or *region-read-write-integrity* cannot be violated.

However, it is worth noting that the `readKeyValuePair()`/`writeKeyValuePair()` interface presents in practice a rather narrow attack surface, because *VC3* enclave code expects all external key-value pairs to be properly AEAD-encrypted under a job's $k_{in}$ (for mappers) or $k_{inter}$ (for reducers). As attackers in the Classic setting cannot forge valid AEAD-encrypted key-value pairs, they can essentially only hope to trigger and exploit vulnerabilities in those code parts in $E^+$ that handle/parse encrypted key-value pairs prior to authentication. Ideally, these exposed code parts should be proven to be functionally correct using techniques from the realm of formal verification (see e. g., [120]). This complex task is left for future work.

Coping with backdoors in $E^+$ or $E^-$ is generally challenging, as these two components constitute the software TCB of *VC3*. The techniques for the detection and dismantling of backdoors in server applications presented in Chapter 3 are a *best effort* approach, which does not provide any "hard" guarantees. In order to reduce the attack surface

for backdoors in trusted enclave code in a more dependable way, it could be further partitioned into "trusted" and "less trusted" code parts using *software-based fault isolation* (SFI) [77,218]. On the baseline, SFI approaches isolate cooperating code modules running in the same address space from each other such that faults in one module (e. g., a memory access error) are guaranteed to not affect other modules. In general, SFI is implemented by inserting additional runtime checks into code, e. g., checks on memory accesses or control-flow transfers. As such, our *region-write-integrity* and *region-read-write-integrity* invariants can also be considered to be forms of SFI.

A practical SFI-related approach to mitigate the risk of backdoors in $E^+$ or $E^-$ would be to enforce at runtime that enclave memory regions holding secret cryptographic keys (essentially $k_w$ and the keys in $\mathbf{k}$) are only accessible from a well-defined set of privileged cryptographic functions. With such a measure in place, any backdoor in "less trusted" enclave code could at least not leak or manipulate those keys, which constitute the most confidential assets of a *VC3* enclave. We expect that this can be implemented with moderate effort and negligible additional performance overhead atop of the existing *region-read-write-integrity* implementation.

## 4.8 Additional Definitions, Theorems, and Proofs for Key Exchange and Job Execution

This section provides additional definitions, theorems and security proofs for *VC3*'s *key exchange* (Section 4.5.2) and *job execution* (Section 4.6) protocols. We begin with the description of our model of SGX.

### 4.8.1 Modeling SGX

In SGX, *sealing keys* are derived from a secret hardwired in the processor, the digest of the respective enclave, and possibly other ingredients [109]. We denote the node-specific processor secret as $s_w$ and model it as a bit string of length $\lambda$. Furthermore, we model the derivation of the sealing key within the enclave with code identity $C_{j,u}$ running on node $w$ as $k_w = \mathsf{PRF}_{s_w}(C_{j,u})$.

For simplicity, our formal development considers deployments with just one quote from the SGX QE, for a fixed, trusted signing group. The proof directly extends to multiple quotes, as long as *one* of the signing groups is trustworthy.

As explained in Section 4.3.3, we assume that the EPID signature scheme employed by QEs is UF-CMA, as shown by Brickell and Li [42], and that the local attestation protocol between enclaves and QEs is secure. Thus, any quote with a given code identity must have been requested by that code.

### 4.8.2 Key Exchange

We provide a security proof for *VC3*'s key exchange protocol by reduction to the cryptographic assumptions listed in Section 4.3.3. We model the protocol of Section 4.5.2 as a game involving a probabilistic, polynomial-time adversary $\mathcal{A}$ (which intuitively includes the cloud and the network, as explained in Section 4.1.1).

$\mathcal{A}$ is initially given the public group key of the SGX QE. $\mathcal{A}$ can call oracles (defined below) that specify the user and the SGX nodes, any number of times, in any order, but it cannot directly access their private state.

For code confidentiality, we further assume that the private part of the code $E^-$ ranges over AEAD plaintexts of the same lengths (that is, that AEAD hides their lengths, possibly after padding), and we assume given some fixed private code $E_0^-$, used as a "dummy" for specifying code confidentiality.

We let $\mathbf{k} = k_{job} \mid k_{in} \mid k_{inter} \mid k_{out} \mid k_{prf}$ represent the keys for a job.

- $pk_u \leftarrow \mathsf{User.Gen}()$ samples a new user public-key pair $pk_u, sk_u \xleftarrow{\$} \mathsf{PKGen}()$, stores $sk_u$, and returns $pk_u$ to $\mathcal{A}$. By definition, the *honest user keys* are those returned by $\mathsf{User.Gen}$.

- $w \leftarrow \mathsf{Node.Gen}()$ generates a new certified SGX node with public ID $w$, samples and stores the processor secret $s_w \xleftarrow{\$} \{0,1\}^\lambda$, and provisions a QE for the node. In our simple model, all these nodes are honest.

- $C_{j,u} \leftarrow \mathsf{User.Init}(pk_u, E^+, E^-, N)$ starts a user session intended to run the key exchange protocol with $N$ nodes; it fails if (i) $pk_u$ is not an honest user key, (ii) $E^+, E^-$ is not valid, well-behaving, and functionally correct *VC3* code, or (iii) $N$ is larger than a certain $N_{max} \in \mathbb{Z}^+$; otherwise it samples

$$\begin{aligned} k_{code} &\xleftarrow{\$} \{0,1\}^\iota \\ \mathbf{k} &\xleftarrow{\$} \{0,1\}^\iota \times \{0,1\}^\iota \times \{0,1\}^\iota \times \{0,1\}^\iota \times \{0,1\}^\theta \\ j &\xleftarrow{\$} \{0,1\}^\kappa \end{aligned}$$

  where $j$ is the *job ID*, $\iota$ is the key length of the AEAD scheme, and $\theta$ is the length of $k_{prf}$. (For AES-GCM, $\iota = 128$ is fixed.) The values $E^+$, $E^-$, $N$, $k_{code}$, $j$, $\mathbf{k}$ are recorded for the user session indexed by $(pk_u, j)$ and

$$C_{j,u} = E^+ \mid \mathsf{Enc}_{k_{code}}[]\{E^-\} \mid j \mid pk_u$$

  is returned as the first protocol message.

- $* \leftarrow \mathsf{Node.Execute}(w, C, *)$ fails if $w$ is not an honest node; otherwise, it runs a new enclave with code $C$. (We write $*$ for optional arguments and results.)

  In case $C$ is valid *VC3* code $C_{j,u}$ for some user session, in particular, $E^+$ derives $k_w = \mathsf{PRF}_{s_w}(C_{j,u})$. Next, depending on the optional argument $*$, one of the following happens:

  (a) if no argument is passed, $E^+$ creates $m_w = \mathsf{PKEnc}_{pk_u}\{k_w\}$, obtains a quote for this text from the QE, and returns the second message:

$$p_w = m_w \mid \mathsf{ESig}_{SGX}[C_{j,u}]\{m_w\}$$

(b) if an argument $JC_w$ is passed, $E^+$ attempts to AEAD-decrypt it the job credentials $k_{code} \mid \mathbf{k}$. If decryption succeeds, $k_{code}$ is used to AEAD-decrypt $E^-$. On success, the node $w$ completes the key exchange with $(pk_u, j)$, $E^+$, $E^-$, $\mathbf{k}$.

In all other cases for $C$, we do not need to model the details of the enclave execution; the adversary may load arbitrary code into enclaves, and that code may in particular request quotes for arbitrary text from the local QE with code identity $C$ and return these to $\mathcal{A}$.

- $(JC_w)_{w \in \mathbf{w}} \leftarrow$ User.Complete$(pk_u, j, (p_w)_{w \in \mathbf{w}})$ checks that all steps below succeed, and fails otherwise:
  - There exists a user session indexed by $pk_u, j$ for $N$ nodes such that $N = |\mathbf{w}|$.
  - Every $p_w$ consists of a text and a valid QE quote for that text with code identity $C_{j,u}$. (Recall that $C_{j,u}$ includes $j$ and $pk_u$.)
  - Every such text decrypts to a unique $k_w$ using $sk_u$.

  It then issues the third messages of the protocol $JC_w = \mathsf{Enc}_{k_w}[]\{k_{code} \mid \mathbf{k}\}$ for each $k_w, w \in \mathbf{w}$, and records user completion with $(pk_u, j)$, $N$, $E^+$, $E^-$, and $\mathbf{k}$.

(In the node oracles, we use $w$ only as an index for the adversary; conversely, users identify nodes only by their job-specific keys $k_w$.)

Our theorem captures the intended integrity and privacy properties of the key exchange protocol. Its proof provides a more precise, concrete security bound.

**Theorem 1** (Key Exchange)**.** *Consider a game with the adversary $\mathcal{A}$ calling the oracles defined above.*

- ***Agreement:*** *Except for a negligible probability, if a node completes with $(pk_u, j)$, $E^+$, $E^-$, $\mathbf{k}$ and $pk_u$ is honest, then (i) a user completed with $(pk_u, j)$, $N$, $E^+, E^-$, $\mathbf{k}$ and (ii) at most $N-1$ other nodes completed with $(pk_u, j)$ and they all have matching parameters $E^+$, $E^-$, and $\mathbf{k}$.*
- ***Privacy:*** *Consider an indistinguishability variant of the game above, where we initially sample $b \leftarrow \{0, 1\}$; where, only if $b = 1$,*
  1. *we substitute the same fresh random values for the job keys $k_{job}$, $k_{in}$, $k_{inter}$, $k_{out}$, and $k_{prf}$ in all completions indexed by $pk_u, j$;*
  2. *we substitute some fixed code $E_0^-$ for all $E^-$ passed to* User.Init*;*

  *and where $\mathcal{A}$ finally returns some value $g$. The advantage of $\mathcal{A}$ guessing $b$ (that is, the probability that $b = g$ minus $\frac{1}{2}$) is negligible.*

*Proof.* The proof is by reduction to standard security assumptions on the algorithms of the protocol, using a series of cryptographic games [30, 190].

We consider a series of games adapted from the theorem statement, each defined from the previous one by idealizing some part of the protocol. At each step, we bound the probability that an adversary distinguishes between the two successive games. For game $i$, we write $p_i$ for the (maximum of) the probability that $\mathcal{A}$ breaks one of the two properties of the theorem.

**Game 0** is defined in the theorem statement, with an adversary $\mathcal{A}$ that queries User.Gen $\alpha_0$ times, User.Init $\alpha_1$ times, User.Complete $\alpha_2$ times, Node.Gen $\beta_0$ times, Node.Execute $\beta_1$ times, and QE quotes (within Node.Execute) $\gamma$ times.

**Game 1 (Collisions)** is as above, except that we exclude collisions (i) between two honest keys $pk_u$, (ii) between two job IDs $j$ in user sessions, or (iii) between two processor secrets $s_w$. By reasoning on probabilities conditioned by these events, we have

$$p_0 \leq p_1 + \epsilon^{\mathsf{PKGen}}_{collision}(\alpha_0) + \frac{\alpha_1^2}{2^\kappa} + \frac{\beta_0^2}{2^\lambda}$$

where $\epsilon^{\mathsf{PKGen}}_{collision}(\alpha_0)$ denotes the probability that the public-key key pair generation algorithm PKGen produces a collision on public keys for $\alpha_0$ invocations. (This probability is smaller than $\alpha_0$ times the probability of breaking IND-CPA security for that scheme.)

From this game, we can in particular use $(pk_u, j)$ as a unique index for each user session.

**Game 2 (UF-CMA)** is as above, except that User.Complete$(pk_u, j, (p_w)_{w\in\mathbf{w}})$ fails on receiving any quote $\mathsf{ESig}_{SGX}[C_{j,u}]\{m^\star_w\}$ without a matching quote request from Node.Execute with code identity $C_{j,u}$. $\mathcal{A}$ can distinguish between this game and the previous one only if it can forge a QE quote attributed to $C_{j,u}$. This happens at most with probability $\epsilon_{\mathsf{UF\text{-}CMA}}(\gamma)$, the probability of breaking our unforgeability assumption on the quoting scheme given at most $\gamma$ chosen-text quotes, hence we have

$$p_1 \leq p_2 + \epsilon_{\mathsf{UF\text{-}CMA}}(\gamma)$$

**Game 3 (PRF)** is as above, except in Node.Execute, where we replace the key $k_w$ derived from $s_w$ with a key $k_w \xleftarrow{\$} \{0,1\}^\iota$ sampled for each node $w$ created by Node.Gen and each user sessions $(pk_u, j)$ (that is, we replace PRF with a perfect random function).

Note that we write $k_w$ for the key associated with $(pk_u, j)$ at node $w$; we keep this additional indexing implicit when it is clear from the context.

Let $\epsilon^{\mathsf{PRF}}_{distinguish}(\lambda, n)$ bound the advantage of an adversary breaking our PRF assumption in $n$ calls to PRF with a random seed of size $\lambda$. For each node, $s_w$ is used only for keying the PRF, hence the adversary distinguishes between the key derivation in Game 2 and the randomly sampled key in Game 3 only with advantage $\epsilon^{\mathsf{PRF}}_{distinguish}(\lambda, \beta_1)$. We obtain

$$p_2 \leq p_3 + \beta_0 \cdot \epsilon^{\mathsf{PRF}}_{distinguish}(\lambda, \beta_1)$$

**Game 4 (IND-CPA with $pk_u$)** is as above, except for public key encryptions of $k_w$ keys. For each honest user key $pk_u$, we maintain a table $T_{pk_u}$ from ciphertexts to plaintexts; (1) Node.Execute encrypts a dummy value $0^\iota$ instead of $k_w$ and records in $T_{pk_u}$ the resulting ciphertext and $k_w$; and (2) Node.Complete first attempts to retrieve $k_w$ by a lookup in $T_{pk_u}$, and decrypts only if the lookup fails.

Node.Complete decrypts only after verifying the quote, which guarantees that Node.Execute has entered the quoted ciphertext in the table, so (2) never actually decrypts. By

definition of IND-CPA security for $pk_u$, the adversary distinguishes between concrete encryptions and dummy encryptions only with probability $\epsilon^{\mathsf{AEAD}}_{\mathsf{IND\text{-}CPA}}(\beta_1)$, since $\beta_1$ bounds the number of encryptions under $pk_u$ within Node.Execute. We get

$$p_3 \leq p_4 + \alpha_0 \cdot \epsilon^{\mathsf{AEAD}}_{\mathsf{IND\text{-}CPA}}(\beta_1)$$

Now that $k_w$ is sampled at random and used only for keying AEAD, we are ready to apply INT-CTXT and IND-CPA assumptions on it.

**Game 5 (INT-CTXT with $k_w$)** is as above, except that decryption in Node.Execute with key $k_w$ rejects any ciphertext not produced by AEAD encryption in User.Complete with key $k_w$. For each key $k_w$, this differs from Game 4 only on decryptions of forged ciphertexts. By definition of INT-CTXT, this happens at most with probability $\epsilon^{\mathsf{AEAD}}_{\mathsf{INT\text{-}CTXT}}$. Since there are at most $\beta_1$ such keys, we get

$$p_4 \leq p_5 + \beta_1 \cdot \epsilon^{\mathsf{AEAD}}_{\mathsf{INT\text{-}CTXT}}$$

**Game 6 (IND-CPA with $k_w$)** is as above, except that for each user session, we maintain a table $T$ that maps ciphertexts under key $k_w$ to plaintexts $k_{code} \mid \mathbf{k}$; (1) User.Complete is modified to encrypt dummy credentials instead of $k_{code} \mid \mathbf{k}$ in any $JC_w$ message and to record in $T$ the resulting ciphertext and $k_{code} \mid \mathbf{k}$; and (2) Node.Execute is modified to first attempt to retrieve $k_{code} \mid \mathbf{k}$ from $T$ before decrypting with $k_w$.

Both in Games 5 and 6, any ciphertext not in the table is rejected, hence decryption never occurs in Game 6, and we can apply our IND-CPA assumption for AEAD to each key $k_w$. Let $\epsilon^{\mathsf{AEAD}}_{\mathsf{IND\text{-}CPA}}(n)$ be the advantage of an IND-CPA adversary that performs $n$ oracle encryptions. We arrive at

$$p_5 \leq p_6 + \beta_1 \cdot \epsilon^{\mathsf{AEAD}}_{\mathsf{IND\text{-}CPA}}(\alpha_2 \cdot N_{max})$$

From this game, $k_{code}$ and the keys $k_{job}$, $k_{in}$, $k_{inter}$, and $k_{out}$ are fresh, random values used only as AEAD keys, so we can also apply IND-CPA and INT-CTXT for them; and $k_{prf}$ is a fresh random value only used for keying PRF.

**Game 7 (IND-CPA with $k_{code}$)** is as above, except that (similarly to Game 6), for each user session, we maintain a table $T'$ that maps ciphertexts under key $k_{code}$ to code $E^-$; (1) User.Init oracle is modified to encrypt a dummy data blob of equal length instead of $E^-$, and records the resulting ciphertext and $E^-$ in $T'$; and (2) Node.Execute is modified to first attempt to retrieve $E^-$ by a lookup from $T'$, before actually decrypting with $k_{code}$. Since $k_{code}$ is specific to $(j, pk_u)$ and the node necessarily runs the correct $C_{j,u}$, in Game 7, $k_{code}$ is never actually used for decryption as all look ups from $T'$ succeed. Hence, we can apply our IND-CPA assumption for AEAD for each session's $k_{code}$ and obtain

$$p_6 \leq p_7 + \alpha_1 \cdot \epsilon^{\mathsf{AEAD}}_{\mathsf{IND\text{-}CPA}}(1)$$

From this final game, we show that both properties of Theorem 1 perfectly hold, that is, $p_7 = 0$:

**Agreement** Valid $JC_w$ messages can only ever be obtained as a user identified by $pk_u$ completes with $j$, $N$, $E^+, E^-$, $\mathbf{k}$ as a result of a call to User.Complete. When a user completes, User.Complete creates one message $JC_w$ for exactly $N$ distinct $k_w$. (The adversary may intercept these messages, but can never obtain others for the session indexed by $(pk_u, j)$.) The valid quote for each $p_w$ message guarantees that it originates from Node.Execute and is unambiguously tied to $C_{j,u}$.

Ciphertext integrity and the correctness of AEAD decryption yields a similar guarantee whenever Node.Execute accepts a message $JC_w$ for node $k_w$ and decrypts $k_{code} \mid \mathbf{k}$: it must be running $C_{j,u}$, which also authenticates the user session index $(pk_u, j)$ and $E^+$. Given the correct $C_{j,u}$, Node.Execute necessarily also obtains the correct $E^-$ for the node's completion.

**Privacy** The privacy property of Theorem 1 holds information-theoretically, since none of the data exchanged with the adversary depends on the value of $b$, and thus $p_7 = 0$.

Collecting the probabilities from all games yields:

$$\begin{aligned} p_0 \leq\ & \frac{\alpha_1^2}{2^\kappa} + \frac{\beta_0^2}{2^\lambda} + \epsilon^{\mathsf{PKGen}}_{collision}(\alpha_0) + \epsilon_{\text{UF-CMA}}(\gamma) \\ & + \alpha_0 \cdot \epsilon^{\mathsf{AEAD}}_{\mathsf{IND\text{-}CPA}}(\beta_1) + \beta_0 \cdot \epsilon^{\mathsf{PRF}}_{distinguish}(\lambda, \beta_1) \\ & + \alpha_1 \cdot \epsilon^{\mathsf{AEAD}}_{\mathsf{IND\text{-}CPA}}(1) \\ & + \beta_1 \cdot (\epsilon^{\mathsf{AEAD}}_{\mathsf{INT\text{-}CTXT}} + \epsilon^{\mathsf{AEAD}}_{\mathsf{IND\text{-}CPA}}(\alpha_2 \cdot N_{max})) \end{aligned}$$

More abstractly, every factor $\alpha_0$, $\alpha_1$, $\alpha_2$, $\beta_0$, $\beta_1$, and $\gamma$ is bounded by the total number of oracle calls made by the adversary, so $p_0$ becomes negligible for large security parameters $\lambda$ and $\kappa$ under the given assumptions. □

### 4.8.3 Job Integrity and Privacy

Since the key exchange yields agreement on code, a fresh job ID, and keys between the user and a set of nodes $\mathbf{w}$, it is sufficient to study the security of *VC3* for every job execution in isolation.

**Distributed MapReduce** We first introduce additional notations for specifying this MapReduce job as two functions. We represent splits of input, intermediate, and output files as multisets of key-value pairs $KV$.

- We let $Map(\_)$ be the function from input- to intermediate- key-value pairs, and let $Map(R, r, \_)$ with $r \in 0..R-1$ be the functions that yield the intermediate key-value pairs mapped to $r$ out of $R$ reducers.

  Seen as functions from multisets to multisets, these functions distribute over multiset unions, yield key-value pairs with distinct keys for each value of $r$, and are such that

  $$Map(\_) = \biguplus_{r \in 0..R-1} Map(R, r, \_).$$

  where $\uplus$ denotes multiset disjoint union.

- We let *Reduce*(_) be the function from intermediate- to output- key-value pairs. This function distributes over multiset unions *with distinct keys*. (This reflects the constraint that all pairs with the same key should be reduced together.)

Hence, we specify the whole MapReduce job as

$$Output = Reduce(Map(Input)).$$

We now describe the intended distributed execution of the job given any number of mappers, $R$ reducers, and an input file that consists of $I$ splits, $Input = \bigcup_{i=0..I-1} Input_i$. Each reducer $r \in 0..R-1$ should compute its *output split*

$$Output_r = Reduce(\bigcup_{i \in 0..I-1} Map(R, r, Input_i))$$

(where $\cup$ denotes multiset union) and, relying on the distributive properties of *Map* and *Reduce*, we should have $Output = \bigcup_{r \in 0..R-1} Output_r$. Informally, our protocol should also ensure that each mapper $w \in \mathbf{m}, \mathbf{m} \subseteq \mathbf{w}$ processes input splits $(Input_i)_{i \in \pi_w}$ such that $(\pi_w)_{w \in \mathbf{m}}$ is a partition of $0..I-1$.

(Note that, for flexibility, the same node may implement a mapper and some of the reducers. In our model, this allocation choice and the total number of mappers is up to the adversary. In contrast, the number of reducers $R$ is fixed in the job code.)

**The Job Integrity Game** The job integrity game starts with the state at the end of the successful completion of the key exchange protocol. It models interactions between a probabilistic polynomial-time adversary $\mathcal{A}$ that includes the network and the Hadoop framework and that controls the trusted roles of the job protocol: the nodes (used as mappers and/or reducers) and the verifier. In particular, the adversary has already provided (valid *VC3* implementations of) the functions *Map* and *Reduce*, as well as the number of reducers $R$; and the nodes, the user, and the trusted verifier already share fresh distinct keys $\mathbf{k}$. The game runs as follows:

1. The adversary provides the input file, as a sequence of plaintext splits $(Input_i)_{i \in 0..I-1}$ for some $I \geq 0$.

2. The game (modeling the user) checks that their format is correct, then, as detailed in Section 4.6, Step 1,

   - it samples a split ID $\ell_{in,i} \xleftarrow{\$} \{0,1\}^\kappa$ for each $i \in 0..I-1$;
   - it records $I$, $R$, and $(\ell_{in,i})_{i \in 0..I-1}$ within the job specification $S_{job}$; and
   - it gives to the adversary the encryptions

   $$(Input'_i)_{i \in 0..I-1} = (\mathsf{Enc}_{k_{in}}[\ell_{in,i}]\{Input_i\})_{i \in 0..I-1}.$$

3. The adversary interacts with the nodes $w \in \mathbf{w}$ allocated for the job, by calling the oracles defined below, any (polynomially-bounded) number of times, in any order.

   The adversary finally returns some evidence $s^\star$ that the job completed successfully.

4. The game (modeling the verifier) parses $s^\star$ as
   - final mapper messages $FM$ authenticated using $k_{job}$;
   - final reducer messages $FR$ authenticated using $k_{job}$;
   - encrypted output key-value pairs that decrypt to outputs splits $(Output_i)_{i\in 0..O-1}$ with pairwise-distinct IDs $(\ell_{out,i})_{i\in 0..O-1}$ using key $k_{out}$.

   The game checks that the final messages are complete with regard to the recorded job specification $I$, $R$, and $(\ell_{in,i})_{i\in 0..I-1}$ and the output IDs $(\ell_{out,i})_{i\in 0..O-1}$, as defined in Section 4.6, Step 4.

   If all these tasks succeed, the game completes with the resulting output file:

$$\cup_{i\in 0..O-1} Output_i$$

Next, we define the oracles for Step 3, parameterized by the abstract functions *Map* and *Reduce* encoded in $E^+$ and $E^-$ and the number of reducers for the job, $R$. These oracles maintain private local state; their input/output behavior follows the definitions of Section 4.6, Step 2 (mapping) and Step 3 (reducing). The messages $s$ and $s'$ range over binary inputs and outputs exchanged with the adversary (modeling the untrusted Hadoop infrastructure).

- $s' \leftarrow \mathsf{Node.Map}(w, s)$ models Hadoop calls to the protected mapper function.

  If no mapper state exists for $w$, the oracle samples $\ell_w \xleftarrow{\$} \{0,1\}^\kappa$, records it as $w$'s mapper ID, and initializes sequence numbers $i_{w,r} = 0$ for each reducer $r \in 0..R-1$.

  Then, if $s$ is an input split, the oracle checks that

  1. mapping for $w$ is not recorded as complete;
  2. message $s$ AEAD-decrypts using $k_{in}$ into an input split $Input_i$ with ID $\ell_{in,i}$; and
  3. $\ell_{in,i}$ has not already been mapped by $w$. (The $i$ index is for reference in the proof—the mapper does not know $i$, only $\ell_{in,i}$.)

  The oracle records that $w$ has mapped $\ell_{in,i}$; for each $r \in 0..R-1$, it computes $Map(R, r, Input_i)$ and it builds the resulting intermediate key-value pairs $KV'_{inter}$ using AEAD with $k_{inter}$ and authenticated data including $\ell_w$, $r$, and the current sequence number $i_{w,r}$ from $w$ to $r$. It increments and records $i_{w,r}$, and returns the concatenation of all these messages as $s'$.

  Otherwise, if $s$ is empty (modelling the end of input), the oracle creates $R$ closing key-value pairs $KV_{close}$ that authenticate $\ell_w$ and the final value of $i_{w,r}$ using key $k_{inter}$. It also creates the final mapper verification message $FM$ that contains the set of labels $\ell_{in,i}$ mapped by $w$. It records mapping for $w$ as complete, and returns the concatenation of all these messages as $s'$.

- $\mathsf{Node.Reduce}(w, r, s)$ models Hadoop calls to the protected reduce function.

  It checks that $r \in 0..R-1$ and that reducing for $(w, r)$ is not recorded as complete.

Then, if $s$ is a message that decrypts using $k_{inter}$ to a $KV_{inter}$ key-value pair from mapper ID $\ell$ to reducer $r$ with the expected sequence number $i_{\ell,w,r}$. (The oracle maintains a sequence number for each received $\ell$ and each $w, r$, starting with $i_{\ell,w,r} = 0$ and incremented after each successful decryption authenticating $\ell$ and $r$.) The oracle records the decrypted $KV_{inter}$ for $(w, r)$ and returns.

Otherwise, if $s$ is a series of messages $KV_{close}$ that authenticate using $k_{inter}$ pairwise-distinct mapper IDs $\ell$ with $r$ and their expected final sequence numbers $i_{\ell,w,r}$ (including at least one message for each non-zero $i_{\ell,w,r}$) then the oracle finally invokes *Reduce* on all $KV_{inter}$ recorded for $(w, r)$ and, for each resulting output split, produces an output key-value pair $KV'_{out}$ using key $k_{out}$ and a fresh ID $\ell_{out} \xleftarrow{\$} \{0,1\}^\kappa$.

The oracle also creates a final message $FR$ that carries all these IDs $\ell_{out}$ and the sorted list $P_r$ of all the mapper IDs $\ell$ received for $(w, r)$ (see $KV'_{out}$ and $FR$ in Section 4.6, Step 3).

It records reducing for $(w, r)$ as complete, and returns the concatenation of all these messages.

(For simplicity, our reducers produce output only after receiving all their intermediate key-value pairs; this is not essential.)

In all other cases, the oracles return an empty result.

**The Job Confidentiality Game** For confidentiality, we adapt the game above as follows.

- The adversary initially provides *pairs* of functions $Map^b$, $Reduce^b$ leading to private code $E^{b,-}$ of the same size (possibly after padding) and plaintext input splits $(Input_i^b)_{i \in 0..I-1}$, for $b = 0, 1$.
- The game selects $b$ at random, performs all key-value pair computations for both $b = 0$ and $b = 1$, but consistently uses the values indexed by $b$ as it interacts with the adversary.
- At each step, the game also checks that the shape of the encrypted messages it returns to the adversary does not depend on $b$:
  - each of the AEAD encryptions in Step 1 of the game must have plaintext encodings of $Input_i^0$ and $Input_i^1$ with the same size (possibly after padding);
  - for each mapper and each $r$, the two sequences of intermediate key-value pairs must have the same *key repetitions* and the same plaintext sizes (possibly after batching and padding);
  - each reducer must issue the same number of output splits $Output_i^0$ and $Output_i^1$, with the same plaintext size (possibly after padding).

  Otherwise, the game stops at the first discrepancy.

  As discussed in Section 4.1.1, we leave traffic analysis outside of our attacker model. Accordingly, the mechanism above simply excludes any adversary that may distinguish between the two jobs by traffic analysis.

- Instead of Step 4 (checking the job integrity and re-assembling the job output), the adversary provides $g$, and the game returns $b = g$.

**Theorem 2** (Job Execution)**.** *Consider the two games above, following a safe complete key exchange, against a probabilistic polynomial-time adversary $\mathcal{A}$.*

- ***Job Integrity:*** *Except for a negligible probability, if the game completes and returns Output, then we have*

$$Output = Reduce(Map(Input))).$$

- ***Privacy:*** *The advantage of the adversary (that is, the probability that $b = g$ minus $\frac{1}{2}$) is negligible.*

*Proof.* Building on the key exchange protocol (§4.5.2), we assume that the nodes for the job and the verifier share freshly-generated random keys $\mathbf{k}$.

We use a series of game transformations, writing $p_g$ for the probability (or the advantage) proved negligible in the theorem.

**Game 0** is the one given in the theorem statement.

**Game 1 (Collisions)** is as above, except that we exclude collisions (i) between two node IDs $\ell_w$, (ii) between two input split IDs $\ell_{in,i}$, or (iii) between two output split IDs $\ell_{out,i}$. By reasoning about the probabilities with which these events occur, we get

$$p_0 \leq p_1 + \frac{\sigma^2}{2^\kappa}$$

where $\sigma$ and $\kappa$ are the total number and the size (in bits) of IDs created during the game.

**Game 2 (INT-CTXT with each key of $\mathbf{k}$)** is as above, except that—analogously to Game 5 in Section 4.8.2—all AEAD decryptions performed by the game and its oracles succeed only for authentic messages (composed of ciphertexts and additional authenticated data). Since we use 4 keys, we get

$$p_1 \leq p_2 + 4 \cdot \epsilon^{\mathsf{AEAD}}_{\mathsf{INT\text{-}CTXT}}$$

**Game 3 (IND-CPA with keys $k_{in}$, $k_{inter}$, and $k_{out}$)** is as above, except that plaintexts are replaced with fixed dummy values of the same length in AEAD encryptions with $k_{in}$, $k_{inter}$, or $k_{out}$. As in Games 6 and 7 in Section 4.8.2, a separate table is maintained for each of these keys that maps corresponding ciphertexts to plaintexts. For each AEAD decryption with $k_{in}$, $k_{inter}$, or $k_{out}$, the new game first attempts to retrieve the plaintext from the corresponding table. With the modification from Game 2 in place, any ciphertext not contained in the corresponding key's table is automatically rejected, hence AEAD

decryption never occurs in Game 3 and we can apply our IND-CPA assumption for $k_{in}$, $k_{inter}$, and $k_{out}$. (The key $k_{job}$ is used only for authentication.)

$$p_2 \leq p_3 + \epsilon^{\mathsf{AEAD}}_{\mathsf{IND\text{-}CPA}}(I_{max}) + \epsilon^{\mathsf{AEAD}}_{\mathsf{IND\text{-}CPA}}(\rho) + \epsilon^{\mathsf{AEAD}}_{\mathsf{IND\text{-}CPA}}(O_{max})$$

by summing up the probabilities of breaking IND-CPA for $k_{in}$, $k_{inter}$, and $k_{out}$ and with $I_{max}$ being the maximum number of input splits, $\rho$ being the maximum number of intermediate key-value pairs, and $O_{max}$ being the maximum number of output splits processed in the game.

**Game 4 (PRF with $k_{prf}$, only required for confidentiality)** is as above, except that we replace the PRF used for hiding the keys of the intermediate key-value pairs with a perfect random function with lazy sampling (that is, a function that maintains a table from input to outputs, and samples a new output when given a new input). By definition of PRF, we have

$$p_4 \leq p_3 + \epsilon^{\mathsf{PRF}}_{distinguish}(\theta, \rho)$$

(Recall that $\theta$ is the length of $k_{prf}$.) We now show that the properties of the theorem hold perfectly in Game 4 ($p_4 = 0$) then conclude that they hold with overwhelming probability in Game 0.

**Job Invariant** Let $\pi_w \subseteq 0..I-1$ record the indexes of input splits that have been mapped by node $w$ (these sets are not necessarily a partition of the input IDs), let $X_{w,r} = \cup_{i \in \pi_w} Map(R, r, Input_i)$, and let $\mathbf{m} \subseteq \mathbf{w}$ be the set of nodes that Node.Map was invoked on. By induction on the number of calls to the oracles in Game 3, we show the following invariants:

1. (Map:) For each node $w \in \mathbf{m}$ and each $r \in 0..R-1$, the logged AEAD messages $KV'_{inter}$ using $k_{inter}$ with authenticated data $\ell_w$, $r$ and $i_{w,r}$ carry a sequence of intermediate key-value pairs (indexed by $i_{w,r}$) that represent $X_{w,r}$.

2. (Map Completion:) For each $w \in \mathbf{m}$, there is at most
   a) for each $r \in 0..R-1$, one logged message $KV_{close}$ using $k_{inter}$ that authenticates the final sequence number $i_{w,r}$ in the representation of $X_{w,r}$ above; and
   b) one logged message $FM$ using $k_{job}$ that authenticates $\ell_w$ and the final set of IDs $(\ell_{in,i})_{i \in \pi_w}$ for the input splits mapped by $w$.

3. (Reduce:) For each $w \in \mathbf{m}$ , $w' \in \mathbf{w}$, and $r \in 0..R-1$, the intermediate key-value pairs received with mapper ID $\ell_w$ are included in $X_{w,r}$ (up to the current index $i_{\ell_w,w',r}$).

4. (Map-Reduce Completion:) Moreover, once the $(w', r)$-reducer accepts a $KV_{close}$ with mapper ID $\ell_w$, the node $w$ has completed mapping, and they agree on the intermediate key-value pairs $X_{w,r}$.

5. (Reduce Completion:) For each $w' \in \mathbf{w}$ and $r \in 0..R-1$, there exists at most one logged message $FR$ using $k_{job}$ each authenticating $r$ as well as
   a) the final set $P_{w',r} \subseteq (\ell_w)_{w \in \mathbf{m}}$ of mapper IDs received by the $(w', r)$-reducer; and
   b) the final set $(\ell_{out,i})_{i \in \pi_{w',r}}$ of IDs for the output split produced by reducing their intermediate key-value pairs $X_{w,r}$.

**Integrity** Given these invariants, we reason about the verification steps:

- Invariant 4 ensures that reducers receive all intermediate key-value pairs from the mappers they know of—otherwise, a reducer's local verification fails before sending $FR$. (Conversely, it does not exclude multiple reducer nodes for the same $r$, possibly with different subsets of received mapper IDs.)
- The verifier knows $R$ from the job description $S_{job}$. The verification of the $R$ messages ensures that there is exactly one for each $r \in 0..R-1$ and that they all agree on the set of mapper IDs.
- The verification of the $FM$ messages ensures that they are from pairwise-distinct nodes, with mapper IDs that agree with those received by the reducers.
- Thus, the verifier knows that each reducer necessarily communicated with exactly that set of mappers with distinct IDs for which it received verification messages $FM$. We write $\mathbf{m'} \subseteq \mathbf{m} \subseteq \mathbf{w}$ for this set of mappers.
- Finally, given all messages $FM$ for $w \in \mathbf{m'}$, the verifier collects the combined multiset of IDs $M = \biguplus_{w \in \mathbf{m'}} \{\ell_{in,i}, i \in \pi_w\}$ of the input splits that have been effectively mapped, then reduced to *Output*.

  By comparing $M$ with the input specification $\{\ell_{in,i}, i \in 0..I-1\}$ in $S_{job}$, the verifier ensures that each input split contributed exactly once to *Output*.

Finally, when the verifier accepts the evidence presented by the adversary, the sets of IDs $\pi_w$ form a partition of the input file and, for each $r$, the reducer instance endorsed by the verifier has received and processed exactly the key-value pairs $\bigcup_{i \in 0..I-1} Map(R, r, Input_i))$.

By definition of the Node.Reduce oracle, each endorsed reducer instance has produced encrypted output splits for

$$Reduce(\bigcup_{i \in 0..I-1} Map(R, r, Input_i))$$

using disjoint sets of IDs $\{\ell_{out,i}, i \in \pi_r\}$, where $\{\pi_r, r \in 0..R-1\}$ is a partition of $0..O-1$. Thus, the AEAD ciphertexts using key $k_{out}$ accepted in the final step of the game to re-assemble *Output* are exactly the splits of the output file for the set of IDs $\biguplus_{r \in 0..R-1} \{\ell_{out,i}, i \in \pi_r\}$ accepted by the verifier. By definition, we obtain $p_3 = 0$.

**Confidentiality** In Game 4, we rely on our traffic-analysis assumption, which guarantees that the encryption sizes, numbers of splits, and repetitions of outer keys do not depend on $b$. We show that, at every step of the confidentiality game, the values given to the adversary also do not depend on $b$.

For the second step (encoding the input), this holds by assumption on the input splits provided by the adversary, the independent sampling of IDs, and the fact that (after applying IND-CPA) we are encrypting the same dummy values, irrespective of $b$.

For the third step (interacting with the Map and Reduce oracles), the same reasoning applies for all encrypted intermediate key-value pairs and encrypted output splits. Similarly, the verification messages only depend on IDs and numbers of splits and key-value pairs, not their contents.

This leaves the sequence of values of the 'outer' keys for the intermediate key-value pairs sent from each mapper $w$ to each reducer index $r$. By traffic-analysis assumption, and induction on the state of the random function, we show that these sequences of 'outer' keys do not depend on $b$: since the actual keys have the same repetitions, either both sides use their actual keys for the first time, thereby causing the random function to sample and record a fresh outer key—the two actual keys may be different, but the sampling does not depend on them—or both sides re-use actual keys that first occurred at the same time, and the random function returns the same outer key by table lookup.

We conclude that the final guess $g$ of the adversary does not depend on $b$, hence that its advantage is $p_4 = 0$.

**Concrete Security** Collecting the probabilities from the game sequence yields

$$p_0 \leq \frac{\sigma^2}{2^\kappa} + 4 \cdot \epsilon^{\mathsf{AEAD}}_{\mathsf{INT\text{-}CTXT}} + \epsilon^{\mathsf{PRF}}_{distinguish}(\theta, \rho) + \epsilon^{\mathsf{AEAD}}_{\mathsf{IND\text{-}CPA}}(I_{max}) + \epsilon^{\mathsf{AEAD}}_{\mathsf{IND\text{-}CPA}}(\rho) + \epsilon^{\mathsf{AEAD}}_{\mathsf{IND\text{-}CPA}}(O_{max})$$

where $\sigma$ is bounded by the total number of calls to Node.Map made by the adversary and $I_{max}$, $O_{max}$, and $\rho$ are job-specific values that are bounded by the total number of messages processed and produced by the oracles. Accordingly, $p_0$ becomes negligible for a large security parameters $\kappa$ and $\theta$ under the given assumptions. Theorem 2 follows. □

## 4.9 Implementation

We implemented the *VC3* framework, namely $F$ and $E^-$, in C, C++, and x86-64 assembly for Windows 8 64-bit / Windows Server 2012 64-bit. While the user mode (*fw.exe*) and kernel mode parts (*fw.sys*) of $F$ are specific to Windows, our $E^+$ code is operating system agnostic. We created a toolchain of C++ programs for the user that allows for the automation of the described approach: *keygen.exe* generates a user's symmetric and asymmetric keys. *packer.exe* statically resolves dependencies between $E^-$ and $E^+$, encrypts $E^-$, and merges both into a self-contained and signed mapred.dll which constitutes $C_{j,u}$. The packer also makes sure that the sections in mapred.dll (e. g., `.text` and `.data`) are page-aligned, as they would be when loaded into a user mode process by the standard Windows *image loader* [171]. It is necessary to make sure that the enclave code can be

loaded into memory and run unaltered without the help of the standard image loader, because users need to be able to reliably compute an enclave's measurement in advance. Otherwise, they cannot reasonably verify statements by QEs. The toolchain is incorporated into a Visual Studio template that automatically creates mapred.dll from a user's implementation of $E^-$.

Jobs are deployed in the form of fw.exe and mapred.dll. Fw.sys is expected to be already installed on all nodes. Fw.sys becomes obsolete as soon as SGX support is incorporated in mainstream operating systems. Fw.exe performs I/O interaction with Hadoop via a simple protocol [17] over stdin/stdout.

$E^+$ contains a custom heap allocator that serves requests originating from `malloc()` and `new`. Per default memory is allocated inside the enclave. It is possible to explicitly allocate memory outside the enclave in the shared memory area in order to communicate with the outside world. All heap metadata is stored out-of-band inside the enclave and cannot be corrupted from the outside.

As transitions in and out of the enclave come at a cost, we avoid them where possible by batching read/writes of key-value pairs from within the enclave. We also implemented an experimental mode where the enclave is never left during the execution of a job and all I/O operations are performed asynchronously over the shared memory region. While this minimizes the overhead that stems entering/exiting the enclave, it also requires a second thread/core to wait for commands from the enclave.

Our implementation of $E^+$ consists of roughly 5500 *logical lines of code* (LLOC) of C, C++ and x86-64 assembly. About 2500 LLOC of these implement standard cryptographic algorithms. The user can inspect, change and recompile the code of $E^+$, or even use our protocol specification to completely re-implement it.

**In-enclave Library** As a convenience for application development, we have created an enclave-compatible C++ runtime library. Existing C/C++ libraries which have operating system dependencies cannot be used in an enclave environment, because system calls are conceptually not available [109]. Accordingly, we could neither use common implementations of the *Standard C Library* nor of the *C++ Standard Template Library*. Our library contains functions which we found useful when writing our own applications: a set of mathematical functions, string classes, containers, and a heap allocator which manages the in-enclave heap and is the default backend for `new`. This library is relatively small (3702 LLOC) and we stress that users may choose to change it, use other libraries instead, or write their own libraries.

## 4.10 Evaluation

We used the applications listed in Table 4.1 to evaluate *VC3*. We chose a mix of real-world applications and well-known benchmarks, including IO-intensive and processor-intensive applications. We measured the performance of the applications on Hadoop, and also in isolation to remove the overhead-masking effects of disk I/O, network transfers, and spawning of Hadoop tasks. Before discussing our results, we briefly describe each application.

| Application | LLOC | Size input | Size $E^-$ (vc3) | Map tasks |
|---|---|---|---|---|
| UserUsage | 224 | 41 GB | 18 KB | 665 |
| IoVolumes | 241 | 94 GB | 16 KB | 1530 |
| Options | 6098 | 1.4 MB | 42 KB | 96 |
| WordCount | 103 | 10 GB | 18 KB | 162 |
| Pi | 88 | 8.8 MB | 15 KB | 16 |
| Revenue | 96 | 70 GB | 16 KB | 256 |
| KeySearch | 125 | 1.4 MB | 12 KB | 96 |

Table 4.1: Applications used to evaluate *VC3*

**UserUsage and IoVolumes** The resource usage information from a large compute/storage system, consisting of tens of thousands of servers, is processed. Users issue tasks to the system, which spawn processes on multiple servers. The measured resource usage information is written to two event logs: a process log with one row per executed process, and an activity log that records fine-grained resource consumption information.

Two real applications to process this data were ported to *VC3* [176]: *UserUsage* counts the total process execution time per user. *IoVolumes* is a join: it filters out failed tasks by reading the process log and then computes storage I/O statistics for the successful tasks from the activity log.

**Options** The prices of European call options are simulated using *Monte Carlo* methods [140]. Mappers perform the actual simulation, whereas a single reducer aggregates the results. The large size of the application in terms of LLOC (see Table 4.1) stems from the inclusion of a set of optimized mathematical functions.

**WordCount** The occurrences of words in the input are counted. Mappers parse an input split into individual words and output an intermediate key-value pair of the form *<[word]:1>* for each word. Reducers sum the counts for each word. An excerpt from the source code was already given in Listing 4.1 on page 101.

**Pi** The value of Pi is statistically estimated [6]. Mappers generate random points inside a unit square, and count the number of points falling inside a circle inscribed within that square. A single reducer collects the counts from all mappers and computes the fraction of points that fall inside the circle approximating $\pi/4$ thereby.

**Revenue** The synthetic log files of websites are processed and the total ad revenue per visitor IP is accumulated (adapted from Pavlo et al. [155]). Mappers process individual log files. Reducers aggregate overall results per IP.

**KeySearch** A known plaintext attack on a 16-byte message encrypted with RC4 is conducted [213]. The key space is divided into blocks. Mappers search iteratively through their assigned key space blocks, whereas a single reducer simply forwards the key once it was found by a mapper.

[6] `http://hadoop.sourcearchive.com/documentation/0.20.2plus-pdfsg1-1/PiEstimator_8java-source.html`

### 4.10.1 Experimental Setup

Our *VC3* code has been successfully tested in a hardware-based SGX emulator provided by Intel [176]. While this emulator precisely implements many functional aspects of SGX [109], it is not performance accurate. Hence, we used a software emulator for SGX implemented as Windows kernel driver [176] to assess the expected performance of *VC3*. Most significantly for our experiments, this software emulator applies a penalty of one TLB flush and 1,000 delay cycles for every control-flow transition from or to an enclave. This includes enclave transitions due to interrupts as well as explicit ones.

All experiments ran under Microsoft Windows Server 2012 R2 64-bit on workstations with a 2.9 GHz Intel Core i5-4570 (Haswell) processor, 8 GB of RAM, and a 250 GB Samsung 840 Evo SSD. For distributed experiments, a cluster of eight workstations connected with a Netgear GS108 1Gbps switch was used. The code of each of the seven applications in Table 4.1 was compiled with the Microsoft C++ compiler version 18.00.30501 for x86-64, optimizing for speed in two configurations:

`baseline` runs the applications on plaintext data and without following the job execution protocol. Also, no performance penalty for enclave transitions (TLB flush, delay cycles, and swapping of the stack) is applied and unnecessary copying of data across (non-existent) enclave boundaries is avoided.

`vc3` runs the same application on *VC3* with encrypted mapper and reducer inputs and outputs in the described SGX software emulator. Sizes of the $E^-$ DLL range from 12 KB for KeySearch to 42 KB for Options (see Table 4.1); the generic $E^+$ DLL has a size of 210 KB. The enclave memory size was set to be 512 MB. This version provides the security guarantees of *VC3*.

### 4.10.2 Performance on Hadoop

We measured the execution times of `baseline` and `vc3` in an unmodified Hadoop environment. We used the Hortonworks distribution of Hadoop 2 (HDP 2.1) for Windows with eight worker nodes (one per workstation). We used the default configuration options for resource management, and configured our jobs to use eight reduce tasks; except for Pi, Options, and KeySearch that conceptually use one. We ran each job and each configuration at least ten times and measured the execution time. To facilitate comparisons, we normalized the running times with the average running time for each job using the `baseline` configuration. Figure 4.5 plots the average ratios for each job and configuration, and the values of two standard deviations below and above each average.

Figure 4.5 shows that `vc3`'s performance overhead is negligible as the differences in performance are well below the experimental variability for all jobs.

### 4.10.3 Performance in Isolation

When running applications, Hadoop performs many activities, such as spawning mappers and reducers, waiting for disk I/O, network transfers, and others, that may mask the overheads of *VC3*. To better understand the performance impact of *VC3* on the execution times of individual map and reduce tasks, we ran the mappers and reducers in isolation, i. e., from the command line on a single machine without Hadoop. We repeated each

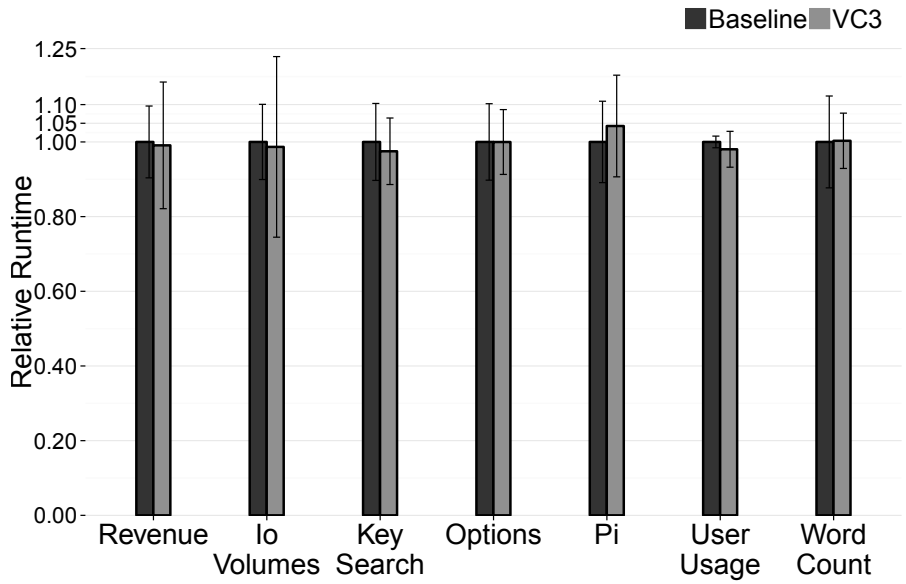


Figure 4.5: Execution time of running MapReduce jobs in a Hadoop cluster over typical input data-sets; running times are normalized to the performance of running the same job in normal mode and with unencrypted data (`baseline`).

experiment ten times as in Section 4.10.2. Figure 4.6 plots the average ratios for the map tasks, as well as the values of two standard deviations below and above the average. The results for reduce tasks are similar. They are omitted for brevity.

Here, `vc3`'s average overhead was 4.3% compared to `baseline`. The overheads were negligible for the three compute intensive jobs KeySearch, Options, and Pi. These jobs spend little time in copying and encryption/decryption operations, and most of the time they compute using plain-text data off of the processor's caches. Likewise, barely a performance difference between `baseline` and `vc3` can be observed for Revenue and WordCount, which have balanced I/O and compute demands.

In contrast, the I/O-intensive jobs IoVolumes and UserUsage exhibited a performance overhead of 23.1% and 6.1% respectively for `vc3`. As these jobs perform little computation, the relative cost of encryption is higher. While Revenue and WordCount also have a relatively high I/O throughput, these applications implement a combine operation which increases the computation performed at the mapper, hence reducing the relative cost of encryption. The combine operation performs a group-by of the key-value pairs generated by the map function, and calls a combine function that performs a partial reduction at the mapper.

Overall, we observe that the overhead of `vc3` should be well within a practical range for most applications.

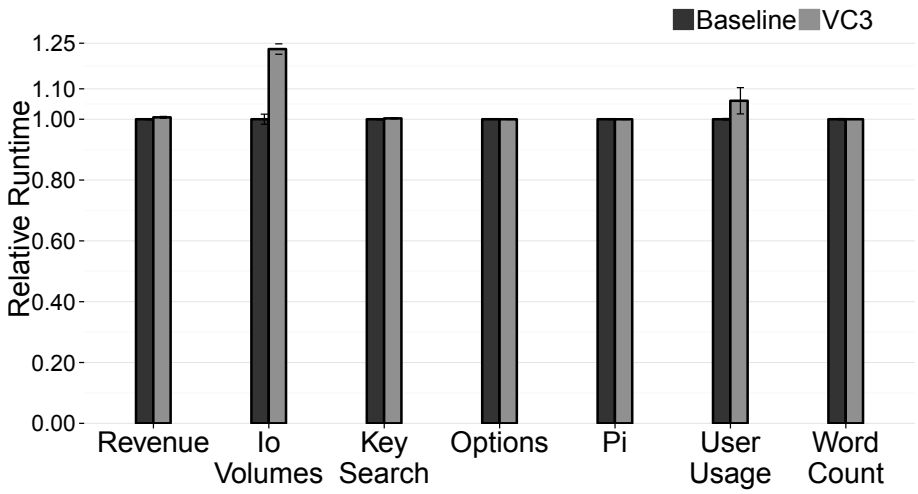


Figure 4.6: Execution time of running the map phase of MapReduce jobs in isolation over typical input data-sets; running times are normalized to the performance of running the same computation in the `baseline` configuration.

## 4.11 Further Applications

We now discuss potential future applications for *VC3* that we find interesting.

### 4.11.1 P2P MapReduce

*VC3* could possibly be leveraged for the secure creation of distributed cloud infrastructures similar to conventional P2P networks. Private owners of SGX-enabled PCs could rent out unused computing capacities. This would though increase the risk of sophisticated physical attacks on processors as practically no trusted party could guarantee for their physical integrity (see discussion in Section 4.5.1). However, for certain types of computations, the inherent hardware security of SGX could very well proof sufficient. For example, in the case of *volunteer computing* [14] *VC3* could not only improve overall security but also reduce the overhead of existing integrity protection schemes that often rely on performing redundant computations on different participating nodes [13, 73, 207].

### 4.11.2 Single-Run MapReduce Job Licensing

*VC3* enables new fine-grained software licensing models for the cloud. We specifically envision a single-run licensing of MapReduce jobs that offers strong security properties and flexibility for both user and software vendor. In essence, a software vendor would provide private enclave code $E^-$ that only accepts input splits that the user paid for. The user could choose the actual cloud provider independently. Of course, the user would need to fully trust the vendor's private code. The following outlines a corresponding protocol:

1. The user buys a single-run MapReduce job license from a software vendor and chooses an arbitrary cloud provider for executing the job.

2. The user communicates the job ID $j$ and the IDs of all input splits for the run (i. e., $B_{Input}$) to the vendor.

3. The vendor prepares an extended enclave code package

$$C_{j,u,v} = C_{j,u} \mid B_{Input} \mid pk_v$$

where $pk_v$ is the vendor's public key and $C_{j,u}$ is the basic enclave code package defined in Section 4.5.2. The vendor ensures that the enclave code rejects input splits whose IDs are not contained in $B_{Input}$.

4. The user inspects the public enclave code $E^+$; if $E^+$ is as expected, the user uploads $C_{j,u,v}$ to the cloud provider and initiates the key exchange as described in Section 4.5.2.

5. On each node $w$, inside the enclave, $E^+$ derives a symmetric key $k_{w,u}$ for the user and a symmetric key $k_{w,v}$ for the vendor from its $k_w$:

$$k_{w,u} = \mathsf{PRF}_{k_w}(0)$$
$$k_{w,v} = \mathsf{PRF}_{k_w}(1)$$

6. Each enclave securely communicates $k_{w,u}$ to the user and $k_{w,v}$ to the vendor analogously to step 2 in Section 4.5.2. (Note how here, $k_w$ never leaves the enclave and remains secret to the enclave code.)

7. Analogously to step 3 in Section 4.5.2, the user uses $k_{w,u}$ to securely communicate $\mathbf{k}$ and the vendor uses $k_{w,v}$ to securely communicate $k_{code}$ back to the enclave.

8. Finally, step 4 from Section 4.5.2 can be executed.

This extended key exchange can be implemented *in-band* as described in Section 4.5.2.1.

The actual job can be executed as normal (see Section 4.6). The job only produces outputs for a subset of the input splits whose hashes the user communicated to the vendor. Of course, the user needs to trust the private code of the vendor to not actively leak secrets or produce false outputs

## 4.12 Related Work

We now briefly discuss works related to our *VC3* system.

**SGX-based Applications** Hoekstra et al. were the first to discuss (client-side) applications that use SGX [100]. Haven [26] is a recently proposed SGX-based system for executing Windows applications in the cloud. Haven loads a given application together with a *library OS* variant of Windows 8 into an enclave. Haven makes a different trade-off between security and compatibility: it can run unmodified Windows binaries, but its TCB is larger than *VC3*'s by several orders of magnitude. The Haven approach is orthogonal to *VC3*, it neither guarantees integrity for distributed computations, nor does it provide

region self-integrity properties. In practice, Haven and *VC3* could be combined: Haven's enclave-adapted *library OS* variant of Windows 8 could be included in *VC3*'s public enclave code $E^+$. The availability of full a Windows 8 runtime environment inside enclaves could greatly increase flexibility for MapReduce job developers, but would consequently also tremendously enlarge the TCB. Brenner et al. presented an approach for confidential data processing in untrusted Apache ZooKeeper environments [41]. The foundation of their approach are additional *trusted proxies* that mediate connections within distributed computations. The trusted proxies apply symmetric cryptography to achieve data confidentiality and are foreseen to be protected at runtime using hardware isolation technologies like SGX or TrustZone that allow for remote attestation.

**Confidential Computing** Several systems protect confidentiality of data in the cloud. Fully homomorphic encryption and multiparty computation [81, 87] can achieve data confidentiality, but they are not efficient enough for general-purpose computation. CryptDB [159] and MrCrypt [208] use partial homomorphic encryption to run some computations on encrypted data; they neither protect confidentiality of code, nor guarantee data integrity or completeness of results. On the other hand, they do not require trusted hardware. TrustedDB [23], Cipherbase [19], and Monomi [214] use different forms of trusted hardware to process database queries over encrypted data, but they do not protect the confidentiality and integrity of all code and data. Monomi splits the computation between a trusted client and an untrusted server and it uses partial homomorphic encryption at the server. Mylar [160] is a platform for building Web applications that supports searches over encrypted data.

Several systems combine hardware-based isolation [125, 149, 203] with trusted system software [53, 101, 124, 130, 172, 200, 237], which is typically a trusted hypervisor. The Flicker [131] approach uses TXT [108] and avoids using a trusted hypervisor by time-partitioning the host machine between trusted and untrusted operation. Virtual Ghost [60] avoids using a trusted hypervisor and specialized hardware-based isolation mechanisms by instrumenting the kernel.

**Verifiable Computing** Some systems allow the user to verify the result of a computation without protecting the confidentiality of the data or the code [40, 154]. Pantry [40] is a system for proof-based verifiable computations that embraces untrusted storage in a novel way. They show how their system can be used to verify the integrity of MapReduce jobs which are implemented in a subset of C and are compiled to a set of *constraints*. The computational overhead Pantry incurs results in an execution time that is even for small input data sets (<1MB) several orders of magnitude larger than the baseline execution time (milliseconds vs. minutes)—for both prover and verifier. Hawblitzel et al. presented the concept of formally verified Ironclad Apps [96] running on an in turn formally verified software stack on partially untrusted hardware. The *dynamic root of trust measurement* (DTRM) trusted hardware feature—available in Intel TXT-enabled processor and their AMD counterparts—is used for remote attestation of the bootstrapping process of the secure software stack. Hawblitzel et al. report on significant runtime overhead (up to two orders of magnitude) for different applications in their prototype implementation.

**Security-enhanced MapReduce** Several security-enhanced MapReduce systems exist. Airavat [170] defends against possibly malicious map function implementations using differential privacy. Their approach is orthogonal to ours as we trust the user-supplied map and reduce functions. In practice, Airavat and *VC3* could be used in conjunction. SecureMR [223] is an integrity enhancement for MapReduce that relies on redundant computations. An attack is detected when two nodes produce different results for identical inputs. Ko et al. published a hybrid security model for MapReduce where sensitive data is handled in a private cloud while non-sensitive processing is outsourced to a public cloud provider [118]. PRISM [34] is a privacy-preserving word search scheme for MapReduce that utilizes private information retrieval methods. Lin et al. apply the concept of "threshold cryptography" to MapReduce [126]. Their basic idea is that data can only be encrypted/decrypted when at least $n$ mappers collaborate. Attackers are expected to be unable to compromise $n$ or more mappers.

## 4.13 Conclusion

We presented the *VC3* system, a novel approach for the verifiable and confidential execution of MapReduce jobs in untrusted cloud environments. Our approach provides strong security guarantees, while relying on a small TCB rooted in hardware. We showed that our approach is practical with an implementation that works transparently with Hadoop on Windows, and achieves good performance. We believe that *VC3* shows that practical general-purpose secure cloud computation can be achieved. Overall, we think that *VC3* is the first practical approach to answer one of the central open security questions in cloud computing: *how to guarantee confidentiality and integrity while executing sensitive code over sensitive data in a distributed manner?*

Chapter 5

# Conclusion

Modern application software faces a multitude of security risks from very different angles: an attacker may pass malicious input in an attempt to provoke a critical bug, a software component may contain a backdoor that eventually triggers, or, in the case of the highly topical cloud computing, a malicious administrator may virtually at any time manipulate or steal code and data. These are the three settings (abbreviated CLASSIC, BACKDOOR, and CLOUD throughout this work) that were examined in this dissertation. It goes without saying that this list of adversarial settings is far away from capturing all aspects and challenges in the field of "software security". A completely orthogonal setting is for example one where the attacker attempts to extract secrets, e. g., an algorithm or a cryptographic key, from a piece of software by means of *reverse engineering* and the defender attempts to prevent this by applying forms of obfuscation (see e. g., *white-box cryptography* [24, 226]); or an attacker may attempt to learn certain secrets by observing an application's memory access patterns and the defender counters this by applying techniques from the field of *oblivious RAM* [92]; or the trusted hardware a software is running on may not be reliable and the attacker may attempt to not trigger a bug in the software but rather a bug in the hardware by passing carefully crafted malicious inputs to the application (see e. g., the very recent and much-discussed *rowhammer* attack [114, 183]); and this is just to name a few. Further, we also only examined certain instantiation of the given CLASSIC, BACKDOOR, and CLOUD settings.

## 5.1 Summary and Future Work

In the CLASSIC setting, we limited the discussion to software programmed in unsafe programming languages. We focused specifically on the C and C++ programming languages and the threat of memory corruption vulnerabilities and related code-reuse exploitation techniques. We presented novel advanced forms of return-oriented programming (ROP) and showed how existing heuristics-based defenses against ROP conceptually fall short to prevent these. Based on the *loop* technique used in our 64-bit ROP attack approach, we developed a completely new kind of code-reuse attack named *counterfeit object-oriented programming* (COOP). COOP breaks with many up to here commonly held assumptions

on the nature of code-reuse attacks. Specifically, other than in ROP-based attacks, in a COOP attack no branches to non-address taken code locations are executed and no "rogue" returns, excessively many indirect branches, pivoting of the stack pointer, or injection of code pointers can be observed. COOP achieves this by injecting counterfeit C++ objects (along with counterfeit pointers to C++ vtables) and misusing C++'s virtual function dispatch mechanism to direct control flow. As a result, the control flow in a COOP attack resembles much the orderly execution of C++ code. In essence, it cannot be distinguished by defensive measures that do not have a sufficient approximation of an application's high-level C++ semantics. As such, as discussed in Chapter 2, preventing COOP without access to a to-be-protected application's source code is challenging. It is one of the key-insights of our work on COOP that binary-only defenses are maybe generally unsuited for the reliable prevention of advanced code reuse attack techniques. In fact, we are not aware of any available binary-only that can (fully) prevent COOP.

However, we do believe that it is not necessarily impossible to construct effective binary-only defenses against COOP. In fact, in future work, we plan to address this open problem with a hybrid static/dynamic analysis approach: it has already been shown that it is feasible to largely reconstruct an application's C++ class hierarchy by means of static analysis, even if no symbols or RTTI information are available [80]; linking this class hierarchy to *vcall* sites (see Section 2.2.2) in binary code is though yet unsolved (i. e., this amounts to C++ *type inference* on binary code level). Nonetheless, if this would be possible, effective binary-only defenses against COOP could be created. To achieve this, we envision an approach where vcall sites are monitored at runtime—maybe during a training phase—and are gradually linked to the statically reconstructed class hierarchy. In the next step, a relatively precise C++-aware CFI policy could be enforced. Another promising direction for future research is the application of the COOP concept to other object-oriented programming languages. In particular Objective-C [1] appears to be an interesting target in this context, because *(i)* it is widely used, *(ii)* it is unsafe, and *(iii)* its objects in memory carry a rich set of metadata which is used to dynamically dispatch function calls. This metadata is inherently more complex than C++ vptrs/vtables, as Objective-C for example allows to dynamically overwrite member functions of a class at runtime (known as *method swizzling*). As such, "object metadata hijacking" (cf. *vtable hijacking* attacks in C++ as described in Section 2.2.2) could be even more powerful for Objective-C than for C++ and could consequently enable interesting new variants of COOP.

In the Backdoor setting, we likewise limited the discussion to two specific types of backdoors; namely flawed authentication routines and hidden commands. We also only considered binary server applications and emphasized again C and C++. Considering certain kinds of backdoors only, one could say the "lower hanging fruits", allowed us to set a clear focus. Within this focus we demonstrated a heuristics-based dynamic analysis technique for the identification of suspicious binary code artifacts. We successfully demonstrated the viability of this technique across different platforms including a corporate VoIP desk telephone; and demonstrated in a case study that our implementation in the form of

[1] Objective-C is an object-oriented dialect of C which is primarily used on Apple's popular iOS and Mac OS X operating systems.

the tool Weasel can cope with real-world backdoors. However, it should be clear that our approach cannot provide a strong protection against backdoors—in fact, not even against those two types of backdoors in our focus. This follows already from the circumstance that our detection approach relies heavily on heuristics. As has been shown in the Classic setting in Chapter 2, heuristics-based defenses are generally likely to fail when faced with an attacker who is aware of their presence. For example, our detection approach can likely in many cases be evaded by the backdoor installation technique proposed by Andriesse and Bos [15].

However, we believe that the possibility to build forms of *shadow authentication* on top of Weasel as outlined in Section 3.3.4.3 is a promising approach that could help to contain advanced backdoors in the future: legacy binary server applications could be retrofitted to adhere to fine-grained access control policies. This could, at least in some scenarios, prevent attackers from remotely triggering dormant backdoors.

In the Cloud setting we described *VC3*, a novel approach for the provably secure execution of distributed MapReduce applications in the untrusted cloud. We specifically reported on a C++ prototype for the Windows Server operating system. For this prototype strong performance numbers were measured in realistic experimental settings, indicating that *VC3* is likely the first practical and secure general purpose cloud computing framework. Despite the choices we made for our prototype implementation, the *VC3* concept is not bound to any programming language or operating system. In fact, it does not even necessarily rely on Intel's SGX technology. Instead, any hardware TCB which offers isolated execution, remote attestation, and sealing of data may be used.

Although *VC3* offers strong guarantees, there are certain realistic threats outside its current attacker model. This leaves different opportunities for future work atop of *VC3*. For example, the attacker model neglects all kinds of side channel and traffic analysis attacks. However, as we discussed in Section 4.7, an attacker observing the distribution of intermediate key-value pairs between worker nodes in a protected MapReduce job instantly learns the distribution of intermediate keys. While we already partly addressed this problem with *batching* of key-value pairs, one can certainly still do better, e. g., by incorporating an intermediate *shuffle* step akin to techniques used for *oblivious storage* [147].

An important feature of SGX is that it's the operating system's duty to manage an enclave's virtual memory. While this arrangement has different advantages, it also has the severe disadvantage that the operating system necessarily learns about all page faults within an enclave. A malicious operating system may amplify this side channel by loading or evicting pages in a targeted manner; a corresponding comprehensive attack has recently been demonstrated [229]. We plan to address this side channel for *VC3* in the future.

The trusted *VC3* code inside enclaves is written in unsafe C++ and x86-64 assembly and untrusted code from outside the enclave is able to interact with it through a narrow external interface (see Section 4.4). Hence, the Classic and Backdoor settings also apply to *VC3*. For example, untrusted code may try to provoke and to exploit a memory corruption error within *VC3* trusted enclave code. This attack avenue has already been addressed comprehensively in our original *VC3* publication [176] with enclave-adapted compiler techniques that enforce forms of memory safety in combination with imprecise CFI. However, the correctness of *VC3*'s public enclave code should ideally be formally verified and we plan to investigate this in our future research.

# Publications

While working on this dissertation the author contributed to the following publications.

### Peer-reviewed Publications

- Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. Scriptless attacks—stealing the pie without touching the sill. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2012
- Felix Schuster, Stefan Rüster, and Thorsten Holz. Preventing backdoors in server applications with a separated software architecture. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2013
- Felix Schuster and Thorsten Holz. Towards reducing the attack surface of software backdoors. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2013
- Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. Scriptless attacks: Stealing more pie without touching the sill. *Journal of Computer Security, Web Application Security—Web @ 25*, 22(4), 2014
- Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-ROP defenses. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2014
- Jannik Pewny, Felix Schuster, Lukas Bernhard, Christian Rossow, and Thorsten Holz. Leveraging semantic signatures for bug search in binary programs. In *Annual Computer Security Applications Conference (ACSAC)*, 2014
- Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE Symposium on Security and Privacy (S&P)*, 2015
- Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *IEEE Symposium on Security and Privacy (S&P)*, 2015

- Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a TRAP: Table randomization and protection against function reuse attacks. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2015

### Technical Reports

- Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-ROP defenses. Technical Report TR-HGI-2014-001, Ruhr-Universität Bochum, 2014

- Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud. Technical Report MSR-TR-2014-39, Microsoft Research, 2014

### Patent

- Manuel Costa, Felix Schuster, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, and Antony Ian Taylor Rowstron. Trusted execution within a distributed computing system. United States Patent Application Publication, August 2015. Pub. No. US 2015/0229619 A1

# Curriculum Vitae

**Date of birth:** 22/04/1986

**Work experience:**

- Feb. 2012 – July 2015: Ruhr-Universität Bochum (PhD candidate, Research Assistant)
- Aug. 2013 – Nov. 2013 and Aug. 2014 – Oct. 2014: Microsoft Research, Cambridge, UK (Research Intern)
- July 2011 – Dec. 2011: SEC Consult Unternehmensberatung GmbH, Vienna (Security Consultant)
- Dec. 2006 – Mar. 2011: zynamics GmbH, Bochum (Security Analyist)
- Mai 2006 – Sep. 2006: Rotobee 3D Realtime GmbH, Berlin (Intern Game Development)
- Aug. 2005 – Apr. 2006: Wilmersdorfer Seniorenstift, Berlin (Zivildienstleistender)

**Education**

- Oct. 2006 – Oct. 2011: Ruhr-Universität Bochum (Dipl.-Ing. Sicherheit in der Informationstechnik)
- Aug. 1996 – Aug. 2005: Gymnasium Holthausen (Abitur)

# List of Figures

# List of Tables

# List of Listings

# Bibliography


[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control-flow. In *International Conference on Formal Engineering Methods (ICFEM)*, 2005.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity: Principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1), 2009.

[4] Advanced Micro Devices, Inc. AMD64 architecture programmer's manual volume 2: System programming, December 2013. Publication no. 24593 Rev. 3.24.

[5] Dakshi Agrawal, Selcuk Baktir, Deniz Karakoyunlu, Pankaj Rohatgi, and Berk Sunar. Trojan detection using IC fingerprinting. In *IEEE Symposium on Security and Privacy (S&P)*, 2007.

[6] Dave Aitel. An introduction to SPIKE, the fuzzer creation kit. `www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt`, 2002. Presented at Black Hat US.

[7] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, 2010.

[8] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *IEEE Symposium on Security and Privacy (S&P)*, 2008.

[9] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, 2009.

[10] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 1996.

[11] Pedram Amini and Aaron Portnoy. Fuzzing sucks! Introducing Sulley fuzzing framework. pentest.cryptocity.net/files/fuzzing/sulley/introducing_sulley.pdf, 2007. Presented at Black Hat US.

[12] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[13] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11), 2002.

[14] D.P. Anderson and G Fedak. The computational and storage potential of volunteer computing. In *IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, 2006.

[15] Dennis Andriesse and Herbert Bos. Instruction-level steganography for covert trigger-based malware. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2014.

[16] Apache Software Foundation. Hadoop. http://wiki.apache.org/hadoop/ (accessed 11/05/2014).

[17] Apache Software Foundation. Hadoop typed bytes format. http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/typedbytes/package-summary.html (accessed 12/07/2015).

[18] Apache Software Foundation. HadoopStreaming. http://hadoop.apache.org/docs/r1.2.1/streaming.html (accessed 11/05/2014).

[19] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramaratnam Venkatesan. Orthogonal security with Cipherbase. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[20] ARM Ltd. ARM security technology. Building a secure system using TrustZone technology, 2009.

[21] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4), 2010.

[22] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2014.

[23] Sumeet Bajaj and Radu Sion. TrustedDB: A trusted hardware-based database with privacy and data confidentiality. In *IEEE Transactions on Knowledge and Data Engineering*, volume 26, 2014.

[24] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. In *Advances in Cryptology—CRYPTO*, 2001.

[25] Scott Bauer, Pascal Cuoq, and John Regehr. Deniable backdoors using compiler bugs. `http://blog.regehr.org/archives/1241` (accessed 07/07/2015), 2015. Article in PoC-GTFO online magazine issue 8.

[26] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[27] Georg T Becker, Francesco Regazzoni, Christof Paar, and Wayne P Burleson. Stealthy dopant-level hardware trojans. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2013.

[28] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology—CRYPTO*, 1998.

[29] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology—ASIACRYPT*, 2000.

[30] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Advances in Cryptology—CRYPTO*, 1993.

[31] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, 2003.

[32] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Hacking blind. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[33] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[34] Erik-Oliver Blass, Roberto Di Pietro, Refik Molva, and Melek Önen. Prism—privacy-preserving search in MapReduce. In Simone Fischer-Hübner and Matthew Wright, editors, *Privacy Enhancing Technologies*, volume 7384 of *Lecture Notes in Computer Science*. Springer, 2012.

[35] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[36] Microsoft BlueHat Prize. `http://www.microsoft.com/security/bluehatprize/`, 2012. Accessed: 07/04/2015.

[37] Erik Bosman and Herbert Bos. Framing signals—a return to portable shellcode. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[38] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: Tiny trust anchor for tiny devices. In *Annual Design Automation Conference (DAC)*, 2015.

[39] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. Exploit programming: From buffer overflows to "weird machines" and theory of computation. *USENIX ;login:*, 36(6), 2011.

[40] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[41] Stefan Brenner, Colin Wulf, and Rüdiger Kapitza. Running ZooKeeper coordination services in untrusted clouds. In *USENIX Workshop on Hot Topics in Systems Dependability (HotDep)*, 2014.

[42] E. Brickell and Jiangtao Li. Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. In *IEEE International Conference on Social Computing (SocialCom)*, 2010.

[43] Ernie Brickell and Jiangtao Li. Enhanced privacy ID: a direct anonymous attestation scheme with enhanced revocation capabilities. In *ACM Workshop on Privacy in Electronic Society (WPES)*, 2007.

[44] Ernie Brickell and Jiangtao Li. Enhanced privacy ID from bilinear pairing. *IACR Cryptology ePrint Archive*, 2009, 2009.

[45] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Xiaodong Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*. Springer, 2008.

[46] David Brumley and Dawn Song. Privtrans: automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, 2004.

[47] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[48] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2010.

[49] Stephen Checkoway, Ariel J Feldman, Brian Kantor, J Alex Halderman, Edward W Felten, and Hovav Shacham. Can DREs provide long-lasting security? the case of return-oriented programming and the AVC advantage. In *Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE)*, 2009.

[50] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[51] Wei Chen. Here's that FBI Firefox exploit for you (CVE-2013-1690). `https://community.rapid7.com/community/metasploit/blog/2013/08/07/heres-that-fbi-firefox-exploit-for-you-cve-2013-1690` (accessed 07/04/2015), August 2013.

[52] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[53] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[54] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[55] Manuel Costa, Felix Schuster, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, and Antony Ian Taylor Rowstron. Trusted execution within a distributed computing system. United States Patent Application Publication, August 2015. Pub. No. US 2015/0229619 A1.

[56] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large scale analysis of the security of embedded firmwares. In *USENIX Security Symposium*, 2014.

[57] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[58] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[59] Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a TRAP: Table randomization and protection against function reuse attacks. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2015.

[60] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual Ghost: Protecting applications from hostile operating systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[61] Shuaifu Dai, Tao Wei, Chao Zhang, Tielei Wang, Yu Ding, Zhenkai Liang, and Wei Zou. A framework to eliminate backdoors from response-computable authentication. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[62] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: preventing authentication & access control vulnerabilities in web applications. In *USENIX Security Symposium*, 2009.

[63] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology—CRYPTO*, 2012.

[64] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Annual Design Automation Conference (DAC)*, 2014.

[65] Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[66] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[67] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[68] Leonardo De Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *Formal Methods in Computer Aided Design (FMCAD)*, 2009.

[69] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.

[70] Jared DeMott. Bypassing EMET 4.1. `http://bromiumlabs.files.wordpress.com/2014/02/bypassing-emet-4-1.pdf` (accessed 07/04/2015), Feb 2014.

[71] Solar Designer. `http://insecure.org/sploits/linux.libc.return.lpr.sploit.html` (accessed 30/06/2015), August 1997.

[72] David Dewey and Jonathon T Giffin. Static detection of C++ vtable escape vulnerabilities in binary code. In *Symposium on Network and Distributed System Security (NDSS)*, 2012.

[73] Patricio Domingues, Bruno Sousa, and Luis Moura Silva. Sabotage-tolerance and trust management in desktop grid computing. *Future Generation Computer Systems*, 23(7), 2007.

[74] Loïc Duflot. CPU bugs, CPU backdoors and consequences on security. In *European Symposium on Research in Computer Security (ESORICS)*, 2008.

[75] Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann. A framework for automated architecture-independent gadget search. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2010.

[76] Sébastien Duquette. Linux/SSHDoor.A backdoored SSH daemon that steals passwords. `http://www.welivesecurity.com/2013/01/24/linux-sshdoor-a-backdoored-ssh-daemon-that-steals-passwords/` (accessed 07/04/2015), January 2013.

[77] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. XFI: Software guards for system address spaces. In *USENIX Security Symposium*, 2006.

[78] Mplayer (r33064 lite) buffer overflow + ROP exploit. `http://www.exploit-db.com/exploits/17124/` (accessed 07/04/2015), 2011.

[79] Halvar Flake. Structural comparison of executable objects. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2004.

[80] A Fokin, E. Derevenetc, A Chernov, and K Troshina. SmartDec: Approaching C++ decompilation. In *Working Conference on Reverse Engineering (WCRE)*, 2011.

[81] Cedric Fournet, Markulf Kohlweiss, George Danezis, and Zhengqin Luo. ZQL: A compiler for privacy-preserving data processing. In *USENIX Security Symposium*, 2013.

[82] Mike Frantzen and Mike Shuey. StackGhost: Hardware facilitated stack protection. In *USENIX Security Symposium*, 2001.

[83] Ivan Fratric. Runtime Prevention of Return-Oriented Programming Attacks. `http://ropguard.googlecode.com/svn-history/r2/trunk/doc/ropguard.pdf` (accessed 07/04/2015).

[84] Debin Gao, Michael K Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*. Springer, 2008.

[85] Robert Gawlik and Thorsten Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[86] Dimitris Geneiatakis, Georgios Portokalidis, Vasileios P. Kemerlis, and Angelos D. Keromytis. Adaptive defenses for commodity software through virtual application partitioning. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2012.

[87] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing (STOC)*, 2009.

[88] Craig Gentry and S. Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In *Advances in Cryptology—EUROCRYPT*, 2011.

[89] Patrice Godefroid. Random testing for security: blackbox vs. whitebox fuzzing. In *International Workshop on Random Testing*, 2007.

[90] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Gerogios Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[91] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security Symposium*, 2014.

[92] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3), 1996.

[93] Yoann Guillot and Alexandre Gazet. Automatic binary deobfuscation. *Journal in Comp. Virology*, 2010.

[94] WG Halfond, Jeremy Viegas, and Alessandro Orso. A classification of sql-injection attacks and countermeasures. In *IEEE International Symposium on Secure Software Engineering (ISSSE)*, 2006.

[95] Jeffrey S Havrilla. Borland/Inprise Interbase SQL database server contains backdoor superuser account with known password. `http://www.kb.cert.org/vuls/id/247371` (accessed 07/04/2015), 2001.

[96] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[97] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. Scriptless attacks—stealing the pie without touching the sill. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2012.

[98] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. Scriptless attacks: Stealing more pie without touching the sill. *Journal of Computer Security, Web Application Security—Web @ 25*, 22(4), 2014.

[99] Matthew Hicks, Murph Finnicum, Samuel T. King, Milo M. K. Martin, and Jonathan M Smith. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[100] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Carlos Rozas, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[101] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[102] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size does matter in Turing-complete return-oriented programming. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[103] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.

[104] Ralf Hund, Thorsten Holz, and Felix C Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*, 2009.

[105] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[106] Frank Imeson, Ariq Emtenan, Siddharth Garg, and Mahesh V Tripunitara. Securing computer hardware using 3D integrated circuit (IC) technology and split manufacturing for obfuscation. In *USENIX Security Symposium*, 2013.

[107] Intel Corp. Intel 64 and IA-32 architectures software developer's manual—combined volumes 1, 2a, 2b, 2c, 3a, 3b and 3c, September 2013. 325462-048US.

[108] Intel Corp. Intel Trusted Execution Technology. software development guide, 2013. No. 315168-009.

[109] Intel Corp. Software guard extensions programming reference, 2013. No. 329298-001.

[110] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[111] Nicolas Joly. Advanced exploitation of Internet Explorer 10 / Windows 8 overflow (Pwn2Own 2013). `http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php` (accessed 07/04/2015), 2013.

[112] Lori M Kaufman. Data security in the world of cloud computing. *IEEE Security & Privacy*, 7(4), 2009.

[113] Douglas Kilpatrick. Privman: A library for partitioning applications. In *USENIX Anual Technical Conference, FREENIX Track*, 2003.

[114] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *International Symposium on Computer Architecture (ISCA)*, 2014.

[115] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[116] Samuel T. King, Joseph Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang, and Yuanyuan Zhou. Designing and implementing malicious hardware. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[117] Amit Klein. Cross site scripting explained. *Sanctum White Paper*, 2002.

[118] Steven Y Ko, Kyungho Jeon, and Ramsés Morales. The Hybrex model for confidentiality and privacy in cloud computing. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[119] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. `http://users.suse.com/~krahmer/no-nx.pdf` (accessed 07/04/2015), 2005.

[120] Robbert Krebbers. *The C standard formalized in Coq. Draft of PhD thesis.* PhD thesis, Radboud Universiteit Nijmegen, September 2015. `http://robbertkrebbers.nl/research/thesis_draft.pdf` (accessed 26/10/2015).

[121] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity website. `http://dslab.epfl.ch/proj/cpi/` (accessed 19/06/2015).

[122] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[123] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled reverse engineering of types in binary programs. In *Symposium on Network and Distributed System Security (NDSS)*, 2011.

[124] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. MiniBox: A two-way sandbox for x86 native code. In *Usenix ATC*, 2014.

[125] D. Lie, M. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[126] HuaYi Lin, Che-Yu Yang, and Meng-Yen Hsieh. Secure map reduce data transmission mechanism in cloud computing using threshold secret sharing scheme. In Yanwen Wu, editor, *Software Engineering and Knowledge Engineering: Theory and Practice*, volume 115 of *Advances in Intelligent and Soft Computing*. Springer, 2012.

[127] 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 *SIGPLAN Not*, volume 40.6, 2005.

[128] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[129] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface: AMD64 architecture processor supplement. `http://x86-64.org/documentation/abi.pdf` (accessed 07/04/2015), 2013.

[130] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, A. Datta, Virgil D. Gligor, and Adrian Perrig. Trustvisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[131] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for TCB minimization. In *European Conference on Computer Systems (EuroSys)*, 2008.

[132] D. McGrew and J. Viega. The Galois/counter mode of operation (GCM). Submission to NIST Modes of Operation Process, 2004.

[133] Frank Mckeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[134] Microsoft Corp. Enhanced mitigation experience toolkit (EMET) 5.1. `http://technet.microsoft.com/en-us/security/jj653751` (accessed 07/04/2015), November 2014.

[135] Microsoft Corporation. Enhanced mitigation experience toolkit 4.1—user guide, 2013.

[136] Microsoft Developer Network. Argument passing and naming conventions. `http://msdn.microsoft.com/en-us/library/984x0h58.aspx` (accessed 07/04/2015).

[137] Microsoft Developer Network. C run-time library reference: _onexit. `http://msdn.microsoft.com/en-us/library/zk17ww08.aspx` (accessed 07/04/2015), 2012.

[138] Microsoft Security Research & Defense. Introducing enhanced mitigation experience toolkit (EMET) 4.1. `http://www.microsoft.com/security/bluehatprize/` (accessed 07/04/2015), November 2013.

[139] H. D. Moore. Shiny old VxWorks vulnerabilities. `https://community.rapid7.com/community/metasploit/blog/2010/08/02/shiny-old-vxworks-vulnerabilities` (accessed 07/04/2015), 2010.

[140] G. Morris and M. Aubury. Design space exploration of the European option benchmark using hyperstreams. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2007.

[141] Derek G. Murray and Steven Hand. Privilege separation made easy: trusting small libraries not big processes. In *European Workshop on System Security (EuroSec)*, 2008.

[142] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[143] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: Compiler enforced temporal safety for C. In *International Symposium on Memory Management*, 2010.

[144] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 58(4), 2001.

[145] Ben Niu and Gang Tan. Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2013.

[146] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in MapReduce. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2015.

[147] Olga Ohrimenko, Michael T. Goodrich, Roberto Tamassia, and Eli Upfal. The Melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2014.

[148] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: Defeating return-oriented programming through gadget-less binaries. In *Annual Computer Security Applications Conference (ACSAC)*, 2010.

[149] Emmanuel Owusu, Jorge Guajardo, Jonathan McCune, Jim Newsome, Adrian Perrig, and Amit Vasudevan. Oasis: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2013.

[150] Vasilis Pappas. kBouncer: Efficient and transparent ROP mitigation. `http://www.cs.columbia.edu/~vpappas/papers/kbouncer.pdf` (accessed 07/04/2015).

[151] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[152] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, 2013.

[153] Bryan Parno. Bootsrapping trust in a "trusted" platform. In *USENIX Workshop on Hot Topics in Security (HotSec)*, 2008.

[154] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[155] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *ACM SIGMOD International Conference on Management of Data*, 2009.

[156] Mathias Payer, Antonio Barresi, and Thomas R. Gross. Fine-grained control-flow integrity through binary hardening. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2015.

[157] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[158] Jannik Pewny, Felix Schuster, Lukas Bernhard, Christian Rossow, and Thorsten Holz. Leveraging semantic signatures for bug search in binary programs. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[159] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[160] Raluca Ada Popa, Emily Stark, Jonas Helfer, Steven Valdez, Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan. Building web applications on top of encrypted data using Mylar. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[161] Aaron Portnoy. Bypassing all of the things. `https://www.exodusintel.com/files/Aaron_Portnoy-Bypassing_All_Of_The_Things.pdf` (accessed 07/04/2015), 2013.

[162] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (INTERNET STANDARD).

[163] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[164] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *USENIX Security Symposium*, 2003.

[165] Bo Qu and Royce Lu. POWER IN PAIRS: How one fuzzing template revealed over 100 IE UAF vulnerabilities. `https://www.blackhat.com/docs/eu-14/materials/eu-14-Lu-The-Power-Of-Pair-One-Template-That-Reveals-100-plus-UAF-IE-Vulnerabilities.pdf` (accessed 12/07/2015), 2014. BlackHat Europe.

[166] Jeyavijayan Rajendran, Vivekananda Vedula, and Ramesh Karri. Detecting malicious modifications of data in third-party intellectual property cores. In *Annual Design Automation Conference (DAC)*, 2015.

[167] Rapid7 Vulnerability & Exploit Database. Nginx HTTP server 1.3.9–1.4.0 chunked encoding stack buffer overflow. `http://www.rapid7.com/db/modules/exploit/linux/http/nginx_chunked_size` (accessed 07/04/2015), 2013.

[168] Paruj Ratanaworabhan, V Benjamin Livshits, and Benjamin G Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium*, 2009.

[169] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1), 2012.

[170] Indrajit Roy, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for MapReduce. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[171] Mark Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 1.* Microsoft Press, 6th edition, 2012.

[172] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *USENIX Security Symposium*, 2012.

[173] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Anna Shubina. The halting problems of network stack insecurity. *USENIX ;login:*, 36(6), 2011.

[174] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *USENIX Security Symposium*, 1998.

[175] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud. Technical Report MSR-TR-2014-39, Microsoft Research, 2014.

[176] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[177] Felix Schuster and Thorsten Holz. Towards reducing the attack surface of software backdoors. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2013.

[178] Felix Schuster, Stefan Rüster, and Thorsten Holz. Preventing backdoors in server applications with a separated software architecture. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2013.

[179] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[180] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-ROP defenses. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2014.

[181] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-ROP defenses. Technical Report TR-HGI-2014-001, Ruhr-Universität Bochum, 2014.

[182] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[183] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. `http://googleprojectzero.blogspot.de/2015/03/exploiting-dram-rowhammer-bug-to-gain.html` (accessed 11/07/2015), March 2015.

[184] RuggedCom - Backdoor Accounts in my SCADA network? You don't say.. `http://seclists.org/fulldisclosure/2012/Apr/277` (accessed 07/04/2015), 2012.

[185] ProFTPD backdoor unauthorized access vulnerability. `http://www.securityfocus.com/bid/45150` (accessed 07/04/2015), 2010.

[186] Jeff Seibert, Hamed Okhravi, and Eric Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2014.

[187] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, 2012.

[188] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2007.

[189] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice: Automatic detection of authentication bypass vulnerabilities in binary firmware. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[190] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptology ePrint Archive*, 2004.

[191] Richard Skowyra, Kelly Casteel, Hamed Okhravi, Nickolai Zeldovich, and William Streilein. Systematic analysis of defenses against return-oriented programming. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2013.

[192] Backdoor found in Arcadyan-based Wi-Fi routers. `http://it.slashdot.org/story/12/04/26/1411229/backdoor-found-in-arcadyan-based-wi-fi-routers` (accessed 07/04/2015), 2012.

[193] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Symposium on Network and Distributed System Security (NDSS)*, 2011.

[194] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[195] Dawn Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and SSH timing attacks. In *USENIX Security Symposium*, 2001.

[196] Alexander Sotirov. Heap feng shui in JavaScript. `https://www.blackhat.com/presentations/bh-usa-07/Sotirov/Whitepaper/bh-usa-07-sotirov-WP.pdf` (accessed 29/06/2015), 2007. Presented at Black Hat EU.

[197] GDB remote serial protocol. `http://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html` (accessed 07/04/2015).

[198] Sherri Sparks, Shawn Embleton, and Cliff C Zou. A chipset level network backdoor: bypassing host-based firewall & ids. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2009.

[199] Raoul Strackx, Bart Jacobs, and Frank Piessens. ICE: A passive, high-speed, state-continuity scheme. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[200] Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2012.

[201] Bjarne Stroustrup. *The C++ Programming Language, 4th Edition.* Addison-Wesley, 4th edition, 2013.

[202] Subashini Subashini and V Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of network and computer applications*, 34(1), 2011.

[203] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *International Conference on Supercomputing (ICS)*, 2003.

[204] Synergy Research Group. Aws market share reaches five-year high despite microsoft growth surge. `https://www.srgresearch.com/articles/aws-market-share-reaches-five-year-high-despite-microsoft-growth-surge` (accessed 09/07/2015).

[205] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[206] Jack Tang. Exploring control flow guard in Windows 10. `http://sjc1-te-ftp.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf` (accessed 02/07/2015), 2015.

[207] M. Taufer, D. Anderson, P. Cicotti, and C.L Brooks III. Homogeneous redundancy: a technique to ensure integrity of molecular simulation results using public computing. In *IEEE Parallel and Distributed Processing Symposium*, 2005.

[208] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd Millstein. MrCrypt: Static analysis for secure cloud computations. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[209] The PaX Team. Address space layout randomization. `http://pax.grsecurity.net/docs/aslr.txt` (accessed 03/07/2015), 2013.

[210] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8), 1984.

[211] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security Symposium*, 2014.

[212] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2011.

[213] Kuen Hung Tsoi, Kin-Hong Lee, and Philip Heng Wai Leong. A massively parallel RC4 key search engine. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2002.

[214] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. In *International Conference on Very Large Databases (VLDB)*, 2013.

[215] Marten Van Dijk and Ari Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. In *USENIX Workshop on Hot Topics in Security (HotSec)*, 2010.

[216] Mohan Vishwath, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[217] VUPEN Security. Advanced exploitation of Internet Explorer heap overflow (Pwn2Own 2012 exploit). `http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php` (accessed 29/06/2015), 2012.

[218] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

[219] Adam Waksman and Simha Sethumadhavan. Silencing hardware backdoors. In *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[220] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. FANCI: identification of stealthy malicious logic using boolean functional analysis. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2013.

[221] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2012.

[222] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *Machine Learning and Knowledge Discovery in Databases*. Springer, 2011.

[223] Wei Wei, Juan Du, Ting Yu, and Xiaohui Gu. Securemr: A service integrity assurance framework for mapreduce. In *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[224] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: runtime intrusion prevention evaluator. In *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[225] C.V. Wright, L. Ballard, S.E. Coull, F. Monrose, and G.M Masson. Spot me if you can: Uncovering spoken phrases in encrypted VoIP conversations. In *IEEE Symposium on Security and Privacy (S&P)*, 2008.

[226] Brecht Wyseur. *White-Box Cryptography*. PhD thesis, Katholieke Universiteit Leuven, March 2009.

[227] Chris Wysopal, Chris Eng, and Tyler Shields. Static detection of application backdoors - detecting both malicious software behavior and malicious indicators from the static analysis of executable code. *Datenschutz und Datensicherheit*, 34(3), 2010.

[228] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2012.

[229] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[230] 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 Symposium on Security and Privacy (S&P)*, 2009.

[231] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Authentication Protocol. `http://www.ietf.org/rfc/rfc4252.txt` (accessed 07/04/2015), January 2006. RFC 4252 (Proposed Standard).

[232] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C Myers. Secure program partitioning. *ACM Transactions on Computer Systems (TOCS)*, 20(3), 2002.

[233] Samsung printers contain hidden, hard-coded management account. `http://www.zdnet.com/samsung-printers-contain-hidden-hard-coded-management-account-7000007928/` (accessed 07/04/2015), 2012.

[234] Andreas Zeller. Isolating cause-effect chains from computer programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2002.

[235] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Defending virtual function tables' integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[236] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[237] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[238] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.

[239] HongWei Zhou, Xin Wu, WenChang Shi, JinHui Yuan, and Bin Liang. HDROP: Detecting ROP attacks using performance monitoring counters. In *Information Security Practice and Experience*. Springer International Publishing, 2014.