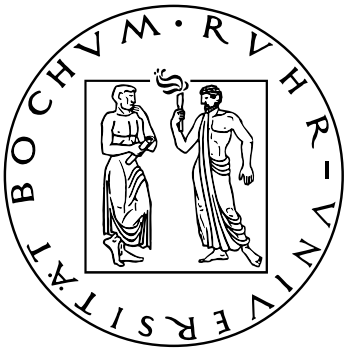
On Improving Automated Program Analysis to Secure Software Systems

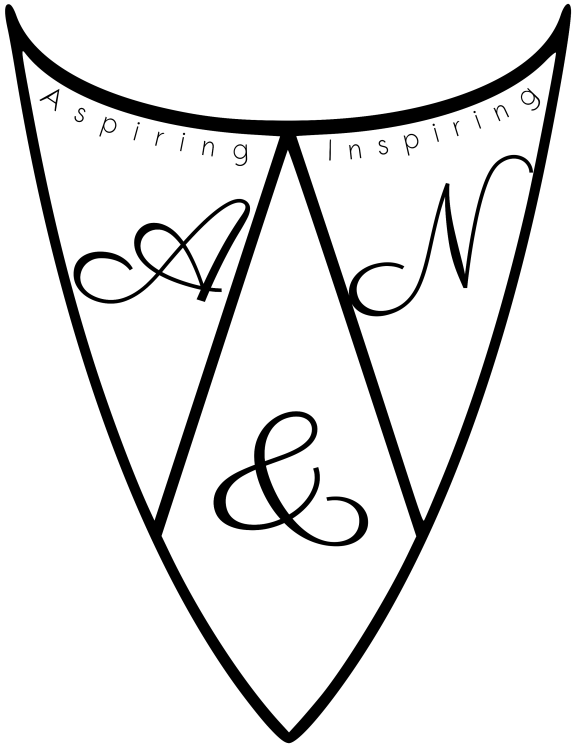
Dissertation
zur Erlangung des Grades eines Doktor-Ingenieurs
der Fakultät für Elektrotechnik und Informationstechnik
an der Ruhr-Universität Bochum



vorgelegt von

Andre Pawlowski
aus Hattingen

2019



Tag der mündlichen Prüfung: 23. Juni 2020

Gutachter:
Prof. Dr. Thorsten Holz, Ruhr-Universität Bochum

Zweitgutachter:
Prof. Dr. Herbert Bos, Vrije Universiteit Amsterdam

Abstract

Software is ubiquitous in our digital age and surrounds our daily life. Therefore, it is essential to assess its correctness and search for vulnerabilities. However, as software is getting bigger and more complex, manually analyzing it becomes harder.

In this thesis, we focus on analyzing software in an automated way to enhance its security and make the following contributions: First, we develop new techniques to analyze software in an automated way. To this end, we implemented tools and algorithms that allow us to extract security-relevant information. Second, we provide techniques to use the extracted information to enhance the security of a given software. Overall, we present four novel automated analysis techniques that can be leveraged to enhance the security of a given software.

First, we developed a generic way to identify pseudo-random number generator (PRNG) and cryptographic hash function (CHF) implementations in closed-source software. To this end, we investigated various PRNG and CHF implementations in popular cryptographic shared libraries and created an interaction model encompassing both algorithm types. Based on this model, we implemented a prototype able to generically find PRNG and CHF implementations in binary executables and show its usefulness for analysts.

Next, we created a method to reconstruct C++ class hierarchies in an automated way in binary executables. To this end, we developed multiple static analysis techniques that allow us to determine the relationship between different classes. Since these classes contain function pointers that are used at various virtual callsites, we can leverage the reconstructed class hierarchies to infer possible target functions. To demonstrate the precision of our developed techniques, we implemented two security instrumentations capable of enhancing closed-source software atop of the analysis results: *vtable protection* and *type-safe object reuse*.

In the third contribution of this thesis, we further advanced the topic of C++ binary-only defenses against so-called vtable-hijacking attacks by developing *VTable Pointer Separation* (VPS). In contrast to related work, this approach sidesteps the accuracy problem by not restricting virtual callsites to a set of valid targets. Instead, VPS achieves more fine-grained protection by restricting virtual callsites to objects created at valid creation sites (i.e., constructors). Furthermore, several inaccuracies in the source-based approach called *VTV* were uncovered by the evaluation, which demonstrates the accuracy improvements VPS provides.

Finally, we present an approach to tailor shared libraries or script interpreters to a given application. To this end, we developed a compiler extension that transforms a given shared library or script interpreter during compile time by removing code not used by the target application. Since not all features a shared library or a script interpreter provides are used by an application, usually unnecessary code is loaded into memory. An attacker finding a vulnerability in an application can leverage this unnecessary code to perform code-reuse attacks. In contrast, shared libraries and script interpreters tailored to an application only contain code necessary to run, therefore, effectively reducing the code an attacker can leverage for her code-reuse attacks. The evaluation demonstrated a significant code reduction and, further, the possibility to remove entire functionalities from an interpreter, such as executing shell commands.

Zusammenfassung

Software ist allgegenwärtig in unserem digitalen Zeitalter. Daher ist es unentbehrlich die Korrektheit von Software zu prüfen und nach Schwachstellen in ihr zu suchen. Da Software jedoch immer komplexer wird, wird es auch schwieriger, sie manuell zu analysieren.

In dieser Dissertation beschäftigen wir uns mit der automatisierten Analyse von Software, um ihre Sicherheit zu verbessern. Zu diesem Zweck haben wir, mit Hilfe neu entwickelter Techniken, Programme und Algorithmen implementiert, mit denen wir sicherheitsrelevante Informationen auffinden können. Zudem zeigen wir auf, wie diese Informationen dazu genutzt werden können, die Sicherheit der Software zu verbessern. Insgesamt unterteilt sich diese Dissertation in vier Themenbereiche.

Im ersten Themenkomplex entwickeln wir eine generische Methode um Implementierungen von Pseudozufallszahlengeneratoren (PRNG) und kryptografische Hash-Funktionen (CHF) in proprietärer, geschlossener Software zu finden. Zu diesem Zweck untersuchten wir eine Vielzahl von verschiedenen PRNG- und CHF-Implementierungen in beliebten kryptografischen Programmbibliotheken und erstellten mit ihrer Hilfe ein Interaktionsmodell. Basierend auf diesem Modell haben wir einen Prototypen zum Auffinden von PRNG- und CHF-Implementierungen in geschlossener Software implementiert und anhand einer ausführlichen Evaluation seine Nützlichkeit für einen Analysten aufgezeigt.

Als Nächstes haben wir eine Methode zur automatisierten Rekonstruktion von C++-Klassenhierarchien in Binärprogrammen erstellt. Dazu haben wir statische Analysetechniken entwickelt, mit denen wir die Beziehungen zwischen Klassen bestimmen können. Da diese Klassen Funktionszeiger enthalten, können wir die rekonstruierten Hierarchien dazu nutzen, Zielfunktionen von indirekten Aufrufen abzuleiten. Um die Präzision unserer entwickelten Techniken zu demonstrieren, haben wir zwei Sicherheitsinstrumentierungen für Binärprogramme entwickelt, die auf die Analyseergbenisse aufbauen.

Im dritten Themenbereich haben wir C++-Verteidigungsmaßnahmen auf Binärebene gegen sogenannte *Vtable-Hijacking*-Angriffe durch die Entwicklung von *VTable Pointer Separation* (VPS) verbessert. Im Gegensatz zu verwandten Arbeiten umgeht VPS Ungenauigkeiten, indem es indirekte Aufrufe auf valide erstellte Objekte beschränkt, anstatt zu versuchen, die Ziele auf eine vorher bestimmte Menge gültiger Ziele zu begrenzen. Die ausgiebige Evaluierung zeigte die Präzisionsgewinnung durch VPS, auch weil mehrere Ungenauigkeiten in einem Quellcode basierten Verfahren namens *VTV* gefunden wurden.

Im letzten Teil dieser Arbeit stellen wir einen Ansatz vor, um Programmbibliotheken und Skriptinterpreter auf ein Zielprogramm zuzuschneiden. Dazu haben wir eine Compilererweiterung entwickelt, die von dem Zielprogramm nicht verwendeten Binärcode aus Programmbibliotheken und Skriptinterpretern entfernen kann. Dieser nicht verwendete Binärcode wird normalerweise zur Laufzeit des Programmes in den Arbeitsspeicher geladen. Ein Angreifer, der eine Sicherheitslücke ausnutzt, kann diesen nicht verwendeten Binärcode für sogenannte *Code-Reuse*-Angriffe benutzen. Wenn die Programmbibliotheken und Skriptinterpreter allerdings auf das Zielprogramm zugeschnitten sind, enthalten diese keinen ungenutzten Binärcode, was den Angreifer einschränkt. Die Auswertung zeigte eine signifikante Programmcode-Reduktion und darüber hinaus die Möglichkeit, ganze Funktionalitäten wie das Ausführen von Shell-Befehlen aus Skriptinterpretern zu entfernen.

Acknowledgements

It is with great pleasure to thank everyone that helped to make this thesis possible. First, I would like to thank my adviser Prof. Dr. Thorsten Holz for giving me the opportunity to work on my thesis. He provided me with advice and gave me the flexibility to work on research topics I was interested in. Next, I want to thank all colleagues I had the pleasure to work with during the course of my studies. I am thankful to Ali Abbasi, Chris Ouwehand, Cristiano Giuffrida, Dennis Andriesse, Elias Athanasopoulos, Erik van der Kouwe, Herbert Bos, Nicolai Davidsson, Philipp Görz, and Victor van der Veen for working with me on the interesting research topics which are part of this thesis. My appreciation goes to Behrad Garmany, Benjamin Kollenda, Cornelius Aschermann, Dennis Tatang, Emre Güler, Florian Quinkert, Jan Wiele, Jannik Pewny, Joel Frank, Moritz Schlögel, Philipp Koppe, Robert Gawlik, Sergej Schumilo, Teemu Rytilahti, Thorsten Eisenhofer, and Tim Blazytko for the enlightened discussions we had at the systems security chair.

My special thanks goes to my friend, best man, and "work wife" Moritz Contag. Our fruitful discussions and your valuable comments on problems I was working on helped a lot to shape this thesis.

Above all, I wholeheartedly want to thank my wife Nisi. Your support and faith in me, especially when I was about to give up, is what picked me up and made this thesis possible. Without you, I am sure, I would not have finished it. Thank you!

Contents

Chapter 1

Introduction

1.1 Motivation

The increasing usage of information technology has lead to a development in which almost every new device has a way to connect with other devices. Managing these connections and providing a convenient way for a user to interact with these devices is done by software. Today, software is ubiquitous. Since software surrounds our daily life, it is essential to assess its correctness and its security. Analyzing software can have multiple reasons, for example, not trusting the vendor, scrutinizing the distribution channel, or verifying the work of the developers. An analyst always prefers analyzing the source code of a program rather than the compiled binary, since the source code provides a wealth of additional information missing in the binary format, such as data-type information. However, proprietary, closed-source software is only available as binary executable. Having no source code for a software available raises additional security concerns. First, if a vendor no longer supports a software or does not fix a known vulnerability, protecting this application becomes way more difficult since modifying high-level source code is not possible. Second, the assessment becomes more difficult since during the compilation information is removed from the code. Implementation issues leading to vulnerabilities, such as wrongly used algorithms, or design flaws, such as using a self-designed cryptographic algorithm, would be easier to detect if source code were available, especially in complex software systems. Consider for example an analyst who wants to check whether the implementation or usage of a cryptographic function is correct (e.g., a pseudo-random number generator or cryptographic hash function). The analyst has to locate this function in the binary before the actual security assessment can begin. Therefore, identifying the function targeted by the assessment places an additional hurdle on the overall analysis process. Considering that over time software gets more complex and the binaries getting larger, we expect that this process is getting more involved [11, 30, 95, 97, 107]. It is thus necessary to automate this initial step of identifying functions of interest, as it would benefit the overall analysis process and, as a result, strengthen the security of our computer systems. Such being the case, two questions follow naturally: *What kind of new approaches can be developed*

to support this initial analysis step, and how can these approaches be used to improve the security of the software?

One way to handle software complexity during the development process on the codebase level is the choice of the programming language. The features of a programming language are an essential factor in handling complexity for a developer. Large software projects often rely on the system programming language C++ (e.g., Chrome Browser). C++ lets the developer follow the object-oriented programming model which allows them to group code and data together as classes. Developers can easily reuse code through class inheritance and are not forced to duplicate it. The concept of polymorphism gives them the additional flexibility to provide a single interface for objects of different types. However, compiling C++ code that uses polymorphism creates constructs on the binary level that complicate binary analysis. As a result, analyzing the binary to assess the security of it gets more involved, and new techniques are required that can precisely analyze these constructs.

To make things worse, an attacker finding a vulnerability in a C++ program can potentially hijack the low-level constructs. Polymorphism on the binary level is implemented through *virtual function tables (vtables)* which contain function pointers. In practice, hijacking these vtables is a common exploitation technique widely used in exploits that target complex programs written in C++ such as web browsers and server applications [148]. To this end, the adversary injects a malicious vtable and diverts the control flow through the usage of a function pointer in this table. Since modern operating systems make executable memory locations not writable at the same time, a technique called $W \oplus X$ [50] or data execution prevention (DEP) [108], attackers resort to re-use existing code in the program as their shellcode. Reusing existing code for exploitation, called a *code-reuse attack*, comes in a variety of different forms (e.g., ret2libc [152] or return-oriented programming [136] (ROP)). To mitigate such attacks different forms of defenses have been developed and are deployed, such as address space layout randomization (ASLR) [122] and control-flow integrity (CFI) [149]. They have in common that they are integrated into the compiler or depend on specific compiler options and thus need source code access to work (e.g., ASLR requires programs to be compiled as position-independent code, and CFI inserts checks at indirect branches). However, vendors of proprietary software can have different reasons for not deploying such mitigations into their product, such as not supporting the software anymore or simply not having the resources to do it. As a result, proprietary software not deploying those mitigations poses a great risk to its users. One way to reduce this risk is to deploy mitigations by instrumenting or rewriting the proprietary application with security mechanisms. However, adding security measures into binaries is a difficult problem as the application has to be precisely analyzed. An inaccurate binary analysis may lead to breaking the program. Generally, there are two ways retrofitted security measures can enhance a program without breaking it: First, the developed binary analysis techniques are precise enough to guarantee that the target application does not break. Second, the instrumentation used for the target application can detect possible errors of the analysis and thus act accordingly. Either way, retrofitted security techniques do not only have to take their protection guarantees into account, but also have to cope with the problems binary analysis entails. And since C++ code using polymorphism complicates the binary program structure, new approaches that can handle the C++ low-level constructs are necessary.

Other than using the features provided by the programming language to tackle software complexity on the codebase level, developers also resort to reuse existing code on the eco-system level. A common way to reuse code in an eco-system is through the use of shared libraries. These give the developers the possibility to focus solely on the user-facing application rather than reimplementing common functionalities such as memory management or string processing functions over and over again. However, since not all code of an integrated shared library is used in a developed program, the downside of this concept is that unnecessary code is loaded into memory. Since attackers reuse existing code for their exploitations, shared libraries offer them a plethora of (mostly) unused code and thus a variety of functions or code parts to choose from. A study across 2,016 applications of the Ubuntu Desktop environment finds that only 5% of the *libc*, the standard library for the C programming language, is used by an application [129]. One way to remove the unused code of a shared library is to statically link it against the target application. Static linking resolves the external functionalities and plainly copies them into the application. This allows the linker to remove the unnecessary code and thus reduce the availability of code snippets an attacker can choose from for a code-reuse attack. As a result, no shared library is needed to execute the application since all library-provided functionalities are part of the program itself and hence available in memory. However, this increases the complexity in managing software updates: since each application has to be statically linked with all used libraries, each has to be updated if a vulnerability is found in the code of the included libraries. Therefore, a user has to decide between unused code an attacker can leverage for a code-reuse attack and manageability of software updates at the moment. That is why new methods of combining manageability and code reduction are required.

A similar problem with unused code holds for applications written in interpreted languages, such as PHP, Python, or Ruby: The interpreter is a complex piece of software and offers more functionality than the application actually requires [128]. Hence, an attacker that is able to inject her own script code into the application can leverage these unused methods to execute her exploit. Some interpreters, such as PHP, offer a configuration option to disable certain functionalities. This is especially interesting in environments in which untrusted scripts are executed (such as Google App Engine [71]). However, since the code that provides these functionalities is still available in the script interpreter, an attacker might be able to bypass the restrictions and escape the interpreter's internal sandbox [109, 118]. In contrast to native code applications, the compiler or linker is not able to remove unused functionalities from the script interpreter. Hence, new approaches and techniques are required to remove unused code from the interpreter.

1.2 Outline and Contributions

In this thesis, we explore how automated program analysis can contribute to securing software. To this end, we developed new analysis techniques to identify security-relevant information, such as cryptographic algorithms or specific low-level code constructs. Mainly, the contributions are twofold: First, excavating security-relevant information from a given program in an automated way. Second, using this information to enhance the program's

security (e.g., with additional, more fine-grained security checks). In the following, we give an outline of this thesis and list the contribution of each chapter.

Chapter 2: Technical Background This chapter provides the technical background needed to understand the rest of this thesis. First, it gives an overview of the field of program analysis, with a focus on the analysis of binary executables. This overview introduces techniques and abstractions used for both static and dynamic analysis. Secondly, this chapter provides a brief introduction to the programming language C++ and the concept of polymorphism. The focus lies on how the compiler implements the concept of polymorphism on the binary level. As program analysis and C++ internals are a very broad field, the given overviews only cover methods relevant to understanding this thesis and are far from complete.

Chapter 3: Automated Identification of Cryptographic Functions in Binaries We present the design of an analysis approach to precisely identify pseudo-random number generator (PRNG) and cryptographic hash function (CHF) implementations in a given program. Since PRNGs and CHFs represent crucial components of today's security ecosystem, a flaw in their implementation or usage can have dire consequences for the security of a given system, such as enabling an adversary to decrypt encrypted messages or forge signatures. Therefore, assessing the correct implementation and usage of such algorithms is essential to ensure their proper operation. If source code is available, finding these algorithms in a given application or library is straightforward due to the wealth of information the code provides. However, during the compilation process most of this information is lost and proprietary, closed-source programs are only available as binary code. Hence, identifying the algorithms in a closed-source program is a crucial step an analyst has to solve before the actual security assessment of the algorithm can start. To simplify the work for an analyst, our novel approach identifies PRNG and CHF implementations in binary executables in an automated way. Instead of relying on signatures of existing implementations or magic constants, our approach works in a generic manner. To this end, we investigated PRNG and CHF implementations in popular cryptographic shared libraries. We found similarities and based on these insights, we built a generic model encompassing the interactions of both types of algorithms with other components of the application and the memory. The model allows us to abstract away the reliance on concrete implementation details. Therefore, we regard the corresponding implementation of the PRNG and CHF as blackbox and only consider their interactions with the rest of the program. Our approach is built on the observation that PRNGs and CHFs create data with a certain level of randomness. To identify this property, we run the function suspected to correspond to a PRNG or CHF multiple times to see if the generated data looks random. We implemented a prototype called *TropyHunter* based on our proposed approach and evaluated it on a diverse set of open and closed-source programs. Although *TropyHunter*'s purpose is to analyze closed-source programs, the evaluation on open-source software allows us to generate a ground truth we can compare against. The results show that our approach is beneficial to an analyst and generically finds PRNG and CHF implementations in binary executables.

Chapter 4: Uncovering Class Hierarchies in C++ Binaries With the help of *TropyHunter* an analyst is able to generically identify PRNG and CHF implementations in a binary executable. However, its evaluation has shown that the presence of function pointers in a program poses a problem to the analysis process. This is especially the case for programs developed in C++, as high-level concepts used in object-oriented programming paradigms are often translated into binary code using function pointers. Such code is harder to grasp than, e.g., traditional procedural code, since features such as polymorphism or inheritance generally add complexity through, e.g., dynamically computed branches or interface code supporting objects of different types. Hence, a deep understanding of interactions between instantiated objects, their corresponding classes, and the connection between classes would vastly reduce the time it takes an analyst to understand the application, and offer the possibility to improve automated analysis tools. To improve the binary analysis on C++ programs, we developed a novel automated analysis approach to reconstruct class hierarchies. To this end, we identify C++ code constructs used for polymorphism, called virtual function tables (short *vtables*). From the usage of these vtables in the code we can then deduce the connection between the classes. Further, we can use this information to resolve the targets of virtual callsites and thus can predict possible values of the corresponding function pointers. The analysis does not rely on optionally embedded metadata information (which is usually removed during the compilation process), does not rely on particular compiler flags, and works on commercial-of-the-shelf (COTS) software. We implemented an analysis framework called *Marx* and added this novel technique to it. Despite its obvious usage to aid the manual analysis process of a C++ program, we further show that the derived information can be reliably used to enhance the security of closed-source applications. To showcase this property, we developed two binary-level defenses using *Marx*'s analysis results. The first application is a *vtable protection* system to prevent virtual calls to methods that do not belong to the class hierarchy and mitigate vtable-hijacking attacks. The second application is a custom heap allocator to support *type-safe object reuse* by placing newly allocated objects in memory pools based on their type. The results show that the extracted information is not only valuable for an analyst, but also is precise enough to enhance the security of binary executables.

Chapter 5: Excavating C++ Constructs from Binaries to Protect Dynamic Dispatching With the help of *Marx*, we can build a vtable protection system and a custom heap allocator to support type-safe object reuse. Although the analysis reconstructs the class hierarchies accurately, the remaining imprecision of the analysis results still leaves wiggle room for an attacker. To further improve binary-level defenses against vtable hijacking in C++ applications, we developed *VTable Pointer Separation* (VPS). In contrast to the vtable protection system build with *Marx*'s analysis results, as well as other previous binary-level defenses, our novel approach does not match classes to virtual callsites. Instead, VPS achieves a more accurate protection by restricting virtual callsites to validly created objects. More specifically, VPS ensures that virtual callsites can only use objects created at valid object construction sites (i.e., constructors), and only if those objects can reach the callsite. Moreover, VPS explicitly prevents false positives (falsely identified vir-

tual callsites) from breaking the binary, an issue existing work does not handle correctly, if at all. Furthermore, just like the type-safe object reuse defense built with the help of *Marx*, VPS also protects against dangling pointers without any modification. The results show that this binary-level defense has a similar accuracy as protection mechanisms relying on source-code access. Further, the evaluation uncovered several inaccuracies in a widely-deployed source-based approach called *VTV* that is considered a state-of-the-art C++ defense.

Chapter 6: Towards Automated Application-Specific Software Stacks While the previous approaches focusing on C++ code try to prevent the exploitation of a vulnerability directly, in our next method, we try to reduce the amount of code an attacker can reuse for her shellcode. Since software complexity has increased over the years, one common way to tackle this complexity during development is to encapsulate features into shared libraries. This allows developers to reuse already implemented features instead of reimplementing them over and over again. However, not all features provided by a shared library are actually used by an application. As a result, an application using shared libraries loads unused code into memory, which an attacker can use to perform code-reuse and similar types of attacks. The same holds for applications written in a scripting language such as PHP or Ruby: The interpreter typically offers much more functionality than is actually required by the application and hence provides a larger overall attack surface. We tackle this problem and propose a first step towards automated application-specific software stacks. We present a compiler extension capable of removing unneeded code from shared libraries and—with the help of domain knowledge—also capable of removing unused functionalities from an interpreter's code base during the compilation process. We evaluated our approach on a diverse set of applications and showed the effectiveness of the code reduction in shared libraries. Further, the results demonstrate that this approach can reduce the number of sensitive functions in an interpreter up to the point where a complete functionality, such as shell command execution, is completely removed.

1.3 Publications

The work presented in this thesis is based on several publications at academic conferences as well as yet unpublished material. An overview of the publications related to the content of this thesis is given in the following. Additionally, other publications are listed that emerged during the Ph.D. studies but were not used in this thesis.

Chapter 3 is based on yet unpublished material. The work was conducted together with Philipp Görtz, Paul Gerste, Ali Abbasi, and Thorsten Holz.

In Chapter 4, we present a publication presented at the Network and Distributed System Security Symposium (NDSS) 2017 [120]. This work was conducted together with Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. The promising results lay the foundation of the follow-up research published at the Annual Computer Security Applications Conference (ACSAC) 2019 [121] which is presented in Chapter 5. This work was performed together

with Victor van der Veen, Dennis Andriesse, Erik van der Kouwe, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos.

Chapter 6 was published at the European Symposium on Research in Computer Security (ESORICS) 2019 [48] and is based on the master's thesis of Nicolai Davidsson. The research was conducted together with Nicolai Davidsson and Thorsten Holz. Chapter 6 also contains information that is included in our technical report [49].

My studies resulted in several other publications that are not part of this thesis. Together with Moritz Contag, Robert Gawlik, and Thorsten Holz, we published a new defense for function table randomization at the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) 2018 [37]. In cooperation with Moritz Contag, Guo Li, Felix Domke, Kirill Levchenko, Thorsten Holz, and Stefan Savage, a study on emission defeat devices in automobiles was presented at the IEEE Symposium on Security and Privacy (S&P) 2017 [38]. Together with Victor van der Veen, Enes Goktas, Moritz Contag, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida, we published a control-flow hijacking defense on the binary level at the IEEE Symposium on Security and Privacy (S&P) 2016 [158]. An information-leak detection system for scripting engines was published at the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) 2016 [64] together with Robert Gawlik, Philipp Koppe, Benjamin Kollenda, Behrad Garmany, and Thorsten Holz. In cooperation with Moritz Contag and Thorsten Holz, we published a probabilistic obfuscation approach at the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) 2016 [119]. A security assessment of the Eduroam network was published at the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec) 2015 [23] together with Sebastian Brenza and Christina Pöpper.

A comprehensive list of publications that the author has contributed to during the course of the work on the dissertation is given on page 131.

Chapter 2

Technical Background

Before we describe in detail the analysis techniques developed and evaluated throughout this thesis, we introduce the technical knowledge needed to understand the following chapters. We start by giving a brief introduction to *program analysis*. More specifically, we focus on the specialized field of *binary analysis*, which is used in Chapter 3, Chapter 4, and Chapter 5. We follow the common division into static and dynamic analysis approaches to further categorize them. Although this thesis uses mostly static analysis, an additional overview of dynamic analysis gives valuable insights into existing techniques and a more comprehensive understanding of the advantages and disadvantages of both categories.

Next follows a brief introduction into the system programming language C++ and the concept of *polymorphism*. How polymorphism translates from a high-level construct into the binary level is explained in detail. This technical background is necessary to understand the analysis techniques presented in Chapter 4 and Chapter 5.

2.1 Program Analysis

Program analysis describes the process of analyzing the behavior of computer programs. It is used amongst others for checking the correctness of the application, optimizing code, or finding security vulnerabilities. For example, compilers such as LLVM or GCC use program analysis techniques to optimize the given code during the compilation process. There are mainly two strategies to perform program analysis: static program analysis and dynamic program analysis. Static program analysis reasons about the application without executing it, whereas dynamic program analysis is performed during runtime of the application. Moreover, an analysis using a combination of both strategies is also possible. However, the distinction between static and dynamic program analysis is not always clear.

Program analysis targets mainly the control or data flow of a program. In control-flow analysis, the reachability of code locations is determined to, for example, search for vulnerabilities. More specifically, it is examined which parts of the program can be reached from a specific code location or from where the control flow reaching a code location might originate. When analyzing the data flow, the dependencies and relationships of different

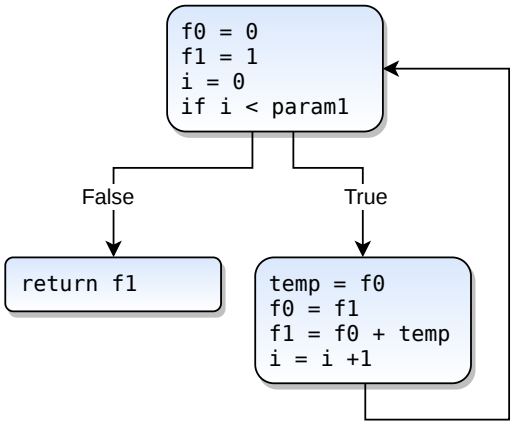


Figure 2.1: Example of a control-flow graph. Each node represents a basic block that contains a code sequence which only has one entry and one exit. The edges between the nodes depict the possible control flow.

Normal

```
1  x = a + b
2  a = x - b
3  y = a + x
4  x = y + a
5  a = b * 2
```

a)

SSA

```
1  x_0 = a_0 + b_0
2  a_1 = x_0 - b_0
3  y_0 = a_1 + x_0
4  x_1 = y_0 + a_1
5  a_2 = b_0 * 2
```

b)

Figure 2.2: Difference between code in a normal representation in a) and the SSA form in b). In the case of the variable `a`, only one distinct variable name exists in the normal representation. Whereas in the SSA form, the variable `a` is split into `a_0`, `a_1` and `a_2`.

data structures are examined. A common abstraction used for control-flow analysis is the Control-Flow Graph (CFG) and for data-flow analysis, the Static Single Assignment form (SSA).

A CFG represents the control flow of an application in a directed-graph representation. Each node of this graph is called a basic block. A basic block is a code sequence executed consecutively with only one entry and one exit (i.e., no branching code instruction in between). The edges between the nodes in the CFG depict the control flow between the basic blocks [7]. For example, an edge can represent a branch of an if-statement at the end of a basic block. Figure 2.1 depicts an example of a CFG. This representation is often internally used by compilers (e.g., LLVM or GCC) to reason about the code and perform optimizations on it.

SSA is a code representation in which each variable is assigned exactly once, and each variable is assigned before it is used [45]. This is mostly done by creating new versions of a variable each time it gets assigned a new value. In the example shown in Figure 2.2, each variable name has a numerical number as a version. As soon as the variable gets a new

value assigned, the numerical version gets incremented. This way, the locations assigning variables in the code can be easily identified, and the data flow of variables can be tracked.

Various additional representation forms exist in program analysis, such as the data-flow graph, which is a directed graph representing the data flow of each variable in the application, or the call graph, a directed graph representing the control flow between functions in the application. However, describing all of them is out of scope for this thesis, and we refer to the corresponding literature for a more exhaustive list [5, 113, 140].

Binary Analysis Program analysis performed on the compiled binary of the application is called *binary analysis* (also called *reverse code engineering* or simply *reverse engineering*) [54]. This form of program analysis is mostly performed if the source code of an application is not available. Typical goals for binary analysis is finding vulnerabilities in closed-source software or understanding the inner workings of an application (e.g., analyzing malware to understand what it does).

During the compilation process, information available in the source code, such as data-type information or function names, is removed. Although it is possible to preserve a large part of the information during compilation, it is only used for debugging purposes. Hence, this wealth of information is usually not available in released application binaries. Therefore, in addition to the problems that program analysis has on source code (e.g., the *halting problem* [153]), binary analysis also suffers from a lack of information. Furthermore, even if the source code of the analyzed application is available, the compiler transforms the code during compilation to optimize it. As a result, the source-code structure does not have to match the binary-code structure.

The same abstract representations used for program analysis on source code can also be used for binary analysis. For example, in the case of a CFG, the basic blocks contain assembler instructions instead of source-code instructions. In the case of the SSA form, the variables are represented through CPU registers instead of actual source-code variables. However, creating some of the representations is more involved on the binary level than on source code. For example, creating a complete CFG for a function is not always feasible since unequivocally identifying the code of a function is not always possible [10]. Hence, an analyst or analysis tool may have to work with an imprecise representation.

Static Binary Analysis The *static binary analysis* is performed without executing the application. This gives an analyst the possibility to reason about the application as a whole. However, in contrast to dynamic analysis, no concrete values (memory or register values) are available for the analysis since the application is not executed.

One method to tackle this shortcoming is *Symbolic Execution* (SE) [88]. This technique emulates the target application by using symbolic values as input. More specifically, the emulator replaces each input by a symbol representing it instead of using concrete input values. As a result, instead of actual values, an expression is created for each register and memory value. For example, at a branching instruction, the symbolic execution creates an expression representing the condition containing the constraints that have to be satisfied to follow the branch. Popular tools providing symbolic execution for analysis purposes are Klee [27] and S2E [33].

Dynamic Binary Analysis The *dynamic binary analysis* is performed during the runtime of the application. Therefore, it can rely on concrete values that are processed during the execution (e.g., inspect register values or memory locations). A typical usage of dynamic analysis is debugging, such as attaching a debugger to an application running, stopping the execution on specific instructions, and inspecting the contents of registers and memory.

Executing an application is not always possible, e.g., if the host system has a different CPU architecture than the target application requires. Nor is executing an application always desirable, e.g., if an executed malware could access the data on the host system. To this end, the target application can be emulated in a virtual environment. During emulation, the code of the application is run by an emulator. Because of the virtual environment, the emulated code does not interact with the host system. As a result, changes to files or the system are not forwarded to the host's environment.

One major drawback of dynamic analysis is that only executed code can be inspected. As a result, dynamic analysis suffers from code coverage problems. Take, for example, an application with multiple error-handling routines. If no error occurs during the execution of the application, the error-handling routine is not executed. Hence, an analyst searching for bugs might miss one in such a routine. One popular way to automate searching for bugs and increasing code coverage is *fuzzing*. In fuzzing, the target application is executed with different inputs while monitoring it for unexpected behavior (e.g., crashes). The input generation is done by the fuzzer and can be created following different strategies (e.g., by mutating an already existing input) [14,163]. As with the most prominent fuzzer AFL [163], fuzzing is often performed on the source-code level. However, fuzzer targeting binary applications to search for vulnerabilities also exist [14,130].

2.2 C++

This section provides background on C++ internals needed to understand how Chapter 4 and Chapter 5 handle C++ binaries. We focus on how high-level C++ constructs translate to the binary level. For a more detailed overview of high-level C++ concepts, we refer to the corresponding literature [144].

2.2.1 Object-oriented Programming

C++ is an object-oriented programming (OOP) language which compiles to native code. In OOP, *classes* are data types used to instantiate concrete *objects*. On the latter, one can call *functions* (also called *member functions* or *methods*) as defined by the class an object was instantiated from. Apart from functions, classes can also define *attributes* and hence couple code (via functions) and data (via attributes) together.

OOP allows classes to *inherit* functions and attributes from other classes. This defines a relation. The class providing functions and attributes to another class is commonly called the *base* class in that relation, whereas the class inheriting these is called the *derived* class. This concept leads to what is called a *class hierarchy*: Every class is related to either zero or multiple bases as well as zero or multiple derived classes. Class hierarchies can be depicted as a directed graph in which the inheritance relation is given by the direction

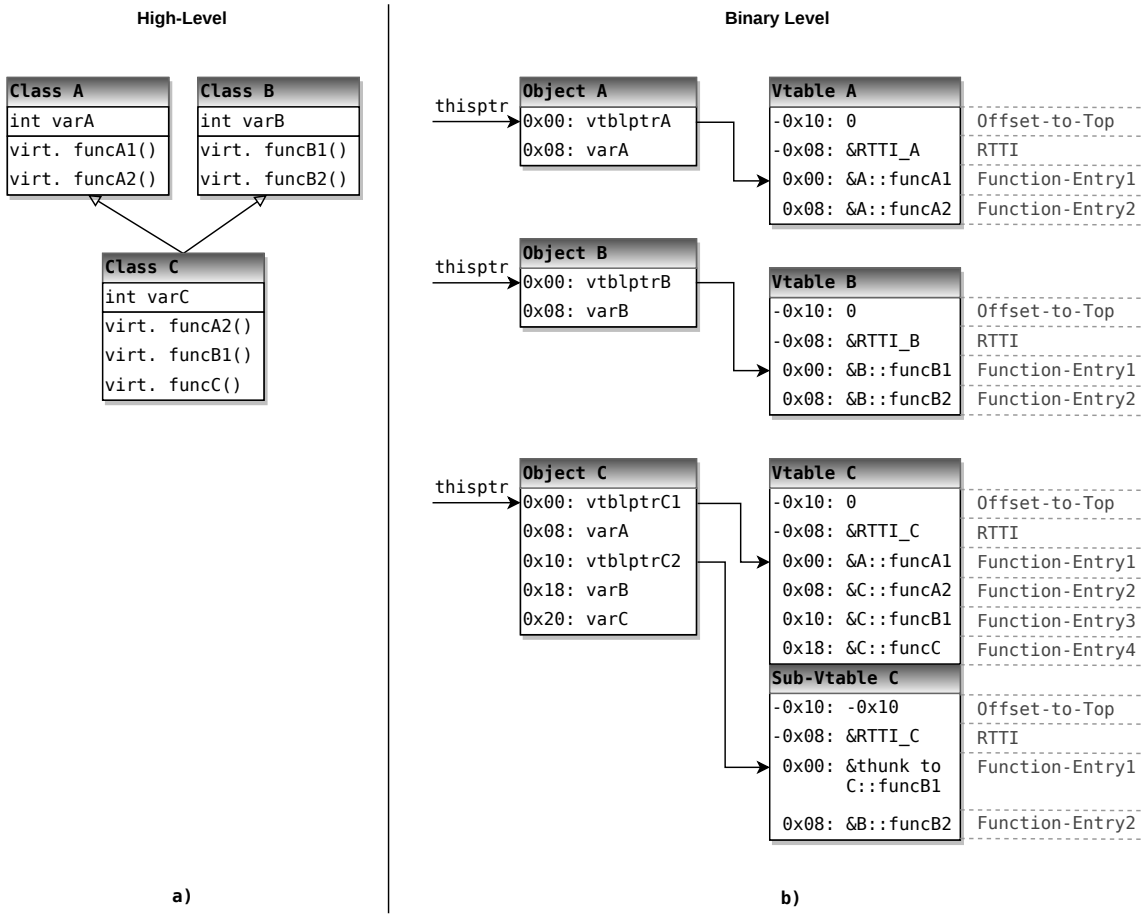


Figure 2.3: Example C++ class hierarchy layout shown at a high-level in a) and b) shows its layout in native code. Class `C` inherits attributes and functions of both class `A` and `B`. Further, class `C` overrides `funcA2` and `funcB1` and provides its own implementation, `C::funcA2` and `C::funcB1`.

(base or derived class). If a class inherits from multiple base classes, this is referred to as *multiple inheritance*; otherwise, it is called *single inheritance*.

Classes may add several modifiers to their functions. One of the most important throughout this thesis is the `virtual` modifier. If this modifier is used on a function, a derived class is free to *override* said function and provide its own implementation. This concept is called *polymorphism*, where a single function invocation may behave differently, depending on the context in which it is called. More concretely, the programmer can call a virtual function on *either* an object of the base or any derived classes. Depending on the type of the object, the appropriate implementation of the function is called. This, in turn, allows programmers to work on the most general class and vastly simplify their code. In cases where the compiler cannot determine statically on which object the function is to be invoked, the selection of the appropriate implementation is made at runtime. Furthermore, *abstract* base classes represent an edge case: They provide several virtual functions, but *no* implementation on their own (*pure virtual* functions). This forces the deriving class to implement functions conforming to the declaration the base class provides.

Figure 2.3 a) depicts an exemplary relation of three classes `A`, `B`, and `C`. Class `C` inherits the functions from classes `A` and `B`, i.e., one may call the functions `funcA1`, `funcA2`, `funcB1`, `funcB2` on an object of class `C`, in addition to the functions class `C` provides itself. The same is true for the attributes; hence, class `C` allocates space for attributes `varA`, `varB`, and `varC`. Further, `C` overrides `funcA2` and `funcB1`, i.e., it specifies a more fitting implementation for its class.

To create an object of a specific class, the operator `new` can be used, amongst others. It allocates space for the object (whose size is mostly determined by its attributes) and calls a designated initialization function that initializes the object's attributes with meaningful values. This function is called a *constructor*. Similarly, a *destructor* releases further resources the constructor requested previously, and is usually invoked through the operator `delete`.

In the following, we explain how the aforementioned concepts are implemented on the binary level.

2.2.2 Virtual Function Tables

On the binary level, polymorphism is implemented with the help of what is called a *virtual function table* (*vtable* for short). It contains the addresses of all virtual functions a class provides. Each class containing at least one virtual function has a vtable. Each object of such a class contains a pointer to the corresponding vtable, which is typically stored in read-only memory. Since each class has its own corresponding vtable, it can also be considered as the type of the object. In the following, we refer to the pointer to the vtable as *vtblptr*, while the pointer to the object is called *thisptr*.

The Itanium C++ ABI [60] defines the vtable layout for Linux systems[1]. Figure 2.3 b) depicts this layout on the binary level. The *vtblptr* points to the first function entry in the vtable, and the vtable contains an entry for each virtual function (either inherited or newly declared) in the class. For example, in Figure 2.3 b), class *B*'s vtable contains two

[1]Linux uses the Itanium C++ ABI for x86-64 (amd64).

function entries because the class implements virtual functions *funcB1* and *funcB2*. Class C inherits from two classes, A and B, and therefore has two vtables (a base vtable and one sub-vtable). The base vtable contains all virtual functions inherited from class A and implemented by class C. The sub-vtable is a copy of class B's vtable with a special entry that refers to the overwritten virtual function (called a *thunk*). Preceding the function entries, a vtable has two metadata fields: *Runtime Type Identification* (RTTI) and *Offset-to-Top*.

RTTI holds a pointer to type information about the class. Among other things, this type information contains the name of the class and its base classes. However, RTTI is optional and often omitted by the compiler. It is only needed when the programmer uses, e.g., *dynamic_cast* or *type_info*. Hence, a reliable static analysis cannot rely on this information. Classes that do not contain RTTI have the RTTI field set to zero.

Offset-to-Top is needed when a class uses multiple inheritance (hence has a base vtable and one or more sub-vtables) as class C does. Offset-to-Top specifies the distance between a sub-vtable's own *vtblptr* and the base *vtblptr* at the beginning of the object. In our example, the *vtblptr* to class C's sub-vtable resides at offset `0x10` in the object, while the *vtblptr* to the base vtable resides at offset `0x0`. Hence, the distance between the two, as stored in the Offset-to-Top field in sub-vtable C, is `-0x10`. Offset-to-Top is `0` if the vtable is the base vtable of the class or no multiple inheritance is used.

In cases of virtual inheritance, an advanced C++ feature for classes that inherit from the same base multiple times (diamond-shaped inheritance), vtables can contain one additional metadata field, called *Virtual-Base-Offset*. However, virtual inheritance is rarely used in practice.

2.2.3 C++ Object Initialization and Destruction

Constructors are used for initializing the memory area previously allocated to hold a specific object. Since our analysis approaches focus on objects containing virtual functions, we only consider object initialization and destruction of classes having a vtable.

During object instantiation, the *vtblptr* is written into the object by the *constructor*. Part b) of Figure 2.3 depicts an object's memory layout at the binary level. The *vtblptr* is at offset `0x0`, the start of the object. For classes with multiple inheritance, the constructor also initializes *vtblptr*s to the sub-vtable(s). In addition, the programmer may initialize class-specific fields in the constructor. These fields are located after the *vtblptr* and, in case of multiple inheritance, after any sub-*vtblptr*s.

For classes that have one or more base classes, the constructors of the base classes are called before the derived class's own initialization code (*top-down*). As a result, the base class places its *vtblptr* into the object, which is subsequently overwritten by the derived class's *vtblptr*. Depending on the optimization level, constructors are often inlined, which may complicate binary analysis that aims to detect constructors.

An analogous principle is applied for object destruction through *destructor* functions. However, the destructors are executed in reversed order (*bottom-up*, destructor of the base class is executed last).

Abstract classes form a special case: although programmers cannot instantiate abstract classes, and despite the fact that their vtables contain *pure_virtual* function entries, the

compiler can still emit code that writes the *vtblptr* to an abstract class into an object. However, this happens only when creating or releasing an object of a derived class, and the abstract *vtblptr* is immediately overwritten.

2.2.4 Virtual Function Dispatch

As opposed to regular functions (which are implemented using direct calls), virtual function calls require a specific type of callsite (*virtual callsite*, or *vcall*). They handle the selection of the proper virtual function depending on the object on which the function is invoked using the object's vtable.

Consider a virtual callsite invoking `funcA2` on an object of either class `A` or `C`. Independent of the class the object at the callsite is instantiated from, in this case, one merely has to call whatever function is referenced at offset `0x08` in the object's vtable. As seen in Figure 2.3 b), this offset either points to `A::funcA2` or `C::funcA2` and always calls the correct implementation for the given object. Note that this offset has to be the same across all related vtables. In this case, this constraint applies for vtable `A` and `C`, as classes `A` and `C` are the only candidates when invoking function `funcA2`. This mechanism effectively implements polymorphism at the binary level.

The compiler emits code that directly implements this mechanism. At each vcall, the *thisptr* to the object is also set as an implicit argument (meaning the argument is not specifically set in the source code). Depending on the calling convention, the *thisptr* is either stored in a specific register or on the stack. In the Itanium C++ ABI on x86-64, a vcall always has the following structure:

```
mov RDI, thisptr
call [vtblptr + offset]
```

The *thisptr* is stored in the `RDI` register as the first argument and the *vtblptr* is used to select the correct virtual table. The value `offset` denotes the offset into the selected vtable in order to branch to the correct function. Note that it may be zero or omitted if the first function of the vtable is targeted.

The same code structure holds for cases that use multiple inheritance. Given that the dispatching mechanism targets certain offsets in the vtable, the order of the functions must be preserved throughout the hierarchy. In the given example, class `C` has a copy of the vtable of class `A` with a modified entry (pointer to function `C::funcA2` instead of `A::funcA2`) and appends pointers to its own implementations of virtual functions to it. Further, a modified copy of the vtable of class `B` is added as a *sub-vtable* of class `C`. In this sub-vtable, only the function entries of *overridden* virtual functions have changed. These entries have been replaced by a pointer to special functions called *thunks*. Note that if a class derives from only one base class, both vtables can be merged without conflicts and no sub-vtable is necessary – the current class simply uses higher offsets when accessing its part of the vtable.

When accessing an object of class `C` as an instance of class `B`, the layout has to match the one expected by vcalls of class `B`. Hence, the *thisptr* has to be increased by `0x10` to

point to the position the *vtblptr* to the sub-vtable is stored (offset `0x10`). Thunks are used to call back into virtual functions belonging to class `C`. In the given example, `funcB1` was overwritten by class `C`. At the position of the entry of *funcB1* in the sub-vtable, the pointer to the thunk, `thunk to C::funcB1`, has been written. When executing this thunk, the *thisptr* is modified to point to the beginning of the object `C` and invokes `funcB1` of class `C`. This ensures that the function uses the correct offsets into the object, i.e., the *thisptr* points to the start of object `C`. Because the code structure of the vcall is the same, the program treats calls through sub-vtables and base vtables as analogous.

Chapter 3

Automated Identification of Cryptographic Functions in Binaries

Pseudo-random number generators (PRNGs) and cryptographic hash functions (CHFs) play a fundamental role in the broader security ecosystem. PRNGs provide the basis for various security mechanisms, such as exploit mitigations (e.g., ASLR [122] and stack cookies [42]) and encryption key generation. Security-sensitive applications such as signature generation or integrity verification use CHFs. Interestingly, both PRNGs and CHFs are closely related types of functions—to such a degree that CHFs represent the basis for some PRNGs in their implementation [87, 106]. In practice, some security-sensitive libraries or applications (e.g., OpenSSL or OpenSSH) assume the existence of a cryptographically secure PRNG in the operating system (OS) [3]. Due to the security-critical nature of PRNGs and CHFs, a weaknesses in their implementation can have severe consequences for security platforms such as passively decrypting VPN traffic [36], impersonating TLS secured servers [161], or forging Windows updates to deliver malware [57]. Therefore, it is essential to evaluate the security of PRNG and CHF implementations on different platforms.

The first step to assess the security of these algorithms is to find the corresponding functions within the target application—a straightforward task for open-source software. However, for proprietary, closed-source software, finding these functions is a crucial hurdle to the overall analysis success.

Existing research tried to find cryptographic algorithms in binaries by either using dynamic analysis [73, 101, 160] or relying on static signatures [16, 99]. However, both approaches have crucial shortcomings. Dynamic analysis struggles with code coverage and requires specific inputs that trigger the usage of the encryption algorithm, which is a problem, especially for closed-source applications. Whereas static signatures cannot identify modified or future variants of algorithms. Moreover, to the best of our knowledge, no previous work has focused on identifying PRNGs or CHFs in binary executables in a generic way.

3.1 Introduction

In this chapter, we present the design of an analysis approach to precisely identify PRNG and CHF implementations in a given binary executable. Unlike previous related work, our novel analysis approach works generically without signatures or relying on magic constants and inputs provided by an analyst. In a first step, we investigated 24 PRNG and 61 CHF implementations in popular cryptographic shared libraries. We found similarities and based on these insights we built a generic model encompassing the interactions of both types of functions with other components of the application and the memory. The model allows us to abstract away the reliance on concrete implementation details. Therefore, we regard the corresponding implementation of the PRNG and CHF as blackbox and only consider their interactions with the rest of the program.

Our approach assumes that PRNGs and CHFs create data with a certain level of randomness. For this, we run the function suspected to correspond to a PRNG or CHF multiple times to see if the generated data looks random. Our approach emulates the target function directly, to sidestep coverage problems that related works using dynamic analysis [73, 101, 160] suffer from. Being able to emulate a function directly allows us to analyze applications and shared libraries a-like, without the need for test cases that drive the execution to the target function. However, emulating a function without prior knowledge about the expected memory layout comes with its own set of challenges. To tackle these, we introduce techniques built on PRNG and CHF domain knowledge. We automatically infer an initial memory layout for the function we want to emulate based on the manually investigated implementations. Additionally, we use coverage-guided fuzzing to refine the memory layout and fully emulate the function without crashes.

Based on our proposed design, we implement a prototype named *TropyHunter* to find PRNGs and CHFs in binary programs in a fully automated way. We evaluate *TropyHunter* against six binaries of popular open-source cryptographic libraries and six different open-source application binaries. Since a PRNG or CHF algorithm is implemented with the help of multiple functions, we distinguish in our evaluation between *identified algorithms* and *identified functions*. Our evaluation shows the accuracy of *TropyHunter*'s analysis: for example, in the case of *PostgreSQL*, a database server containing over 12,000 functions, *TropyHunter* identified 11 functions as being either a CHF or PRNG. A comparison against the ground truth available in the source code revealed that indeed all the pinpointed functions are part of a CHF or PRNG implementation. Overall, *TropyHunter* identified 72 out of 81 CHFs and 24 out of 33 PRNGs in our open-source evaluation set in a fully automated way. Furthermore, we evaluated *TropyHunter* against different closed-source applications. The analysis could again identify various CHF and PRNG functions in these closed-source binaries. For example, in the case of the authentication module for QNX, a POSIX-compatible non-Linux OS, *TropyHunter* identified six CHF and PRNG functions (the binary has a total of 194 functions). A manual verification confirmed the identified functions indeed represent the only CHF and PRNG functions in this binary executable.

Contributions In summary, this chapter makes the following three contributions:

- We develop a generic model of the interactions of PRNGs and CHFs based on the investigation of 24 PRNG and 61 CHF implementations in popular cryptographic shared libraries.
- We present the design and implementation of *TropyHunter*, a generic approach to pinpoint PRNGs and CHFs in a given binary executable. Furthermore, we develop techniques to emulate functions without prior knowledge about their expected memory layout.
- We evaluate *TropyHunter* against various open- and closed-source binaries and demonstrate its effectiveness in identifying PRNG and CHF algorithms, even in large, complex applications.

To foster research on this topic, we released the code for *TropyHunter* under `https://github.com/RUB-SysSec/tropyhunter`.

Outline This chapter is structured in the following way: Section 3.2 gives an overview of PRNG and CHF implementations and introduces our generic model. Section 3.3 describes our analysis approach to find PRNG and CHF implementations in binary executables. The implementation of *TropyHunter* is explained in Section 3.4, whereas the evaluation is presented in Section 3.5. We then discuss limitations of *TropyHunter* in Section 3.6 and give an overview of related work in Section 3.7. Finally, we conclude this chapter in Section 3.8 and discuss possible topics for future advancements.

The chapter contains new and unpublished materials which is still in submission. The research was performed together with Philipp Görtz, Paul Gerste, Ali Abbasi, and Thorsten Holz.

3.2 Overview

In this section, we describe the inner workings of PRNGs and CHFs to give an overview and provide the necessary background information to understand the rest of this chapter. To create a generic model of their interaction, we manually investigated various implementations and analyzed how they interact with other components within the application such as the memory and their caller functions. Based on this generic model, we create assumptions that we use to identify PRNGs and CHFs within the application binaries.

3.2.1 Pseudo-Random Number Generator

The purpose of random number generators is to generate seemingly unpredictable data. The randomness of data is critical for the security ecosystem, and its absence can undermine the system's security. Random data is usually created by pseudo-random number generators (PRNGs). PRNGs are software-based implementations of a deterministic algorithm which stretches an initial input seed into a *arbitrary sized* sequence of random-looking bytes. Seemingly random events usually generate these initial input seeds (e.g.,

disk activity, keystroke timings, mouse movement, interrupt requests, etc.) and are aggregated to initialize the PRNG. Usually, PRNGs use initial seeds as a first state for their deterministic algorithm; this algorithm then generates random-looking data. To continually generate data, PRNGs need to refresh their internal state regularly. This refreshing is usually done by adding newly generated random data to it, possibly alongside other sources of entropy. The OS and applications can then use the returned random data. Examples of PRNG functions are Yarrow [87], Fortuna [106], ChaCha [18], and Mersenne Twister [105].

Many operating systems provide a cryptographically secure PRNG which is used by critical security services. For example, Linux offers this functionality in the form of `/dev/urandom`, and Microsoft Windows provides similar functionality in their *CryptoAPI*. It is worth mentioning that many applications and libraries use their own implementations of PRNGs instead of using the ones provided by the OS. Examples for such applications are *PHP*, *Ruby*, *libgcryp*, and *libsodium*.

There are also random number generators that derive randomness from physical phenomena such as radioactive decays, thermal noise, and other sources of randomness. Such generators are called true random number generators (TRNGs) and are physical devices. In modern computers, Trusted Platform Modules (TPMs) or Hardware Security Modules (HSMs) implement TRNGs (or a combination of a TRNG with a PRNG). Additionally, Intel started to provide a hardware-based RNG (actually a combination of a TRNG with a PRNG) with Intel IvyBridge CPUs in 2012 [61]. However, hardware modules are expensive, are not always available, and may also contain bugs [98]. As a result, PRNGs are often used in practice and are the focus of our analysis.

3.2.1.1 Inner Workings of PRNGs

Although all PRNGs generate random data, how the algorithms work and how they are implemented differs. Even PRNGs which use the same algorithm might differ in their actual implementation. Additionally, in the future, there might be new PRNG algorithms which further underlines the importance of having a generic model to identify PRNGs. Hence, we cannot rely on specific implementation artifacts or magic constants to identify PRNGs generically. To find a common denominator, we investigated 24 different implementations of PRNGs in popular cryptographic libraries to find similarities and general features of such algorithms. Based on this analysis, we found that the interactions of the PRNGs remain mostly the same. As illustrated in Table 3.1, we analyzed how a PRNG function interacts internally and externally. To be more precise, we investigate how the PRNG keeps its internal state, how many arguments are passed to the PRNG, what kind of data is returned, how the PRNG is being initialized before being used, and what the output size is.

Based on the information we summarized in Table 3.1, we created a generic model of the inner workings of PRNG implementations, which Figure 3.1 illustrates. Before an application can obtain random data from a PRNG implementation, the PRNG has to be initialized (if it is not initialized already). The initialization process prepares the memory where the PRNG state is stored. It fills this memory with the initial input seed, as well as other data needed by the implementation (e.g., configuration data used by the PRNG

Table 3.1: The features extracted by our investigation of all 24 PRNG function implementations in popular cryptographic shared libraries. The functions selected for this excerpt cover each option of the features. For each function the table shows seven different features of the PRNG interaction. The *Memory State* describes the memory location where the PRNG stores internal data. The location is either given via a pointer, passed as an argument, a static global memory location, or none is needed (e.g., if `/dev/urandom` is used). *Has Init Flag* shows whether the PRNG has an initialization flag. Based on our study, we find that the majority of PRNGs have an internal flag indicating whether the PRNG state is initialized or not. *Init Behavior* describes how the PRNG implementation reacts when an expected initialization flag is not set. The implementations either automatically initialize the memory state, abort the random number generation or fail without raising an error. *Output Size via Argument*, describes whether the PRNG takes an argument which denotes the requested number of random bytes to generate. *Return Data via Argument Pointer* describes if the PRNG function takes an argument holding the destination location for the generated random bytes. *Small Integers in State* depicts if the memory state expects small integer values in some fields. *32 Bit Output* describes whether the output of the PRNG is fixed to 32 bit.

Program	Function Name	Memory State	Has Init Flag	Init Behavior
libnettle	nettle_yarrow256_random	Pointer by Argument	✓	Abort
libnettle	nettle_knuth_lfib_random	Pointer by Argument	✗	No Error
libcrypto	drbg_ctr_generate	Pointer by Argument	✗	No Error
libcrypto	drbg_hash_generate	Pointer by Argument	✗	No Error
libcrypto	drbg_hmac_generate	Pointer by Argument	✗	No Error
libcrypto	OPENSSL_ia32_rdrand_bytes	None	✗	Not Needed
libgcrypt	mix_pool	Pointer by Argument	✗	No Error
libsodium	randombytes_sysrandom_buf	Global Memory	✓	Auto Init
libsodium	randombytes_sysrandom	Global Memory	✓	Auto Init
libtomcrypt	sober128_read	Pointer by Argument	✓	Abort
libtomcrypt	yarrow_read	Pointer by Argument	✓	Abort
libwolfssl	Hash_DRBG_Generate	Pointer by Argument	✗	No Error

Program	Function Name	Output Size via Argument	Return Data via Argument Pointer	Small Integers in State	32 Bit Output
libnettle	nettle_yarrow256_random	✓	✓	✗	✗
libnettle	nettle_knuth_lfib_random	✓	✓	✓	✗
libcrypto	drbg_ctr_generate	✓[1]	✓	✗	✗
libcrypto	drbg_hash_generate	✓[1]	✓	✗	✗
libcrypto	drbg_hmac_generate	✓[1]	✓	✗	✗
libcrypto	OPENSSL_ia32_rdrand_bytes	✓	✓	✗	✗
libgcrypt	mix_pool	✗	✓	✗	✗
libsodium	randombytes_sysrandom_buf	✓	✓	✗	✗
libsodium	randombytes_sysrandom	✗	✗	✗	✓
libtomcrypt	sober128_read	✓	✓	✗	✗
libtomcrypt	yarrow_read	✓	✓	✓	✗
libwolfssl	Hash_DRBG_Generate	✓	✓	✗	✗

[1] *It depicts that next to the size argument another non-pointer argument exists.*

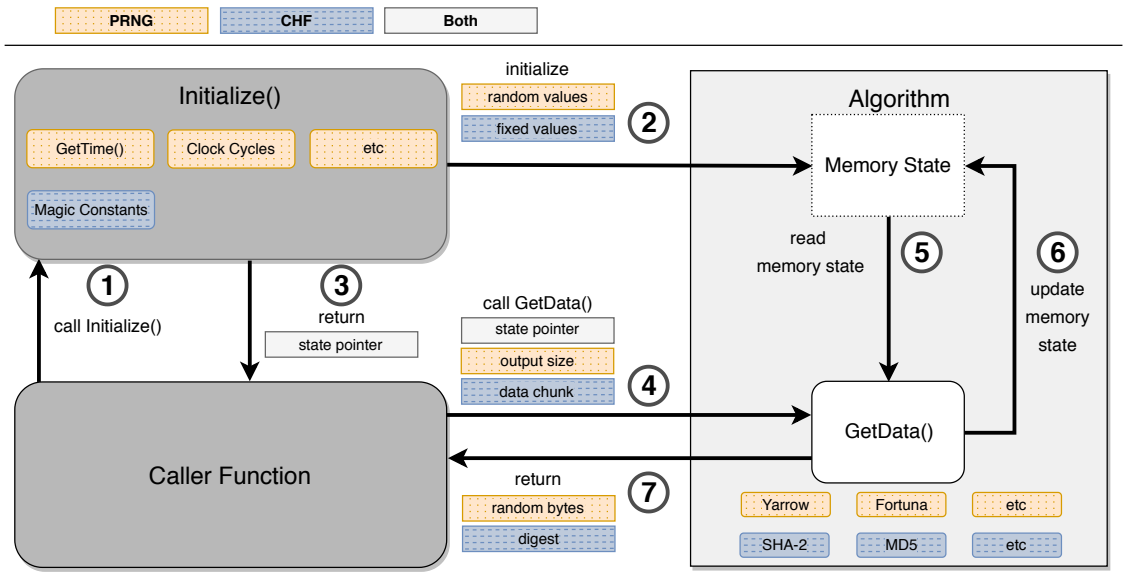


Figure 3.1: Our interaction model of PRNGs and CHFs containing the communication with the caller function and the interaction of the algorithms with the memory. The dotted boxes show components that are only used by a PRNG. The dashed boxes depict parts that are only used by a CHF.

implementation). We call this memory the *PRNG memory state*. The initial input seed is aggregated using different sources (e.g., time, or processor clock cycles). The PRNG initialization returns a pointer to the PRNG memory state to the caller function as shown in steps 1 to 3 of Figure 3.1. Note that these three steps are only required if the PRNG is not already initialized. Next, the caller function executes the PRNG algorithm by providing the pointer to the PRNG memory state (as well as other required arguments), as illustrated in step 4. Step 5 shows the deterministic PRNG algorithm using the PRNG memory state to generate random bytes. Additionally, in step 6, the PRNG memory state is refreshed with new random data. Finally, in step 7, the generated data is returned to the caller function. Note that depending on the implementation of the PRNG, the actual interactions between the caller function of the application and the PRNG can differ. However, the basic principle remains the same: the deterministic PRNG algorithms work on a memory state to generate random data.

3.2.2 Cryptographic Hash Function

Hash functions are implementations of a deterministic algorithm which usually maps arbitrary sized input data to a *fixed sized* sequence of bytes. The hash algorithm always yields the same output data for the same input. Since not all hash functions are cryptographically secure, not all of them are suitable for security-sensitive operations. For such operations, one can use cryptographic secure hash functions (CHFs) such as MD5, SHA-2, or SHA-3. CHFs have additional properties like collision resistance or being one-way functions [117].

Table 3.2: The features extracted by our investigation of all 61 CHF implementations in popular cryptographic shared libraries. The functions selected for this excerpt cover each option of the features. For each function the table shows four features of CHF interactions. The *Memory State* describes the memory location used by the CHF to store internal data. Within the 61 observed CHF implementations, the location is always given via a pointer which is passed as an argument. *Input Size via Argument* describes whether the CHF takes an argument denoting the size of the input data. *Fixed Output Size* depicts if the output size is fixed or dynamic. Finally, *Output Data* describes how the calculated data is returned to the caller function. Depending on the component of the CHF, the data is either integrated into the memory state because the hash calculation is not yet finalized or the final digest is placed into a memory location given as pointer by an argument.

Program	**Function Name**	**Memory State**	**Input Size via Argument**
libnettle	_nettle_ripemd160_compress	Pointer by Argument	✗
libnettle	gost_block_compress	Pointer by Argument	✗
libcrypto	SHA512_Final	Pointer by Argument	✗
libcrypto	WHIRLPOOL_Final	Pointer by Argument	✗
libcrypto	blake2b_compress	Pointer by Argument	✓
libgcrypt	whirlpool_final	Pointer by Argument	✗
libgcrypt	blake2s_transform	Pointer by Argument	✓
libtomcrypt	rmd320_done	Pointer by Argument	✗
libtomcrypt	sha3_shake_done	Pointer by Argument	✗
libwolfssl	Transform_Sha256	Pointer by Argument	✗
libsodium	SHA512_Transform	Pointer by Argument	✗
libsodium	blake2b_compress_ref	Pointer by Argument	✗

Program	**Function Name**	**Fixed Output Size**	**Output Data**
libnettle	_nettle_ripemd160_compress	✓	Integrated in State
libnettle	gost_block_compress	✓	Integrated in State
libcrypto	SHA512_Final	✓	Explicit Output Pointer
libcrypto	WHIRLPOOL_Final	✓	Explicit Output Pointer
libcrypto	blake2b_compress	✓	Integrated in State
libgcrypt	whirlpool_final	✓	Integrated in State
libgcrypt	blake2s_transform	✓	Integrated in State
libtomcrypt	rmd320_done	✓	Explicit Output Pointer
libtomcrypt	sha3_shake_done	✗	Explicit Output Pointer
libwolfssl	Transform_Sha256	✓	Integrated in State
libsodium	SHA512_Transform	✓	Integrated in State
libsodium	blake2b_compress_ref	✓	Integrated in State

3.2.2.1 Inner Workings of Cryptographic Hash Functions

On a high-level, CHFs are similar to PRNGs. Therefore, we cannot rely on algorithm implementation specifics (e.g., magic constants) to find CHFs generically. As we did for PRNGs, we investigated 61 different implementations of CHFs in popular cryptographic shared libraries to find similarities. We found that the internal and external interactions of the CHFs also remain mostly the same. As illustrated in Table 3.2, we analyzed how a CHF interacts with the rest of the application. To be more precise, we investigated how the CHF keeps its internal state, whether the input size is passed via function arguments, if the output size is fixed or not, and how the output data is returned. Note that the table shows two different components of CHF implementations because a CHF usually has two components: one that adds new data to the hash calculation and a second that finalizes it. The former does not return a hash value but instead places the generated data into its internal memory state, while the latter returns the calculated hash digest.

Since the PRNG and CHF models have significant overlaps, we integrated both models in Figure 3.1. The CHF has to be initialized, as illustrated in steps 1 to 3. The only difference to PRNGs is that the initialization process uses well-known constants ("magic constants") instead of random data. As depicted in step 4, when a routine calls the CHF to generate output, it provides a pointer to the hash memory state. Using the provided memory state, the hash algorithm function (e.g., MD5 or SHA-3) generates the output data and updates the hash memory state, as depicted in steps 5 and 6. Finally, the output data is returned to the caller function, as shown in step 7. We can see the similarity between PRNGs and CHFs when viewed in the form of our model. Similar to PRNG algorithm functions, the interactions between the caller function of the application and the CHFs can differ depending on the implementation—however, the underlying principle of the algorithm working on a memory state to generate a hash value is always the same.

3.2.3 Assumptions

PRNGs have specific characteristics which can help us to identify them in a given binary executable. As the purpose of a PRNG is to generate random data, identifying procedures that generate different data each time they are executed is a straightforward way to find PRNG algorithm functions. However, this approach is too broad: any function would be classified as a PRNG function if it returns different data when called multiple times (e.g., `time`, `malloc`, etc.).

To improve this naïve approach, we assume that PRNG functions create data with a higher quality of randomness than other functions. Based on this assumption, we can assess the generated randomness and filter out functions that do not meet a certain threshold of randomness (such as `malloc`). To verify whether a function generates enough randomness, we can use random number generator tests such as *Birthday Spacings Test* or *Parking Lot Test* [24].

A similar approach can be applied to find CHFs in binaries. Since CHFs update their memory state each time they are executed as depicted in Figure 3.1, they also create random-looking data each time. Furthermore, CHFs have the property that a small change in the input value changes the output data extensively to prevent correlation between

similar input values (called avalanche effect [117]). As a result, the output also passes the random number generator test and, thus, CHFs can also be identified by the randomness of the created data.

3.3 Approach

We base our approach on the observation that executing a PRNG function multiple times yields different random data each time. A deterministic PRNG algorithm achieves this by holding a state which is updated after each usage as illustrated in steps 5 and 6 of Figure 3.1. Hence, we execute a possible PRNG function multiple times without re-initializing the memory in between and check the output for sufficient randomness. Because the PRNG algorithm updates its memory state itself each time it is executed (thus re-initializing the memory would revert this update), the output differs after each execution. As the hash algorithm also updates its internal state each time it is used (also depicted in steps 5 and 6 of Figure 3.1), the same holds for CHFs. Executing the CHF multiple times without re-initializing the memory in between also results in seemingly random output. As CHFs also create vastly different-looking output with each change, the randomness of it is sufficient to pass random number generator tests. Note that this approach targets the algorithm of the PRNG and CHF itself, and ignores the initialization routine. Hence, in the following, we refer to *algorithm function* when speaking about the algorithm for both PRNGs and CHFs.

First, our analysis uses heuristics to narrow down the set of candidate algorithm functions. On these candidate functions, a more thorough analysis is performed by executing them multiple times. However, to execute a specific candidate function, we have to provide the application with proper input to reach it. Since our analysis targets binary applications, having proper input to reach all functions in the application that we want to analyze is an unrealistic requirement. For binary shared libraries, this requirement becomes even more unrealistic, since in addition to the needed input the analyst has to provide code that uses the functions of the library. To avoid having to fulfill these requirements, our approach emulates the functions directly instead of executing them. To this end, we identify the arguments the function uses before it is emulated. Next, the function is emulated multiple times to generate output data. However, emulating unknown functions without knowledge about them (e.g., expected memory layout) is a challenging problem in itself. Thus, based on the features shown in Table 3.1 for PRNG and Table 3.2 for CHF implementations, we introduce a technique that allows us to infer concrete input data without prior knowledge of memory layout or content. Finally, we use randomness tests on the generated output data to confirm an algorithm function. If we have indeed found an algorithm function, we try to classify it as a PRNG or CHF to give the analyst an additional hint to base their work on.

Our approach consists of five steps: (A) *Function Preselection*, (B) *Argument Inference*, (C) *Emulation*, (D) *Check for Randomness*, and (E) *Classification*. In the following, we describe each step in detail.

3.3.1 Function Preselection

To reduce the number of functions to analyze in-depth and hence improve the performance of our approach, we first search for candidate functions with specific characteristics. We preselect the functions by using the following three heuristics that aim to identify PRNG and CHF implementations:

Instruction Heuristic Various hardware architectures provide different instructions for cryptographic purposes. For example, the x86-64 instruction set provides `rdrand` to generate a random number or `aesenc` to perform an AES encryption round [80]. Similarly, ARM AArch64 offers instructions to perform various cryptographic operations. For example, AArch64 provides `aese` or `sha256h` to perform AES or SHA-256 operations [13]. Therefore in this heuristic, we mark all functions containing at least one cryptographic instruction as candidate functions.

API Heuristic Operating systems usually offer an API for cryptographic purposes. For example, Linux provides a system call to create random numbers (`sys_getrandom`), Windows provides the `CryptGenRandom` function in their CryptoAPI to provide cryptographically random bytes, and Linux-like operating systems provide a cryptographically secure PRNG via block devices (e.g., `/dev/urandom`). Functions using such APIs (e.g., reading from `/dev/urandom` or using the *sys_getrandom* syscall) are marked as viable candidates.

Arithmetic Heuristic This heuristic follows the insight of work proposed in the literature [73, 101, 159] that cryptographic functions contain considerably more arithmetic and bitwise instructions (e.g., `mul` or `xor`). Hence, a function having basic blocks with a high ratio of arithmetic instructions (exceeding a specific threshold described in Section 3.4) are considered candidates.

Functions marked by these three heuristics are candidates for cryptographic operations. However, some PRNGs and CHFs separate their implementations into multiple functions (e.g., SHA-512 in *libcrypto*) or rely on other algorithms to generate data (e.g., the Yarrow PRNG algorithm uses a block cipher such as AES). Hence, we also consider the caller functions of each function detected by a heuristic as a candidate function for a more thorough analysis.

3.3.2 Argument and Type Inference

Once we have obtained a list of candidate algorithm functions, we try to emulate these. However, since our approach emulates a candidate function directly, we need to know the expected arguments of the function. More specifically, we need to know the number of arguments that are used, as well as their type. In the context of this work, we only need to consider two types of function arguments: pointer (e.g., a register pointing to an object in memory) and value (e.g., the number of output bytes). To identify the number of arguments and their types, we track the data flow from each function argument with the help of the Static Single Assignment (SSA) form [45]. Depending on the usage of the arguments, we can infer their type (and if the argument is used at all). If an argument

is used as a value by an instruction (e.g., as an argument to an arithmetic operation), we consider its type to be a *value*. Analogously, if the argument is used in a memory context (e.g., dereferencing), we consider its type to be a *pointer*. Alternatively, we can identify a variable type if a function uses it with a known prototype. For example, if a candidate function argument is passed as an argument for the *libc* function `memset`, we can infer the type from the known prototype of the *libc* function [96, 102]. Note that this is not limited to the *libc*, but works for other known libraries or APIs (e.g., the Windows API) as well.

While this approach works for most cases, we have to be aware of some edge cases. For example, if the instruction `cmp rdi, 8` is in the data flow, we can safely assume the argument to be of type *value*. In contrast, the instruction `cmp rdi, 0` does not provide a conclusive result as that instruction is also used to check for null pointers. Therefore, we cannot distinguish between *value* or *pointer* as a possible argument type here. Other examples are optimization tricks performed by the compiler, e.g., the instruction `lea` is often used to perform an addition of two values, despite its actual purpose of working with pointers. Hence, we cannot safely infer the type of the argument in these edge cases. As our analysis is done in an intra-procedural way, determining the type may fail. In these cases, we mark the argument registers with the special type *unknown* for the subsequent analysis steps and proceed.

3.3.3 Emulation

Knowing the inferred argument register types, we are nearly able to start the emulation of the target function. However, to emulate a function with an argument of type *pointer*, we need to create a valid memory object. For example, a pointer argument may point to a data structure consisting of multiple data fields. To properly emulate this function, we have to provide valid-looking data in the form of the expected structure. Otherwise, the function might crash. Similarly, for function arguments of type *value*, we also need to generate valid input data. Once we have found valid input data, we emulate the function multiple times. The emulation creates the output data we need to determine if we have found an algorithm function. Figure 3.2 depicts a flowchart of the whole emulation process.

3.3.3.1 Input Generation

Since our goal is to emulate algorithm functions, we can leverage domain knowledge of typical PRNG and CHF implementations to craft appropriate inputs. In the case of PRNGs, an argument of type *value* refers to the number of bytes which the algorithm has to generate as depicted in Table 3.1 in the column *Output Size via Argument*. Note that only three exceptions were found that had an argument of type *value* in addition to the size argument. For CHFs, the argument of type *value* holds the number of bytes the algorithm has to read (as shown in Table 3.2 in the column *Input Size via Argument*). Hence, it is sufficient just to place an integer value into these arguments.

Arguments of type *pointer* are a pointer to the algorithm memory state, a pointer to the destination to which the algorithm should write its data, or a pointer to the source from which data is read. Hence, if an argument has the inferred type *pointer*, we allo-

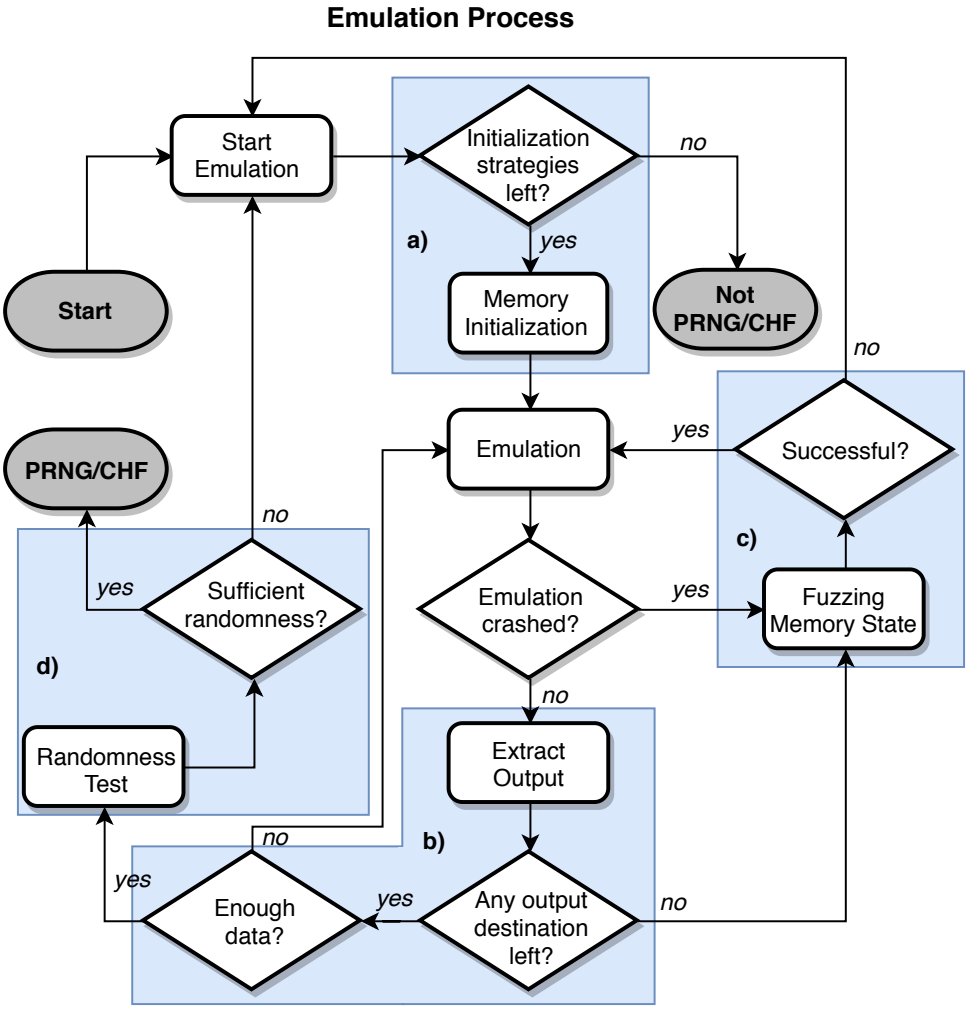


Figure 3.2: Flowchart depicting the different parts of the emulation process to determine if the candidate function is a PRNG/CHF or not.

cate memory and store the address of it in the argument. We discuss our strategies for initializing the allocated memory in the following section.

For the emulation of a function with an argument marked as *unknown*, we try both types: *value* and *pointer*. If one type does not result in a successful emulation, we retry with the other type.

3.3.3.2 Memory Initialization Strategies

Input generation for arguments of type *pointer* is more complex compared to *value* arguments. This complexity is caused by the fact that we cannot know beforehand how the function expects the initialized memory to look like (e.g., does the memory object have certain fields that must be set?). We try different memory initialization strategies to successfully emulate the function and generate random output data. If the emulation is unsuccessful (i.e., it crashes), the memory initialization strategy is changed and the process restarts. Note that we do not need to know the exact purpose of the *pointer* argument (a pointer to the memory state, a pointer to input data, or a pointer to the output destination) for our memory initialization strategies to work. We just assume each of them as a potential pointer to the algorithm memory state. If we initialize the memory of an output destination, it just gets overwritten during the emulation of the candidate function and does not affect our analysis. If we initialize the memory of a pointer to input data, the algorithm just uses the provided data for its calculation and does not affect our base assumption about the algorithm. Part a) in Figure 3.2 depicts the memory initialization process.

For the emulation, we use the following input memory initialization strategies:

- *Zero* initializes the allocated memory with zero. This basic strategy is sufficient for the majority of the CHFs we encountered. Furthermore, we found PRNG implementations having an `is_initialized` flag in their state. If this flag is set to zero (meaning that it is not initialized), some PRNG algorithms automatically initialize the PRNG memory state, as shown in Table 3.1. This behavior helps us to properly emulate the actual algorithm without any further search for the correct memory layout. Thus, we test this strategy first.

- *Random* initializes the allocated memory with random bytes. Some PRNG implementations which use the `is_initialized` flag do not automatically initialize the memory state. Instead, they go to an error-handling routine or fall short of reporting any error at all. In such situations the *Zero* strategy fails, thus filling the allocated memory with random bytes may lead to a successful emulation.

- *Random8Plus* initializes the allocated memory with 8 bytes aligned positive random integers which are equal or smaller than 255. The reasoning behind this strategy is that we encountered PRNG implementations that for example store the number of AES rounds (which at maximum is 14) in their PRNG memory state. When filling this field with a large number, the emulation of the algorithm function would take up too much time to finish. Hence, filling the memory state with small positive random integers circumvent this problem. Table 3.1 depicts in the column *Small Integers*

in State PRNG implementations requiring small integer values in certain memory state fields.

- *Random4Plus* initializing the allocated memory with small positive random integers 4 bytes aligned. This strategy follows the same reasoning as the *Random8Plus* strategy, however, it focuses on fields of the memory state that are 4 bytes in size. We encountered PRNG implementations which even though they were compiled for a 64 bit architecture, only used 32 bit fields.

- *RandomPlus* initializes the allocated memory with positive random integers. We do this by setting the most significant bit of each byte to zero (so it will be a positive integer). Note that in this strategy the integer value can be a large number. The reasoning behind this strategy is that there might be a specific memory state field which has a size less than 4 bytes (e.g., 1 byte). When we use either *Random8Plus* or *Random4Plus* as strategy, the PRNG algorithm function might read the most significant bytes. In such cases, the value read is zero, and thus, the algorithm might not run. To avoid such cases, we use this strategy. Note that this strategy also addresses the shortcomings of the *Random* strategy as it handles cases where a field only expects a positive integer. The *Random* strategy might fill the field with a negative integer and thus fail a check.

During our research, we did not encounter an implementation that requires corresponding strategies for negative values, namely *Random8Minus*, *Random4Minus*, and *RandomMinus*. Hence, we omit those to have a better performance during the analysis process. In Section 3.5.3, we will show with an ablation study that each of the presented strategies contributes to the effectiveness of our approach.

Once we have achieved a successful emulation, subsequent emulations use the same input. Using the same input means that the memory layout is preserved between the emulations and we do *not* re-initialize it. This is an important aspect of our analysis, as our assumption is that the algorithm function updates its memory state each time it is executed to create different-looking output data. Re-initializing the memory layout with our memory initialization strategies would destroy the update made by the algorithm function. We repeat the emulation until we extracted sufficient output data for our randomness tests. We further discuss extracting sufficient output data in Section 3.3.3.3. It is worth mentioning that during the emulation the candidate function may try to access unallocated memory locations. To let the emulation continue, we catch these accesses and allocate the requested memory on-the-fly.

If a strategy does not work, we first try a fine-grained modification of the memory layout to get a successful emulation as we describe in detail in Section 3.3.3.4. If the memory layout modification does not succeed either, we restart the emulation process with the next memory initialization strategy. Finally, when no strategy is left (and we still do not have random output), we discard the candidate function.

3.3.3.3 Output Extraction

To gather data for the randomness tests, we have to extract the output data from the emulator after each successful emulation. To extract the data, we can again leverage domain knowledge of typical PRNG and CHF algorithms to pinpoint possible output destinations. Based on the analyzed PRNG and CHF implementations (see Table 3.1 and Table 3.2), the output data is either returned via a register (e.g., for a randomly generated integer) or via memory pointer (e.g., in case of randomly generated data of arbitrary length or CHF results). Therefore, we only need to consider return registers and the memory location function arguments of type *pointer* point to as output destinations. We store the generated data and start the next emulation, just as PRNGs are called multiple times to create more output data.

We identify output destinations that contain the same data for multiple emulations. This identification allows us to improve performance as we can dismiss these output destinations after a threshold (see Section 3.4 for implementation details). If we dismissed all output destinations, the emulation is either unsuccessful (because the provided memory layout is wrong) or it is not an algorithm function. Since we cannot differentiate between both cases, we continue to search for a working memory layout for the emulation process. Part b) of Figure 3.2 depicts the output extraction process.

3.3.3.4 Fuzzing Memory State

Even though we have different memory initialization strategies, they can potentially be too imprecise to guarantee a successful emulation (see our evaluation of our proposed memory strategies in Section 3.5.3). For example, consider the memory structure for the PRNG implementation in Figure 3.3. The *memory_state* struct consists of three fields, as shown in lines 1 to 5. These fields vary widely in their expected content, making it unlikely that our memory initialization strategies yield a successful emulation as they apply to the whole memory state. Hence, the function *get_random* is likely to fail to generate random data when only resorting to our memory initialization strategies. For example, if we use the *random* strategy, our chance to get 0 in the *error_code* field is low; causing the function to abort (problem 1). If we use anything other than the *Random8Plus* strategy, the *no_rounds* field is likely to have a large value or 0, causing the function to timeout or to never enter the randomization part of the function (problem 2). If we use the *zero* strategy, the *data* field is also initialized to 0; as the update calculation uses multiplication a 0 in the *data* field will always result in an output of 0 (problem 3).

The underlying problem of these initialization strategies is that they are potentially too coarse-grained. Hence, if the emulation was unsuccessful (i.e., the function crashed or every output destination was dismissed) by using a coarse-grained memory layout, we try to create a fine-grained one. More specifically, we search for a fine-grained memory layout by identifying different fields in the memory state and initializing them individually. To identify memory fields in the memory state during the emulation process, our emulator tracks all memory read and write operations. We classify all locations as memory fields that are read before any write operation was performed on it (if any write operation is performed at all). We can safely ignore memory locations that are written by the algorithm

```
1  typedef struct state {
2      int no_rounds;
3      int error_code;
4      int data[128];
5  } memory_state;
6
8  int get_random(memory_state *state) {
9
10     /* Check PRNG if in error state */
11     if(state->error_code) {
12         return 0;
13     }
14
15     /* Generate random number */
16     int result = state->data[0];
17     for(int i=0; i < state->no_rounds; i++) {
18
19         /* Do some arithmetic on result with state->data */
20         [...]
21         result = state->data[0]
22
23         /* Update PRNG state */
24         for(int j=1; j < 128; j++) {
25             state->data[j-1] = state->data[j % 128] * 0xCAFE;
26         }
27     }
28     return result;
29 }
```

Problem 1
Error state check from memory state.

Problem 2
Number of rounds from memory state.

Problem 3
State data update calculation based on multiplication.

Figure 3.3: Code example showing three problems preventing a successful output generation if the different fields of the memory state are not considered individually. Note that this code example is constructed to highlight the problems we encountered in practice.

function before they are read (e.g., temporary data fields), as the function initializes them. Once we identified memory fields, we start fuzzing them by using the same input generation strategies as described in Section 3.3.3.2. Part c) of Figure 3.2 depicts the memory fuzzing process. We start our fuzzing process in the following situations:

- *Emulation Crashed*: if the emulation of the algorithm function crashes with the selected memory initialization strategy, we use fuzzing for a limited time and try to generate a valid memory layout. If the fuzzing is successful (the function does not crash), we start our data collection process with the fuzzed memory layout.

- *Output Destinations Dismissed*: if the emulation dismissed all output destinations, we use fuzzing for a limited time to generate more code coverage. The emulator tracks each basic block that is reached during the emulation. Once our fuzzing process reaches a new basic block, the emulation process restarts with the newly created memory layout as the initial configuration. This way, error checks are bypassed successfully (e.g., problem 1 in Figure 3.3), and a valid memory layout for the candidate function can be found.

3.3.4 Check Generated Output for Randomness

To decide if we have found an algorithm function, we check the quality of the created random data. More specifically, we check the output data against a random number generator test suite consisting of multiple tests. If at least one of its tests pass, we consider the candidate an algorithm function. We use multiple random number generator tests as a single one might incorrectly fail and dismiss an actual PRNG or CHF.

Most random number generator tests need a lot of data to work on (ranging from several MBs to multiple GBs of required data). However, if not enough data is provided, the test suite duplicates the same data to satisfy its data size requirement. This duplication results in a lower quality for the generated random data and may lead to erroneously failing tests. However, generating more data requires more emulation time, which in turn increases the total time needed for the analysis. For example, multiple random number generator tests failed to recognize a PRNG implementation of *libtomcrypt* (using the RC4 algorithm) due to limited input data. By using multiple random number generator tests, we reduce false dismissals, while improving the overall runtime.

Should all random number generator tests fail, we restart with a new memory initialization strategy. Part d) of Figure 3.2 shows the randomness test process.

3.3.5 Classification

Up to this point, we only determined if a function behaves as assumed in our interaction model in Figure 3.1 and generates random data. However, the analysis does not clearly distinguish between PRNG and CHF. To give the analyst an indicator, we developed a classification process based on the observation that PRNG implementations usually generate an output of arbitrary size, while CHFs do not (except for SHA-3 as discussed in detail in Section 3.6).

As Table 3.1 demonstrates, most PRNG implementations have an argument providing the requested size of random data to create. Hence, we assume only function arguments of type *value* as output size arguments. To determine if an argument is the output size, we systematically place different sizes into these arguments. Then we restart the emulation (with the working memory layout we already established in Section 3.3.3) and check if the corresponding output destination contains the given amount of generated data. If the algorithm function generates all the requested sizes and places them into the output destination, we consider this function as a PRNG implementation. Otherwise, we consider it as a CHF. However, note that this is only an indicator for an analyst, as Table 3.1 shows that for example `mix_pool` of *libgcrypt* generate a fixed size output which would lead to a wrong classification. Additionally, as soon as the output is placed into a return register, we neither classify it as CHF nor PRNG. This distinction between PRNG and CHF is only to offer the analyst additional information.

3.4 Implementation

We implemented a prototype called *TropyHunter* as a proof of concept based on the approach presented in Section 3.3. *TropyHunter* uses IDAPython [79] for disassembling binaries and Unicorn [154] to emulate the candidate functions. It is written in Python and targets Linux x86-64 (amd64) binaries. Because IDA Pro [78] and Unicorn support multiple architectures, *TropyHunter* is easily extensible to work with other architectures (e.g., ARM) or platforms (e.g., Windows). Since the concept of our approach is architecture agnostic, extending *TropyHunter* to other architectures or platforms is merely an engineering effort (e.g., adding a new calling convention or writing a loader for the target file format). Our prototype implementation is fully automated and requires no interaction with the user. After providing *TropyHunter* with the binary file, it prints a list of found PRNGs and CHFs for further analysis.

The arithmetic heuristic used for the function preselection searches for basic blocks exceeding a certain ratio of arithmetic instructions (e.g., `mul` or `xor`). Since related work differ greatly in their used thresholds (e.g., 0.15 in [101] and 0.55 in [73]), we determined the optimal ratio value for our implementation after analyzing popular cryptographic libraries. We find that a ratio of 0.3 finds all relevant PRNG and CHF functions, while not missing any algorithm.

We limit the running time for both emulation and fuzzing based on empirical experience, both parameters are configurable if the need arises. Since we do not know the expected memory layout and try to create a valid one, emulation may have long runtimes or not terminate at all (e.g., infinite loops). *TropyHunter* terminates each emulation after 20 seconds to guarantee timely termination. During the fuzzing phase, we again limit the process to 20 seconds. The timeout values are obtained by our empirical trials and provide a balance between performance and accuracy: increasing any timeout value did not result in better results and only increased the runtime. To further reduce the runtime of our analysis, we dismiss output destinations if they contain identical data five times since a function creating the same data over and over again is neither a PRNG nor a CHF

(as described in Section 3.3.3.3). During our tests, the threshold of five times delivered a performance improvement without introducing any false negatives.

The emulation of the candidate function repeats until it creates a configurable number of output bytes. During our tests, we found 200 kB to be sufficient for our random number generator tests to pass. The created random data is checked with multiple tests from the *dieharder* random number generator test suite [24]. To generate the required amount of 200 kB output data, we adopt the following process: PRNG implementations mostly create random data of arbitrary sizes (as illustrated in Table 3.1 in column *Output Size via Argument*) and usually have an argument of type *value* as a size parameter. However, instead of just setting the *value* to 200 kB, we pass the value 8 specifying an output of 8 bytes. We then emulate the function multiple times until we have enough output data (by aggregating each 8 bytes output data until we obtain 200 kB in total). We favor this approach for two reasons: First, our PRNG and CHF model presented in Section 3.2 shows that the algorithm function updates a memory state which is used to generate different random data each time it is executed. Hence, to take this model into account in our analysis, we have to emulate the candidate function multiple times with the same input to determine if it is an algorithm function (and in this case a PRNG function). Second, emulating a function with 200 kB of output data usually takes more than 20 seconds (our timeout threshold). Increasing the timeout at this point is too costly to the overall runtime of the analysis, as the runtime cost has also to be paid for the vast amount of non-PRNG and non-CHF functions. Should the candidate function not take the output size via a function argument, we only extract 4 bytes of output data. This is based on our observations in Table 3.1 in column *32 Bit Output*: if a PRNG function does not have a dynamic output size, some implementations only generate 32 bits output.

3.5 Evaluation

We evaluated *TropyHunter* on both open-source and closed-source applications. Open-source applications allow us to measure the reliability of our analysis by validating the PRNGs and CHFs against a ground truth. To demonstrate the capabilities of *TropyHunter*, we also chose various closed-source applications, ranging from large desktop-level binaries with more than 600,000 functions to smaller embedded applications which are part of a POSIX-compatible real-time OS. Furthermore, to identify the impact of different memory initialization and fuzzing strategies on the results, we describe their effect on identifying PRNGs and CHFs. We performed all experiments for our evaluation on an Ubuntu Server 18.04 LTS virtual machine with 32 cores and 128 GB of RAM running on a server with two Intel Xeon Silver 4114 processors.

3.5.1 Open-Source Applications

We evaluate *TropyHunter* on a diverse set of open-source software to generate a ground truth for our analysis. We chose a set of 12 programs for our evaluation, amongst them six popular cryptographic shared libraries: *libwolfssl* 3.15.7, *libgcrypt* 1.8.4, *libcrypto* 1.1.2 (OpenSSL), *libnettle* 3.4, *libsodium* 1.0.16, and *libtomcrypt* 1.18.2. Note that these are the

Table 3.3: Results of the analysis accuracy of *TropyHunter* for PRNGs and CHFs on the function level for the six cryptographic shared libraries and six other programs. *# Functions Total* indicates the total number of functions within the program that are identified by IDA Pro [78]. *# Found* describes the number of PRNGs or CHFs which were identified by *TropyHunter*. *# Algorithm* are functions that are part of a PRNG or CHF algorithm. *# Other Cryptography* are functions that are directly connected to PRNG or CHF or other cryptographic algorithms. *# False Positives* indicates functions which were identified but could not fit the other categories. We omit a column showing false negatives, as a clear distinction of members of this group on the function level is not always possible.

				Categories		
Program	**Time (hh:mm:ss)**	*# Functions Total*	*# Found*	*# Algorithm*	*# Other Cryptography*	*# False Positives*
libwolfssl	4:50:01	1,194	10	6	4	0
libgcrypt	12:36:23	1,531	77	30	46	1
libcrypto	54:06:22	6,387	103	32	71	0
libnettle	2:55:06	578	34	16	18	0
libsodium	9:28:22	991	15	11	4	0
libtomcrypt	9:41:51	1,192	55	46	7	2
PHP	43:41:05	10,398	35	33	1	1
Ruby	21:43:22	6,153	2	2	0	0
Lighttpd	0:29:31	834	4	4	0	0
PostgreSQL	13:00:51	12,341	11	11	0	0
libaprutil	1:03:24	834	4	4	0	0
libapr	7:03:32	1,339	2	2	0	0
Sum		43,772	352	197	151	4

libraries which we used to create the interaction model for PRNG and CHF implementations depicted in Table 3.1 and Table 3.2. To show that our analysis approach works in a generic way and is not limited to only those libraries, we also evaluated *TropyHunter* on additional programs, namely *PHP* 7.3.2, *Ruby* 2.6.1, *Lighttpd* 1.4.54, *PostgreSQL* 11.4, and the Apache Portable Runtime (*libaprutil* 1.6.1 and *libapr* 1.7.0) which contains the platform-specific implementations for the Apache web server.

We split the evaluation of open-source software into two parts: first, we evaluate based on the functions to show what kind of functions *TropyHunter* identifies. Then we evaluate with regards to the algorithms (e.g., MD5, SHA-256, or Fortuna) to measure the accuracy of *TropyHunter* in actually identifying PRNGs and CHFs in a program. The rationale behind this split is that an algorithm may also use helper functions (e.g., for managing the memory, such as `malloc`, or to perform a specific calculation). Additionally, these helper functions might be reused by multiple functions (possibly unrelated to the PRNG or CHF algorithms). As a result, it is not always possible to decide which functions belong to which implementations. In summary, we evaluate on function and algorithm level, while also respecting ambiguous results.

3.5.1.1 Function Level

Table 3.3 shows the evaluation results. We divided the found functions into three different categories: *Algorithm*, *Other Cryptography*, and *False Positives*. The *Algorithm* category contains functions that implement a PRNG or CHF algorithm (or a part of it). Since an implementation of an algorithm is separated into several functions, *TropyHunter* often finds multiple functions belonging to one algorithm.

The *Other Cryptography* category contains functions related to PRNGs and CHFs or other cryptographic algorithms, that we cannot clearly dismiss as false positives. This category contains functions that use a PRNG or CHF internally and consequently are found by *TropyHunter*. For example, the function `_gcry_md_hash_buffer` in *libgcrypt* provides a simpler interface to the developer and wraps multiple CHFs. Additionally, this category contains other cryptographic algorithms found by *TropyHunter*. For example, stream ciphers and block ciphers in operation modes that transform them into a stream cipher (e.g., counter mode or output feedback mode). Since cryptographically secure PRNGs and stream ciphers are basically the same (i.e., most stream ciphers generate a stream of random data that is xor combined with the plaintext to encrypt it [84]), they are identified. Analogously, message authentication code (MAC) functions are found since they are similar to CHFs and, additionally, CHFs are often used as a building block for MAC functions [84]. Furthermore, *TropyHunter* found functions that were executing different cryptographic operations (e.g., elliptic curve arithmetic).

The *False Positives* category contains the remaining functions that *TropyHunter* found. Interestingly, these functions are still related to cryptographic algorithms as all identified functions implement some kind of checksum.

We do not evaluate the number of false-negative functions—however, we do assess false-negative algorithms in Section 3.5.1.2. We are not able to provide an objective number of false-negative functions due to two reasons: First, an algorithm may use helper functions which could be reused by multiple functions, possibly unrelated to PRNG or CHF algorithms, making it impossible to distinguish if a helper function belongs to a specific algorithm. Second, *TropyHunter* is not designed to find every single function of an algorithm. It is designed to find functions that correspond to our proposed interaction model and create randomness. Not every function of an algorithm satisfies these requirements by itself. As a consequence, we omit false-negative functions from our evaluation and only evaluate false negatives on the algorithm level (see below).

Table 3.3 shows that the majority of the identified functions are part of a PRNG or CHF. Searching a total of 43,772 functions, *TropyHunter* marks 352 (0.8%) functions as relevant. Of the 352 functions, we correctly find 197 *Algorithm* and 151 *Other Cryptography* functions, while producing 4 *False Positives*. We got the worst result for *libcrypto*, where we find 103 in 6,387 functions, of these 103 found functions 32 are *Algorithm* functions. The reliability of our analysis decreases if the target application is a cryptographic shared library. This is because these libraries contain a diverse set of cryptographic functions (e.g., encryption functions, MAC functions, CHFs, or PRNGs) and, therefore, *TropyHunter* finds multiple functions related to PRNGs or CHFs. As a result, cryptographic shared libraries are a stress test for *TropyHunter*.

For applications other than cryptographic libraries, the results are much better (see lower part of Table 3.3). For example, in the case of *PHP*, a complex application with over 10,000 functions, *TropyHunter* identifies 35 functions, 33 of them are *Algorithm* functions. In *Ruby*, *Lighttpd*, *PostgreSQL*, *libaprutil*, and *libapr*, we can report a precision of 100% as all identified functions are *Algorithm* functions.

3.5.1.2 Algorithm Level

The implementation of an algorithm (e.g., MD5, SHA-256, or Fortuna) is usually divided into multiple functions. A CHF, for example, typically has two components: one that processes the input data updating the memory state and a second that finalizes the hash calculation generating the hash digest. In this section, we evaluate if *TropyHunter* can identify algorithms as a whole. To generate a ground truth, we identified all PRNG and CHF algorithms in the source code of all programs in our evaluation set. We additionally determined the algorithms of the identified functions of the *Other Cryptography* and *False Positives* category. Since these algorithms are usually divided into different functions, our result may group multiple functions together into one algorithm. Note that, we omit functions from the *Other Cryptography* category if they are detected as part of a PRNG or CHF algorithm. For example, *TropyHunter* identifies the function `php_session_gc` in *PHP* because a PRNG is used to generate a session ID. Since we already counted the PRNG algorithm, we omit this function from the evaluation on the algorithm level. We consider an algorithm as correctly classified if *TropyHunter* finds and correctly classifies at least one function of that algorithm. Table 3.4 depicts algorithms identified by *TropyHunter*.

The results show that *TropyHunter* can reliably identify CHF algorithms within binary executables. For example, for *libtomcrypt*, we can detect and classify 15 out of 16 algorithms in the library. *TropyHunter* has the lowest detection rate for *libwolfssl*, where two out of five algorithms were missed.

For PRNGs, the results are slightly less reliable. In the best case (*libtomcrypt*), five out of five PRNGs were found and classified correctly. In the worst case (*libcrypto*), one out of four was found and correctly classified. We investigated potential causes for the discrepancy between detecting PRNG algorithms and CHF algorithms: The majority of failures can be traced back to implementations using functions pointers. For example, the PRNG implementation `drbg_ctr_generate` in *libgcrypt* uses a symmetric block cipher to generate random data. The used block cipher function is registered during the initialization of the PRNG, and the memory state stores the corresponding encryption function as a function pointer. Since the function pointer was never seen during our analysis, the emulation crashes when it is used. As function pointers are a hard problem in static analysis [112], we will further discuss possible ways to tackle this issue in Section 3.6. For applications other than cryptographic libraries, the results are much better (see lower part of Table 3.4). For example, the recall and precision for *Lighttpd*, *libaprutil*, and *libapr* are 1.00. These results further substantiate our observation that cryptographic shared libraries are a stress test for *TropyHunter*.

As described in Section 3.2.2.1, we created an interaction model of CHFs based on commonly used algorithms such as MD5, SHA-1, or SHA-256. During our analysis, we found rarely used CHFs such as SM3 [138], MDC2 [22], or HAVAL [166] that were never studied

Table 3.4: Results for the classification accuracy of *TropyHunter* for PRNGs and CHFs on the algorithm level for the six cryptographic shared libraries and six other programs. *# Alg.* describes the number of algorithms that exist in the target program, *# Found* indicates the number of found PRNG or CHF algorithms, or the algorithms for the *Other Cryptography* or *False Positives* category. *# Class.* depicts the number of algorithms correctly classified by *TropyHunter* as PRNG or CHF. *Recall*, *Precision*, and *F1 Score* depict the corresponding accuracy values.

	CHFs			**PRNGs**			**Other Crypto**	**FPs**			
Program	*# Alg.*	*# Found*	*# Class.*	*# Alg.*	*# Found*	*# Class.*	*# Found*	*# Found*	*Recall*	*Precision*	*F1 Score*
libwolfssl	5	3	3	2	2	2	1	0	0.71	0.83	0.77
libgcrypt	13	12	12	6	2	0	10	1	0.63	0.52	0.57
libcrypto	12	12	12	4	1	1	13	0	0.81	0.50	0.62
libnettle	9	8	8	3	2	2	8	0	0.83	0.56	0.67
libsodium	3	3	3	3	3	2	2	0	0.83	0.71	0.77
libtomcrypt	16	15	15	5	5	5	5	2	0.95	0.74	0.83
PHP	14	11	11	4	3	2	0	1	0.72	0.93	0.81
Ruby	0	0	0	2	2	1	0	0	0.5	1.00	0.67
Lighttpd	2	2	2	1	1	1	0	0	1.00	1.00	1.00
PostgreSQL	3	2	2	2	2	1	0	0	0.60	1.00	0.75
libaprutil	3	3	3	0	0	0	0	0	1.00	1.00	1.00
libapr	1	1	1	1	1	1	0	0	1.00	1.00	1.00
Sum	81	72	72	33	24	18	39	4	0.79	0.68	0.73

by us during the creation of the interaction model. This demonstrates that *TropyHunter* generically identifies PRNGs and CHFs without relying on algorithm-specific artifacts.

3.5.2 Closed-Source Applications

To demonstrate that *TropyHunter* is applicable for closed-source applications, we analyzed multiple programs only available as binary executables. We evaluated against *spotify* 1.1.5.153.gf614956d, *TeamViewer* 14.3.4730, and *Beyond Compare* 4.2.10.23938. Furthermore, to demonstrate that *TropyHunter* is OS agnostic, we evaluated against the *Pluggable Authentication Module* binary from a POSIX-compatible non-Linux OS, namely QNX 7.0. Table 3.5 depicts the results of our evaluation. As can be seen, the analyses provide a practical starting point for an analyst for each application. For example, the analysis of *spotify* reduced the more than 647,000 functions to only 30 candidates (26 CHFs and 4 PRNGs). The analysis of the *pam_qnx* shared library, which is part of the authentication module of the QNX OS, pinpointed six candidate functions with five of them classified as CHFs. Luckily, the debug symbols in *qnx_pam* were not removed, allowing us to verify the results. The five classified CHFs are indeed CHFs (SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512). *TropyHunter* could not classify the remaining function as a PRNG, since the function just returns a fixed four bytes random integer value in the return register. Our classification logic expects an arbitrary output size for a PRNG and a fixed output size in an argument pointer for a CHF. Thus, this function was found but not classified. Note that we did not see any additional CHFs or PRNGs in the symbol names, indicating that all CHF and PRNG functions in *qnx_pam* were identified.

Table 3.5: Results of *TropyHunter*'s analysis of closed-source applications. *# Functions Total* indicates the total number of functions within the program that are identified by IDA Pro [78]. *# Found* describes the number of PRNGs or CHFs which were identified by *TropyHunter* before classification. *# CHFs* depicts the number of identified functions classified as CHF, whereas *# PRNGs* is the number of functions classified as PRNG.

				Classifications	
Program	**Time (hh:mm:ss)**	*# Functions Total*	*# Found*	*# CHFs*	*# PRNGs*
spotify	65:36:05	647,035	43	26	4
TeamViewer	71:24:31	89,984	115	81	22
bcompare	40:35:31	33,180	46	35	3
pam_qnx	1:52:30	194	6	5	0

Overall, the results show that *TropyHunter* is a valuable tool for an analyst starting the reverse engineering process: after running *TropyHunter*, the analysis can focus on the few remaining pertinent functions.

3.5.3 Ablation Study for Emulation Strategies

To show the impact of the proposed memory initialization (Section 3.3.3.2) and fuzzing strategies (Section 3.3.3.4), we assess each of them independently on all open-source programs of our evaluation set. To measure the memory initialization strategies, we deactivated the fuzzing mechanism of *TropyHunter* and only activated one memory initialization strategy at a time (e.g., *Zero* or *Random8Plus*). Furthermore, we extracted the functions that were only identified by the corresponding memory initialization strategy and not by the others. To evaluate the fuzzing strategies, we activated all memory initialization strategies and only used one of our fuzzing mechanisms (either the *Emulation Crashed* or *Output Destinations Dismissed* strategy). Again, we also extracted the functions that were identified by the corresponding fuzzing mechanism and not the other. Table 3.6 depicts the results. The evaluation shows that no memory initialization strategy alone can find all algorithm functions. For example in the case of *libgcrypt*, the largest number of functions are identified with the *Random* memory initialization strategy (54 functions), whereas the *Zero* strategy finds 40 functions. However, these 40 functions contain eight functions that were not found by any other memory initialization strategy and thus contributed directly to the overall analysis result. But even so, the memory initialization strategies alone do not find all 77 functions. For the fuzzing strategies, we observe a similar behavior. The fuzzing mechanism used if the *Emulation Crashed* was able to raise the identification to 73 functions. These 73 functions contain five functions that are not found by the fuzzing mechanism for *Output Destinations Dismissed*. However, the fuzzing mechanisms alone are still not able to identify all 77 functions. Overall, the results show an overlap of identified functions, however, no memory initialization or fuzzing strategy alone is able to detect all functions. Therefore, each strategy contributes to the effectiveness of *TropyHunter* and a combination of them is necessary to deliver the best results.

Table 3.6: Results of measuring the impact of the memory initialization and fuzzing strategies. *# All* indicates the number of identified functions if all strategies are used. *# Zero*, *# Random*, *# Random8Plus*, *# Random4Plus*, and *# RandomPlus* depict the number of identified functions if only the corresponding memory initialization strategy is used. The numbers in parentheses represent the number of functions that are only identified by the corresponding memory initialization strategy. *# Emulation Crashed* and *# Output Destinations Dismissed* give the number of identified functions if only the corresponding fuzzing strategy is activated (as well as all memory initialization strategies). The numbers in parentheses represent the functions that are only identified by the corresponding fuzzing strategy.

		Memory Initialization					**Fuzzing**	
Program	*# All*	*# Zero*	*# Random*	*# Random8Plus*	*# Random4Plus*	*# RandomPlus*	*# Emulation Crashed*	*Output # Destinations Dismissed*
libwolfssl	10	6 (0)	7 (0)	6 (0)	6 (0)	7 (0)	7 (0)	10 (3)
libgcrypt	77	40 (8)	54 (0)	52 (5)	50 (0)	53 (0)	73 (5)	71 (3)
libcrypto	103	52 (4)	81 (0)	90 (10)	78 (0)	82 (1)	98 (0)	102 (4)
libnettle	34	18 (1)	26 (0)	28 (2)	26 (0)	26 (0)	31 (1)	30 (0)
libsodium	15	11 (0)	14 (0)	14 (0)	14 (0)	14 (0)	15 (1)	14 (0)
libtomcrypt	55	34 (8)	23 (4)	39 (7)	20 (1)	19 (0)	51 (1)	54 (4)
PHP	35	29 (1)	23 (0)	30 (2)	24 (0)	24 (0)	32 (0)	35 (3)
Ruby	2	1 (0)	2 (1)	1 (0)	1 (0)	1 (0)	2 (0)	2 (0)
Lighttpd	4	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)
PostgreSQL	11	11 (0)	11 (0)	11 (0)	11 (0)	11 (0)	11 (0)	11 (0)
libaprutil	4	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)
libapr	2	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)

3.5.4 Runtime Performance

As illustrated in Tables 3.3 and 3.5, *TropyHunter* exhibits a long runtime to analyze large applications. This is because the research prototype is written in Python and not optimized for performance. Although the emulation of *TropyHunter* is done by Unicorn [154] and therefore written in a system programming language, multiple callbacks into *TropyHunter* make a transition back to Python (e.g., a callback executed after emulating each instruction because of unsupported instructions by Unicorn). We like to stress that this is merely an engineering problem based on the chosen programming language. Finally, it is worth mentioning that the analysis only needs to be performed once.

3.6 Discussion

In the context of this chapter, we assume that PRNGs and CHFs create a certain level of randomness that can pass random number generator tests. Developers, however, can implement self-designed PRNG or CHF algorithms for their application regardless of the quality of the generated data. These functions might not pass the random number gen-

erator tests and hence would be discarded by *TropyHunter*. Since an analyst might be interested in such functions, *TropyHunter* can be configured to disable the random number generator tests. Additionally, we use random number generator tests to determine the randomness quality of the generated data. However, even if the generated data passes these tests, the identified functions are not always suitable for cryptographic purposes. For example, during our evaluation, *TropyHunter* found the CRC32 checksum algorithm, which passed random number generator tests. While the generated data has sufficient randomness to pass these tests, properties of the detected algorithms (e.g., linearity) are not considered.

The biggest limitation we encountered during our evaluation were implementations using function pointers. Primarily cryptographic shared libraries use function pointers to provide a modular architecture. For example, the NIST recommendation SP800-90A [15] gives a design for a PRNG based on CHFs. However, CHFs are merely a building block in the design, and a specific implementation can be replaced by a newer one (e.g., SHA-256 with SHA-3). To support this modular architecture, some implementations in cryptographic shared libraries expect the developer to pass a pointer to a CHF as an argument to the PRNG initialization routine. The pointer is then stored in the PRNG memory state and used when the PRNG algorithm function is executed. Since *TropyHunter* does not know a valid target address for this function pointer, usage of it results in a crashed emulation. One way to tackle this issue is to statically extract all addresses of functions that are written to memory locations in the application. We can then systematically replace the target of an indirect call crashing the emulation by replacing it with the extracted function addresses. However, this prohibitively increases the analysis time of *TropyHunter*. Since we only encountered eight implementations using function pointers (and only in cryptographic shared libraries), we resorted to have them as a limitation.

A similar problem occurs for applications written in C++. Internally, C++ classes use tables of function pointers to execute virtual functions. Each object using virtual functions hold a pointer to the corresponding function table. As a result, to analyze C++ applications techniques to extract these function tables have to be developed and integrated into the same probing approach that could be used for function pointers.

Besides this, *TropyHunter* also has some minor limitations as well. First, due to the probabilistic nature of the presented analysis approach, the results of an analysis may differ when performed multiple times on the same application. This stems from the fact that depending on the used memory initialization strategy (e.g., *Random8Plus*), the memory is filled with random data. Hence, in theory, a field in a memory state might be filled with an incompatible value that results in an unsuccessful emulation or insufficient randomness in the output. However, the evaluation of our memory initialization and fuzzing strategies in Section 3.5.3 shows an overlap in the identified functions between the different strategies. Hence, minimizing the risk of missing a function due to a disadvantageous random value. Furthermore, we did not encounter a missed function during our evaluation because of this probabilistic behavior and thus deemed it acceptable.

Second, we assume the algorithm functions pass the generated data directly to the caller functions. Due to this assumption, we expect data to be returned either via a given pointer to memory or directly in the return register. Since a developer can implement the data flow freely (e.g., writing the generated data into a file and reading this file from

the caller function), we might miss these peculiar implementations. Note that, since we only encountered the assumed direct data flow between the functions in our evaluated open-source applications, we narrowed down our scope.

Third, *TropyHunter* uses simple fuzzing for a short time to generate a fine-grained memory layout of the state used by the algorithm. Still, common fuzzing roadblocks such as checking for magic numbers or checksums cannot be bypassed with such a simple technique. To tackle this problem, a more sophisticated fuzzing approach (e.g., [14,31,124, 162]) could be used. However, during our evaluation, we did not encounter any PRNG or CHF implementation using such constructs.

Finally, the distinction between PRNG and CHF is not always clear in practice. Our classification heuristic tries to distinguish between them by assuming that PRNG implementations create data of arbitrary size and CHF implementations create data of a fixed size. However, there are certain cases where this assumption does not hold. For example, the SHA-3 algorithm is based on an approach called sponge construction [53], which allows it to generate output of arbitrary size. This design allows SHA-3 to be used as PRNG. In these edge cases, accurately classifying an algorithm as PRNG or CHF is not always possible. Similarly, *TropyHunter* usually detects and classifies stream ciphers as PRNGs. This is because stream ciphers are basically the same as cryptographically secure PRNGs. Most stream ciphers generate a stream of random bytes to encrypt the plaintext by just xor combining them [84].

3.7 Related Work

In recent years a relevant stream of work has explored identifying encryption algorithms in executable binaries. For example, *K-Hunt* [101] finds encryption algorithms in binary applications and identifies their insecurity. It utilizes dynamic analysis by using Pin [104] to record execution traces of the target application. However, executing the application sidesteps the problem of finding the correct memory layout, which *TropyHunter* tackles. *K-Hunt* inherits typical issues of dynamic analysis, such as dependency on code coverage. Furthermore, *K-Hunt* assumes that the analyst has test cases that execute the application such that the encryption ciphers are used. However, creating such test cases for closed-source applications (and especially shared libraries) is a considerable effort. Gröbert et al. [73] also presented a dynamic approach to find symmetric encryption algorithms in binaries. Their approach uses the execution trace of an application and heuristics to narrow down the search of the ciphers, and then reconstructs the corresponding memory. Subsequently, patterns are used and compared to a database to identify the exact algorithm. However, a small deviation from the reference implementation of the encryption algorithms can lead to false-negative results. Similarly, Xu et al. [160] presented *CryptoHunt*, a dynamic approach working on execution traces to search cryptographic functions in (obfuscated) binaries. The key insight is to compare formulas created on traces of the loop in a cryptographic function with the ones of a reference implementation.

Lestringant et al. [99] proposed a static approach based on data-flow graph-isomorphism to find encryption ciphers in binaries. To cope with different implementations of the same

algorithm, the created data-flow graph is normalized before it is used. However, it relies on previously generated signatures and thus is only able to find previously known algorithms.

In a different context, *X-Force* [123] tries to execute malware binaries. The goal is to bypass anti-debugging checks and identifying hidden functionalities without having any input available. This is done by forcing the execution of different paths when it reaches a check and allocating accessed memory locations on the fly to prevent crashes. Similarly, Godefroid presented *MicroX* [66], a virtual machine capable of executing specific code parts (e.g., functions) of x86 binary code without having any input data. *MicroX* automatically allocates accessed memory locations on the fly. To provide input values it takes different strategies such as filling memory locations with zero, filling them with random data, or use a user-provided method. Gao et al. [62] presented a semantic-learning and emulation-based approach to search vulnerabilities in binary applications. To emulate functions, they identify the used arguments and initialize them with random integer values.

3.8 Conclusion and Future Work

In this chapter, we presented the design and implementation of *TropyHunter*, an analysis approach to accurately find PRNGs and CHFs in binary executables in an automated way. We demonstrated that PRNGs and CHFs follow a similar interaction model and can thus be detected generically by searching for this behavior. Our evaluation showed that emulation of PRNGs and CHFs without any user interaction and knowledge of the expected memory layout is possible. Furthermore, we demonstrated on a diverse set of applications that our proposed approach is viable and showed that *TropyHunter* can be beneficial to analysts.

Providing function pointer support for *TropyHunter* is an interesting future advancement. Especially support for C++ applications, which rely heavily on function pointers. Since C++ is the programming language of choice when developing large and complex software relying on performance [145], supporting C++ allows the analysis to work on a broader set of programs. As a first step towards this advancement, we focus on C++ class-hierarchy reconstruction in binary applications in our next chapter to provide an analyst or analysis tool with important information about function pointers in C++ programs. Furthermore, extending *TropyHunter* to other architectures such as ARM could open interesting research opportunities. With such an extension, we could evaluate the security of PRNG and CHF implementations in embedded systems such as Programmable Logic Controllers (PLCs) [2] or IoT devices [35]. As embedded systems suffer from bad PRNG implementations [3, 36], analyzing them could uncover severe vulnerabilities.

Chapter 4

Uncovering Class Hierarchies in C++ Binaries

Analyzing C++ programs on the binary level is a challenging problem since high-level concepts used in object-oriented programming are often translated into binary code using function pointers. Such code contains more complexity than, e.g., traditional procedural code, and as a result, it is more difficult to understand. As the evaluation in Chapter 3 has shown, analyzing code using function pointers statically poses a problem for the analysis process. Automated tools that do not take function pointers into account might miss important conclusions because of incomplete results. An analyst manually analyzing a program using function pointers faces additional hurdles because of the indirect branches or even hits a dead end because of them. Furthermore, since the targets of these indirect branches are resolved only at runtime, they can be influenced by an attacker for introducing new malicious control flows by taking advantage of software vulnerabilities.

C++, the choice for implementing a huge industrial software base [145], contains a plethora of indirect branches. Since virtual objects support several methods from different classes in their hierarchy, most compilers implement dynamic dispatching of virtual calls using indirect branches. In practice, C++ programs are thus full of indirect calls, and most of these can be influenced not just by overflow-type vulnerabilities, but also by temporal bugs (i.e., use-after-free vulnerabilities). For instance, according to a recent study [164], in most libraries linked to Firefox, almost 7% of the existing call instructions are indirect calls, and about 40% of these indirect calls are virtual calls. This abundance of indirect calls makes analyzing C++ binaries essential but also significantly hard.

Indirect control-flow transfers rank among the greatest challenges for even the most basic analysis steps, such as the recovery of the control-flow graph (CFG) [9, 139]. Resolving the targets of indirect calls and jumps in a binary is difficult. At the binary level, we have no way to learn class-hierarchy information in the program directly. While we know that every virtual function call indexes a virtual function table (so called *vtable*), we neither know the vtables' exact locations, nor their relationships to each other. Reverse engineering such code from a given binary executable is therefore a very challenging task in practice.

Albeit challenging, vtable reconstruction directly from binaries can be useful in several domains. First, the class hierarchy helps the analysis of C++ legacy or closed-source code (e.g., by providing information about indirect branch targets). Second, since exploits commonly abuse vtables, security analysts can explore incidents affecting C++ applications when source code is not available. Finally, many defenses that harden C++ binaries can leverage the class-hierarchy information for delivering sound protection of programs in the absence of source code. Current state-of-the-art binary-only protection approaches use weaker characteristics typical for C++ applications to protect virtual callsites, such as allowing all existing classes at a virtual callsite [127], or enforcing that the pointer to the vtable resides in read-only memory [63]. This usage of weaker characteristics stems from a lack of precision and scalability of current class-hierarchy reconstruction approaches [59, 83, 85].

Therefore, if we can successfully recover the class hierarchy from a binary, we can improve state-of-the-art binary-level defenses that can benefit from such information. For instance, we can ensure that virtual function calls conform to the class hierarchy, and therefore provide strong guarantees against attempts to hijack the control flow of the program (so called *vtable-hijacking attacks*). Another example is to ensure that objects of different type classes are allocated from different memory pools to prevent the reuse of memory in a type-unsafe manner—a common source of use-after-free exploits. For both these example applications, extracting the class hierarchy of the binary program is essential. Notice that this is *only* a set of mitigations that rely on C++ semantics, although an important one, given that prior work argued that C++ binary-level defenses have trouble stopping control-flow hijacking attacks due to the lack of class-hierarchy information [135].

4.1 Introduction

In this chapter, we focus on the problem of reconstructing class relations directly from binaries. Our approach does not rely on embedded RTTI information (metadata emitted by the compiler for resolving class information at runtime, often stripped), does not rely on particular compiler flags, and works on industrial software. Since reconstructing class relations is a hard problem by itself and information concerning the direction of the relation is not available in binaries, we only focus on reconstructing class hierarchies as a set and ignore the direction of inheritance. Our system, *Marx*, can accurately reconstruct 93.2% of the hierarchies for Node.js and 88.4% of the hierarchies for MySQL Server. Overall, we have successfully applied *Marx* to more than 80 MiB of binary code to demonstrate the practicality of our implementation.

Marx is a valuable framework for the reverse engineering community, however, as we have already mentioned, security applications can leverage class relations for protecting binaries. The information provided by the analysis allows us to implement stronger binary-level defenses using type-based invariants. To showcase the practicality of *Marx*, we develop two binary-level defenses on top of it. The first application is a *vtable protection* system to prevent virtual calls to methods that do not belong to the class hierarchy and mitigate vtable-hijacking attacks. The second application is a custom heap allocator to

support *type-safe object reuse* by placing newly allocated objects in memory pools based on their type.

Both security applications use the class hierarchy recovered from a binary. We demonstrate that, even when the extracted class hierarchy is imperfect, our defenses can improve security at reasonable performance and without breaking programs. To compensate for the imprecision of the analysis, our vtable protection solution treats invariant violations as anomalies and triggers more heavyweight checks on a slow path (trading off on performance). Our type-safe object reuse solution, in turn, can gracefully tolerate type-to-pool mapping mismatches (trading off on security). In short, we show that it is possible to build *fully conservative* binary-level defense solutions on top of imprecise information, exposing new interesting and previously unexplored tradeoffs.

Contributions In summary, the contributions of this chapter are as follows:

1. We design and implement *Marx*, a framework for reconstructing class hierarchies directly from binary executables that do not embed RTTI information and are produced with arbitrary compiler flags.

2. We evaluate *Marx* with more than 80 MiB of binary code, and we show that vtables can be reconstructed from binaries with high precision. As an example, *Marx* can accurately reconstruct 93.2% of the hierarchies for Node.js and 88.4% of the hierarchies for MySQL Server.

3. We develop two security applications for binaries based on class hierarchies exported by *Marx*: *vtable protection* and *type-safe object reuse*. Our applications show it is possible to tolerate imprecise information when building sound binary-level defense solutions by trading off on performance and security.

The prototype implementation of *Marx* and the data we used for the evaluation are available under an open-source license at `https://github.com/RUB-SysSec/Marx`.

Outline This chapter is structured in the following way: Section 4.2 explains our analysis approach to reconstruct class hierarchies. The implementation of *Marx* is described in Section 4.3. Two security applications build atop of *Marx*'s analysis results are given in Section 4.4. Section 4.5 presents the evaluation of *Marx* as well as the two developed security applications. We then discuss our approach in Section 4.6 and give an overview of related work in Section 4.7. Section 4.8 ends this chapter with a conclusion and discussion of possible future work topics.

The chapter is based on collaborated work with Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida, which was published at the Network and Distributed System Security Symposium (NDSS) 2017 [120].

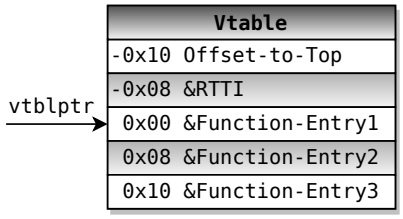


Figure 4.1: Structure of an Itanium C++ ABI vtable. The vtable pointer referenced in the code points to offset 0, where the table of function pointers starts. The two metadata fields (RTTI and Offset-to-Top) precede said table.

4.2 Approach

Given a binary executable, we aim at extracting the C++ class hierarchies as accurately as possible. To this end, we extract distinct properties that result from the way a C++ compiler implements the high-level concepts on the binary level as explained in Section 2.2. In the following, we describe the design of our approach that is implemented in a tool called *Marx*.
Generally speaking, our analysis is divided into two steps:

1. *Vtable Extraction.* Distinct patterns that are typical for vtables in the code are searched and information about the vtables is extracted.

2. *Static Analysis.* Given that these heuristics might lead to an overestimation, a static analysis of the code is conducted which searches for usages of the vtables found in the previous step.

In the following, we focus on the Itanium C++ ABI [60]. However, we stress that the presented methodology is applicable to other ABIs as well, such as the ARM [12] or the Microsoft C++ ABI [72].

4.2.1 Vtable Extraction

Virtual function tables in a binary are the key element to our analysis. By extracting usages of vtables, one easily finds points in the program where objects are either created (constructors) or destroyed (destructors). This, in turn, yields valuable information about the relation of different classes. Hence, one can view vtables as roughly analogous to a certain class and relation of vtables as corresponding to certain class hierarchies.

Our analysis applies multiple heuristics in order to locate vtables in a binary (**H-1** to **H-6**), as we discuss in the following. A rough, albeit simple, estimate can be used to restrict the search space to specific sections: As vtables are fully specified at compile time, they can be placed in *read-only* sections (heuristic **H-1**). Therefore, only those sections that typically hold vtables, such as `.rodata`, `.data.rel.ro`, and `.data.rel.ro.local`, are analyzed.

Figure 4.1 shows the typical structure of an Itanium C++ ABI vtable. A vtable consists of three different parts: the *Offset-to-Top* field, the *RTTI* pointer, and multiple *function*

entries. Each type has different properties which can be used to distinguish between them. Also, their order is fixed, which makes it possible to search for a series of consecutive patterns in a memory range, e.g., a specific section.

As seen in the figure, the *vtblptr* references the beginning of the function entries. Usually, this reference will be used in a constructor or destructor. However, we note that the other fields are usually not referenced at all (heuristic **H-2**).

Offset-to-Top is used to implement multiple inheritance for objects and encodes the offset from the sub-object to the base object. It is a mandatory field and always contains `0` if multiple inheritance is unused. Our approach checks the sanity of this entry by only allowing values in the range from `-0xFFFFFF` to `0xFFFFFF` as proposed by Prakash et al. [127]. In addition, the value cannot be a relocation entry. These checks constitute heuristic **H-3**.

RTTI holds a pointer to further type information for the class. Since this field is optional, the entry is either a pointer or `0`. If the entry is, in fact, a pointer to *data*, it has to point to non-executable memory (heuristic **H-4**). Since the vtable can be part of a shared library, this entry can also be a relocation entry.

Function entries hold a pointer to the virtual functions the class provides. Hence, an entry either points to the `.text`, `.plt`, or the `.extern` section of the binary, or it is a relocation entry. One of these properties has to be satisfied such that the analysis deems the function pointer to be valid (heuristic **H-5**).

In rare cases, the compiler sets the first few function entries of the vtable to `0`. This can happen for multiple inheritance constructs inheriting from abstract classes. To cope with these edge cases, our approach allows the first two function entries of the vtable to be `0`. This number was empirically found to be sufficient (relaxing heuristic **H-6**).

Finally, we can determine the beginning of a vtable by searching three consecutive words in memory that fulfill the properties outlined above. Further, the length of the vtable can be estimated by checking the subsequent function entries for validity.
To sum up, the heuristics we employ are:

H-1 Vtables have to lie in read-only sections.

H-2 In a candidate vtable, only the beginning of the function entries is referenced from the code.

H-3 *Offset-to-Top* lies within a well-defined range and it is no relocation entry.

H-4 *RTTI* either points into a *data* section or is `0`.

H-5 A function entry points into a *code* section or is a relocation entry.

H-6 (relaxing) The first two function entries may be `0`.

Note that the heuristics to find these patterns can lead to an overestimation of extracted vtables. Nevertheless, this does not impact the subsequent analysis notably since only *existing* vtables are referenced in the code (cf. heuristic **H-2**). We note that only in rare cases an overestimated vtable can result in an overestimated hierarchy. On the other hand, only an underestimation of vtables would lower the precision of the analysis, which is unlikely for the presented approach.

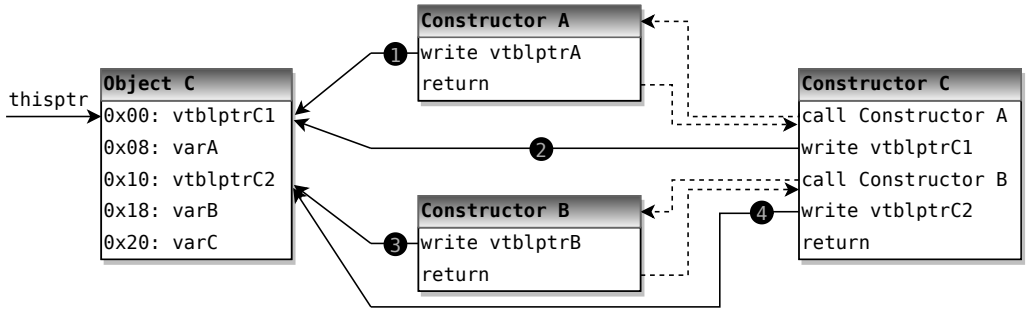


Figure 4.2: Order of *vtblptr* overwrites during the creation of an object of class `C`. The control flow of the execution of `Constructor C` is depicted as dotted lines.

4.2.2 Static Analysis

Now that we obtained a possibly overestimated set of vtable candidates, the second phase of our approach statically analyzes their relation based on indicators found in the binary code. Eventually, it yields distinct sets of vtables that are part of a class hierarchy. In the following, we discuss the various indicators the approach uses.

4.2.2.1 Overwrite Analysis

In Section 2.2.3, we discussed how during object creation, the constructor writes the *vtblptr* into the object. Further, when class `C` inherits from class `A`, as depicted in Figure 2.3 in Chapter 2, the constructor of class `A` is executed before the constructor of class `C` (top-down approach). This ensures that the inherited attributes of class `A` are initialized before the constructor of class `C` accesses them. Consequently, the *vtblptr* of the base class `A` is written into the new objects *before* the derived constructor writes the *vtblptr* of class `C` as shown in Figure 4.2. This also holds for multiple inheritance. In the given example, class `C` also inherits from class `B`. Hence, the constructor of class `B` is also executed before the constructor of class `C`. In this case, however, the *vtblptr* of class `B` is overwritten by the *vtblptr* to the sub-vtable of class `C`.

In contrast, during object deletion, the destructor of class `C` is executed before the destructor of class `A`, i.e., invocation follows a bottom-up approach. Therefore, the *vtblptr* of class `C` is overwritten by the *vtblptr* of class `A`. This also holds in the case of multiple inheritance for the sub-vtable of class `C` and *vtblptr* of class `B`, analogously. We can leverage this and detect the dependency of two classes by tracking if one *vtblptr* in an object is overwritten by another *vtblptr*. Remember that classes are roughly analogous to their vtable for our approach.

Naturally, this also means that we have to track the creation of a potential object by monitoring the `new` operators of the memory allocator. By correctly identifying constructors and destructors, our approach would also be able to make a statement about the *direction* of the inheritance, i.e., detect which class is the base and which the deriving class. In its current implementation, however, our approach reconstructs the class hierarchy as a plain set.

Due to compiler optimizations, constructors are often *inlined* next to the memory allocation of the object in the same function. The same concept is applied to destructors, analogously. While our approach does not identify constructors and destructors directly, it detects the characteristic pattern of *vtblptr* overwrites. As a result, we are able to detect overwrites for which a concrete classification as constructor or destructor is more involved. More precisely, our overwrite analysis is performed on statically calculated paths through multiple functions, as discussed in Section 4.3. Thus, we avoid having problems with function inlining as opposed to other approaches such as the one presented by Jin et al. [83].

4.2.2.2 Vtable Function Entries

Classes in the same hierarchy share attributes and a subset of virtual functions. Also, a class that inherits virtual functions from another class does not have to overwrite it. As a result, the vtables of both classes, base and derived, may contain multiple entries that point to the same function as in the other classes' vtable. In order to work in polymorphic constructs, this function entry has to be at the same position in the vtables.

Hence, we can employ a heuristic that checks if multiple vtables share the same function entry at the same position. If they do, we consider them as related. Obviously, specific entries like a `0` entry or the *pure virtual* function have to be excluded from this heuristic. Note that, similar to the overwrite analysis in its current form, no direction information is included. Naturally, if the compiler places the same function entry at the same position in unrelated vtables due to optimization passes, our analysis would find them to be related. This would lead to an overestimation of the found class hierarchy. However, the evaluation results in Section 4.5.1 show that this case can be neglected in practice.

4.2.2.3 Inter-Procedural Data Flow

We perform our analysis on paths through multiple functions. Even if analysis within a function is well defined, special attention has to be paid to the point where function boundaries are traversed.

Forward Edge Virtual functions are usually only called via an indirect call instruction. As these are dispatched dynamically based on a concrete object, the list of potential call targets is not easily retrieved statically. As the overwrite analysis analyzes paths through multiple functions, indirect callsites pose a roadblock and may prevent the analysis from following the call target. Consequently, vtable relations that are established beyond this point will be missed by the analysis. We say it lacks *context*, as no potential object at the callsite is known with which the callsite could be resolved.

In order to tackle this problem, the static analysis tries to resolve the indirect call instruction with the help of the context (essential a memory state) it built up on the current path. If the argument of an indirect call instruction is known, i.e., it dereferences a known *vtblptr*, we resolve the target function and continue the analysis in the newly discovered function on the path while keeping the current context.

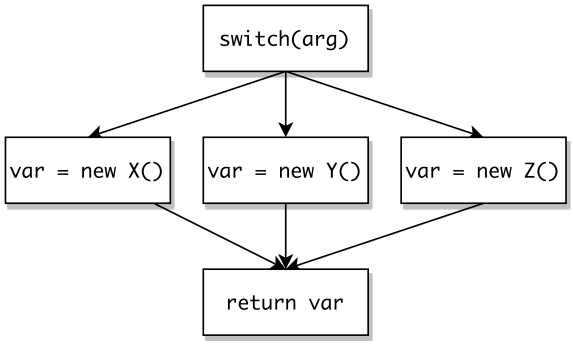


Figure 4.3: A simplified version of the function `GetSortComparisonObject` from the FileZilla FTP client. Depending on the given argument, it creates a new object of different classes and returns it. Without considering the backward edge, no statement about the classes referenced in this function can be made.

As an additional side effect of resolving the branch targets at a vcall in the current analysis run, we know which vtable is used for it. Since in polymorphic constructs only classes within the same hierarchy are allowed, a vcall can only be used by objects of dependent classes. Hence, if during the analysis the same vcall is used by objects containing different vtables, these vtables are related (i.e., the objects are from the same class hierarchy). However, no information about the direction of the inheritance of the classes is obtained.

Backward Edge In addition to passing known context on to the beginning of a more deeply nested function, the analysis also has to take return values of a callee back to its caller into account. Since different paths through the callee can result in multiple different return values, we generalize them into a set of return values, which is effectively the union of the individual return values on each path. Then, if the return value is used in a point where more context is required, such as a vcall, the information provided all possible return values can be used to, e.g., resolve an indirect callsite.

Consider the example given in Figure 4.3, which was encountered in the FileZilla FTP client. This function returns a different object depending on the given argument. The classes that are used to create the object (namely, `X`, `Y`, and `Z`) are part of the same hierarchy. Without tracking the possible return values of this function into a vcall of the caller, it is not possible to find the relation of these classes based on information of the forward edge alone.

4.2.2.4 Inter-Modular Data Flow

Applications are commonly divided into multiple modules (also known as *libraries*), where each module performs a specific task. On Linux, these modules are implemented using *shared objects*, which already include the notion that common functionality can be reused by different applications. Obviously, modules can depend on each other. Specifically, in C++, it is possible to interact with classes exported by a shared object. Such relations

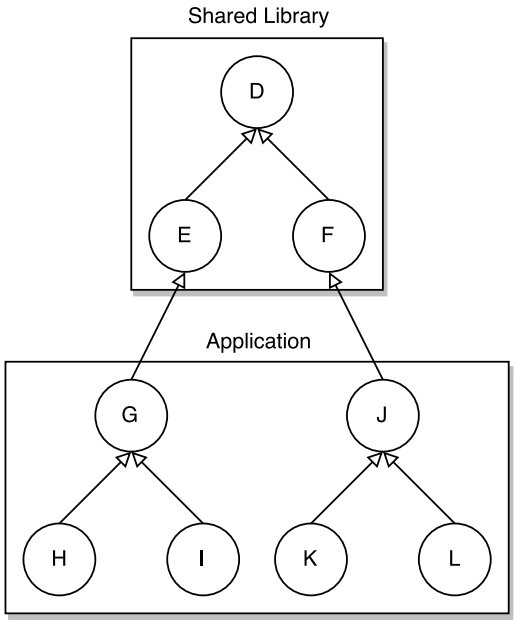


Figure 4.4: A class hierarchy that is connected via a shared library. Classes `G` and `J` both inherit from classes belonging to the same hierarchy in a different module. This fact is only observable when taking module boundaries into account.

would be hidden from our analysis when analyzing single modules only due to missing context. To counter this, we support inter-modular analyses.

Consider the example depicted in Figure 4.4. The application itself contains two class hierarchies that appear to be unrelated when looking at the main module. However, when taking the application's shared library into account, it becomes apparent that the two hierarchies are, indeed, related. Both hierarchies in the main module derive from a class of the very same hierarchy in the shared library. Isolated analysis of the individual modules would not have yielded the same result.

Our approach analyzes shared libraries first and creates data flow summaries of the return values and *vtblptr* overwrites in their respective modules. If a function in a shared library is called during the analysis phase, the aforementioned summaries are added to the current context accordingly. This way, we can also consider all class hierarchy connections that are outside of the currently analyzed module.

4.3 Implementation

Based on the concepts presented in Section 4.2, we implemented an analysis framework called *Marx* in C++ for Linux x86-64 binaries. Note that even though only Linux x86-64 binaries are supported as of now, the implementation can be easily extended to support more architectures. This is supported by the fact that we use VEX-IR from the Valgrind project [51] as our intermediate language.

In the following, we describe the implementation of *Marx*, discuss challenges we encountered, and explain how we solved them.

The first step of our analysis, vtable extraction, is performed on an IDA Pro [78] database using scripting facilities provided by IDAPython [79]. In addition to the candidate vtables identified via the heuristics H1 – H6 described in Section 4.2.1, the control-flow graph (CFG) of all known functions is extracted as well, which is used in the subsequent static analysis step.

The static analysis is mostly driven by a data-tracking engine which updates the context collected upon a path through the binary, as discussed in Section 4.2.2. Said context is used to track assignments of vtables to new objects and their overwrites in constructors and destructors. In its current state, the engine implements basic 64-bit VEX instructions, as this is already sufficient for our needs. The constructs we want to identify hardly involve any complex calculations and with our focus on real-world applicability, we have to weigh up *Marx*'s precision against its performance.

4.3.1 Starting Points and Context Sensitivity

Marx starts its analysis for each function in the target program separately, i.e., each function serves as starting point for at least one path. In order to obtain reasonable results for a specific path, however, enough *context* has to be known in which the function is executed. Otherwise, relations may be missed and the current traversal would not add to the overall results of the analysis.

There are multiple ways to ensure that the analysis visits a function g with a reasonable amount of context. For one, if g lies *deeper* within a path, it is reasonable that its caller f already adds vital information to the context. By starting an analysis path at f, the context added by it is available when arriving at g. If f constructs an object and g overwrites its vtable, this information would be missed by analyzing g out of context. This aspect is discussed further below.

On the other hand, it helps to consider the context a function *inherently* lies in. For example, g may be a virtual function. This, in turn, means that it belongs to (at least) one vtable. Hence, our analysis can be provided with some initial context: a *thisptr* exists and the object's *vtblptr* can be initialized to point to the vtable g lies in. For example, for the x86-64 Itanium C++ ABI, any occurrence of `[rdi + 0]` is then known to resolve to the current *vtblptr*.

This enables the analysis to handle operations on the object itself now that the target object is known (*vtblptr* overwrites or vcalls). Further, if g belongs to multiple vtables, g is analyzed in just as many contexts.

4.3.2 Path Creation and Convergence

The previous section already hinted on where possible paths through the binary *start*—at any known function. However, care has to be taken that a path *ends* at a point where all relations have been picked up by our analysis and no superfluous calculations are performed by further following the path. A naïve approach would be to simply analyze all possible paths at the given starting point. Yet, this leads to what is called the path

explosion problem, as the number of paths easily exceeds a feasible amount for non-trivial CFGs.

Marx decides which paths through a function are worth analyzing by following a heuristic: on each basic block in the CFG, a predicate is run which decides whether the block is considered *interesting*. We consider those blocks as interesting that contain one or more of the following cases: (i) an indirect call (i.e., a possible vcall), (ii) a (direct) call to a `new` operator, or (iii) an instruction operating on a *vtblptr*. With this heuristic, we attempt to visit only those blocks that add to the overall context we are interested in.

We then compute paths that try to visit as many interesting basic blocks as possible before reaching an exit block. In order to avoid a high computational complexity, a threshold t is introduced. If a function contains more than t interesting basic blocks, paths are generated that guarantee to visit *at least* one interesting block, but no attempts are made to maximize this number. Empirically, we found a threshold of 20 interesting basic blocks to be sufficient for our purposes. By trying to visit multiple interesting basic blocks on a path within a function, overwrites in the very same function are more likely to be detected. This is, for example, the case for inlined constructors: one block may allocate memory, whereas another writes the object's vtable.

Loops are only traversed once, i.e., paths are guaranteed to visit every basic block at most once. Empirically, we did not encounter cases where loop unrolling would have yielded better results in terms of object creation coverage.

Up to now, we only considered paths within a certain function. However, such a path may contain calls to other functions. As already stated in the previous section, the call depth of a path impacts the amount of context available to our analysis, which, in turn, strongly impacts its results. Still, it is also an important factor to ensure that the analysis terminates in a reasonable amount of time and, once again, illustrates how one has to weigh up performance and accuracy. Empirically, we determined that a maximum call depth of 2 is sufficient for analyzing large real-world applications. Note that, depending on time and resources available to an analyst, higher precision can be achieved by increasing this number.

4.3.3 Virtual Callsite Identification

Even though the information collected in these steps is already helpful for an analyst, we further refine the result and try to distinguish vcalls from other indirect call constructs and only resolve target sets for the former. To detect a vcall, *Marx* searches for the virtual function dispatch structure described in Section 2.2.4. As this structure applies to both vcalls and other types of indirect calls (e.g., function pointers), we implemented two modes to resolve the targets: *conservative* and *non-conservative* mode.

In conservative mode, an indirect call is only identified as vcall if the *thisptr* holds a known object and a vtable is involved when computing the target address. This ensures that the analysis has an exact state for the *thisptr*.

In non-conservative mode, an indirect call is considered to be a vcall simply if the vtable is involved in the computation of the target address, i.e., we drop the requirement that the *thisptr* has to be valid. Due to missing context during the analysis (i.e., call depth is depleted), memory locations might not be identified as objects and therefore the *thisptr*

check can fail. The non-conservative mode allows the analysis to resolve more vcalls since requirements are relaxed. However, overestimation can lead to a higher false-positive rate. The difference between both modes is further evaluated in Section 4.5.2.

4.4 Security Applications

Beyond applications in the area of reverse engineering, the results of the reconstructed class hierarchies can also be used to significantly improve defenses that mitigate attacks against C++ applications. In this section, we present two protection approaches build on top of the analysis results provided by *Marx*: *vtable protection* and *type-safe object reuse*. In practice, a C++ application can be analyzed by *Marx* before deployment and then set up with the wanted protection.

4.4.1 VTable Protection for Binaries

VTable protection and, more generally, Control-Flow Integrity (CFI) [1] is a promising way to stop advanced code-reuse attacks. In its ideal form, it limits an attacker by enforcing that each indirect branch can only target valid—as intended by the programmer—code paths. Unfortunately, practical CFI implementations suffer from precision loss when determining the set of valid targets for each branch [44, 100, 135]. Naturally, this also goes for CFI implementations that only protect virtual callsites [75]. Since it is even harder to recover class hierarchies of an application without access to its source code, current state-of-the-art binary-level defenses rely on weak characteristics to narrow down the set of call targets [135]. Examples include looking only at argument count information [158], enforcing that the *vtblptr* has to point to read-only memory [63], or allowing all existing vtables at a vcall [127]. Despite drastically reducing the set of valid targets, these approaches may still leave enough wiggle room for attackers to launch devastating attacks [44, 100, 135].

With the reconstructed class hierarchies, we can extend existing binary-level CFI solutions with a vtable protection. Our goal is to increase CFI guarantees for C++ applications by expanding state-of-the-art defenses with a mechanism to enforce correct class hierarchies for indirect branches. On that account, we extract the index into the vtable for each identified vcall that is used to determine the function entry (as explained in Section 2.2.4). With the help of the class hierarchy, we are then able to generate a *function type* for all virtual functions at this position in the class hierarchy. In our example shown in Figure 2.3 in Chapter 2, functions `A::funcA1`, `B::funcB1`, and `thunk to C::funcB1` would get the same *function type*. Obviously, targeting `A::funcA1` is not allowed at vcalls that are used to branch to `B::funcB1` and `thunk to C::funcB1`, indicating an overestimation of our approach. However, the achieved precision of the call target set is a vast improvement in comparison to existing binary-only vtable protection approaches and it remains to be shown that this small overestimation can be exploited by an adversary [164].

Ideally, the vtable protection would merely insert a label check before each vcall that verifies whether the target is of the same *function type* as the virtual callsite. Since our static analysis is in certain cases not able to precisely assign a hierarchy to each vcall, however, we apply two additional techniques:

- *Dynamic Analysis.* To increase coverage, we run the binary in a controlled environment with trusted input (e.g., by running unit tests). During dynamic analysis, we inspect whether (i) executed indirect calls exhibit characteristics of a vcall, (ii) hierarchies used at the same vcall are merged together, and (iii) detected vcalls are in fact vcalls.

- *Slow Path.* Since our analysis may still miss key information about callsites (e.g., class hierarchy relations, leading to false positives), our extension can enter a slow path when a *function type* check failed (treating the failure as an anomaly rather than breaking the program). This slow path can be used to further investigate the branch and to decide if it is allowed or not.

Moreover, as we are only interested in protecting C++ semantics, our static analysis filters callsites that are definitely not vcalls.

We implemented a binary-only vtable protection on Linux for x86-64, using a similar binary run-time instrumentation model as proposed by Van der Veen et al. [158]: we use Dyninst [17] to move all functions to a protected shadow code region and prepend them with a two-byte *function type* value, as obtained from our static analysis. Next, we instrument each vcall with a short sequence of instructions. These instructions check whether the target's *function type* matches that of the vcall. If not, it enters our slow path, which we implemented by using the *PathArmor* open source CFI framework [156]. Note that we did not implement our own user-space JIT verifier, but rather let the kernel module sleep for 10 ms whenever a new path is found. This is to carefully mimic the behavior of the original *PathArmor*'s implementation with an over-approximation of the average values (Table 3 from the *PathArmor* paper [156]). We also remark that we adopted *PathArmor*'s context-sensitive CFI approach to demonstrate the feasibility of our protection strategy similar to other efforts [103], but our system can incorporate any other solution to operate heavyweight security checks on the slow path.

4.4.2 Type-safe Object Reuse

Typically a process reuses freed memory for new allocations blindly. Attackers can abuse this mechanism by exploiting use-after-free vulnerabilities, where a malicious object is carefully placed in memory that was previously occupied and (dangling) pointers still point to it. Type-safe memory allocators such as Cling [6] aim at reducing this risk by preventing memory chunks of different types from being allocated in the same location. Essentially, type-safe memory (and in our case object) reuse maintains pools (i.e., memory regions) that are used for allocating only a particular type of object. Newly allocated objects are placed only in their own typed pool, and objects with different types cannot share a common memory location in the lifetime of the process. Assuming the class hierarchy of a C++ program is known, types can be defined based on class relations. As a result, we can reduce the attack surface by forcing pointers of similar typed objects to overlap. Unlike Cling [6], we focus only on type-safe C++ object reuse, with object types derived from the recovered class hierarchy. On the other hand, Cling is C++ agnostic in principle and the class hierarchy as reconstructed by *Marx* can significantly improve it in handling C++ allocations. The benefit is to reduce the number of typed pools (and memory usage) and

also avoid expensive instrumentation to derive the run-time type (Cling relies on callstack hashes rather than offline type information).

To demonstrate this concept, we built a type-safe object reuse system based on the class hierarchy exported by *Marx*. Our system consists of two parts: an allocator with type-safe object reuse support and a library to instrument object allocations. The allocator enhances `tcmalloc` with functions that leverage type information to place new objects in type-based pools.

Specifically, in `tcmalloc`, pools are subdivided in alignment pools. For performance reasons, `tcmalloc` keeps track of thread-local pools and one central pool. When a thread-local pool reaches a predefined limit of free pages, it transfers some of the free pages to the central pool. Merging pools can be an issue for typed allocations since different types can end up in the same pool. Notice that new typed allocations must be aligned with past ones. When the allocation size does not divide the page size, it is possible that an object overlaps two pages. If this is the case and at least one of the two pages with the overlapping object are given back to the central pool, we cannot guarantee that following allocations are correctly aligned. Therefore, we do not give any memory back from typed pools if the alignment size class does not divide the size of a page.

Finally, our type-safe object reuse application contains a shared library that instruments all allocations at runtime. The shared library is preloaded with the protected binary to trigger type-based allocations when virtual objects are instantiated and to resolve the actual allocation type based on the available class hierarchy. The type resolution works as follows. We start by preprocessing the analysis of *Marx* to construct triples of the form `(location, size, type)`. Here, `location` is the address of the call-site of `new`, `size` the size given to `new`, and `type` is a unique identifier. Moreover, the type identifier (or *type tag*) is generated by assigning a distinct value to each unique class hierarchy found by *Marx*. At runtime, we load the file containing these triples and store them in a hashtable. This hashtable uses the tuple `(location, size)` as key and the `type` as value. The shared library overrides the `new` and `new[]` operators, so we can infer the type information before dispatching to our typed allocation function in `tcmalloc`. Note that some allocations may be missed. We stress here that our intention is to showcase a prototype based on the exported class hierarchy and not a mature defense. In a real setting, the binary should be rewritten [17] by adding the resolution code to all callsites that construct virtual objects—eliminating the need for any run-time type inference instrumentation.

For each occurring allocation, the size of the allocation is used as the key and the location is computed using the return address. With this information, the type tag is retrieved from the hashtable and passed to the modified allocator function, which maintains a pool per allocated type. When no type exists for the particular location and size combination, a value of zero is returned. The allocator uses this value to choose a fast path, where no typed memory pools are used.

4.5 Evaluation

In this section, we evaluate *Marx* and its applications in terms of performance and accuracy. Unless stated otherwise, all test cases are compiled using GCC 4.8.5. Our test cases

include a variety of real-world applications and shared libraries. Consequently, no alterations to the compiler options specified by a test case have been made, i.e., each program is compiled with the compiler flags intended by the authors. Although *Marx*'s purpose is to analyze closed-source software, our evaluation uses open-source software since otherwise, we are not able to generate a ground truth to compare against. We evaluated our class hierarchy reconstruction and virtual callsite target resolution on Ubuntu 14.04 LTS running on an Intel Core i7-2600 CPU with 16 GB of RAM.

The evaluation testbed for our binary vtable protection and type-safe object reuse implementations is a system with an Intel Core i7-6700K CPU @ 4.00GHz and 16 GB of RAM, running Ubuntu 14.04 LTS with Linux kernel 4.2.0 and transparent huge paging disabled.

4.5.1 Class Hierarchy Reconstruction

The main goal of our framework is to provide an analyst with accurate information about the class hierarchies. Hence, we evaluated the precision of *Marx* by comparing the analysis results with the class hierarchies of the application as reported by the compiler. More specifically, the ground truth is obtained by parsing the RTTI of the target application. Remember that our analysis reconstructs an individual class hierarchy as a set and does not contain information about the direction of inheritance. Hence, the ground truth is also extracted as a set. Table 4.1 shows the accuracy of our class hierarchy reconstruction for various real-world applications and shared libraries. Sizes in the table are given in MiB and are taken from the stripped binaries without debug information.

Overall, we observe that *Marx* is capable of precisely recovering the information about the class hierarchies for many types of applications. We find that the results are better for applications than for shared libraries. For applications, the analysis process was able to correctly reconstruct 84.8% of the hierarchies on average. Only 6.3% are underestimated and also 6.3% of the hierarchies were not found. For shared libraries, on average, 72.3% of the hierarchies were correctly reconstructed, while 8.5% of the hierarchies were underestimated and 11.3% were not found. Consequently, we conclude that *Marx* is able to recover most of the class hierarchies of the target binaries completely and therefore provides helpful information for an analyst. The difference between applications and shared libraries results stem from the fact that the analysis of a shared object misses a lot of context (cf. Section 4.3.1). Shared objects are not written to be executed as a standalone application. Hence, most functions are not called from within the shared object, but only from an application, using the interface exposed by the library.

This is also evident when looking at the time needed to analyze an application in comparison to a shared library. Almost all of the tested shared libraries are analyzed in under a minute. The functions of applications are more connected with each other through calls. Since *Marx* follows these connections and analyzes the called functions within the current context, it needs more time to analyze the whole application. In contrast, shared libraries tend to provide a rather "flat" functionality and do not have so many connected functions. Hence, analyzing them is faster.

The application with the best results is *VboxManage*. *Marx* underestimated only one hierarchy and correctly reconstructed the remaining 32. However, *Marx* also found 9

Table 4.1: Results of the class hierarchy reconstruction analysis. *size* gives the size of the stripped binary in MiB. # *GT* and # *analysis* give the number of hierarchies in the ground truth and found during the analysis, respectively. # *matching* gives the number of hierarchies that are correctly reconstructed. # *overestimated* and # *underestimated* give the number of reconstructed hierarchies that are overestimated and underestimated, respectively. # *not found* gives the number of hierarchies that were not found during the analysis. # *not existing* gives the number of hierarchies that were found during the analysis but do not exist in the ground truth. *time needed* gives the time that the static analysis needs to complete.

Program	*size (MiB)*	*# GT*	*# analysis*	*# matching*	*# overestimated*	*# underestimated*	*# not found*	*# not existing*	*time needed (hh:mm:ss)*
VboxManage 5.0.24	0.97	33	45	32	–	1	–	9	0:06:12
MySQL Server 5.7.11	23.91	78	117	69	1	7	1	–	11:36:17
MongoDB 3.2.4	27.72	158	253	137	–	8	13	63	1:08:41
Node.js 5.10.1	15.18	59	84	55	2	2	–	14	0:33:16
FileZilla 3.13.1 (GCC 4.9)	4.42	21	9	3	6	4	8	1	1:19:59
VboxRT.so 5.0.24	2.27	3	3	2	–	–	1	1	0:00:02
VboxXPCOM.so 5.0.24	1.06	8	14	3	–	2	3	1	0:00:05
libFLAC++.so 6.3.0	0.10	3	3	3	–	–	–	–	0:00:01
libebml.so 1.3.3	0.14	2	2	2	–	–	–	–	0:00:01
libmatroska.so 1.4.4	0.65	2	2	2	–	–	–	–	0:00:17
libmusicbrainz5cc.so 5.1.0	0.56	3	2	1	–	1	1	–	0:00:01
libstdc++.so 6.0.18	0.93	5	24	2	–	2	1	–	0:00:01
libwx_baseu-3.1.so 3.1.0	2.55	33	26	26	–	–	7	–	0:00:47
libwx_baseu_net-3.1.so 3.1.0	0.29	5	7	4	–	1	–	–	0:00:01
libwx_gtk2u_adv-3.1.so 3.1.0	1.94	20	23	17	1	1	1	–	0:00:21
libwx_gtk2u_aui-3.1.so 3.1.0	0.59	7	7	5	1	1	–	–	0:00:01
libwx_gtk2u_core-3.1.so 3.1.0	5.92	41	46	31	6	2	2	1	0:01:17
libwx_gtk2u_html-3.1.so 3.1.0	0.79	5	9	2	2	1	–	–	0:00:06
libwx_gtk2u_xrc-3.1.so 3.1.0	1.06	4	4	2	1	1	–	–	0:00:03

hierarchies that do not exist in the application. Note that non-existing hierarchies are most likely not used in code constructs such as vcalls or object creation at a `new` operator. Hence, in applications such as vtable protection or type-safe object reuse such overestimations have no effect and do not influence the results.

For the largest application, *MongoDB*, *Marx* was able to reconstruct 137 out of 158 hierarchies correctly. Only 8 hierarchies were underestimated and 13 were not found during the analysis. Most of these missing hierarchies are connected via an abstract class which was not referenced in the binary code (most likely due to compiler optimizations) and hence not found during the analysis. For the largest shared library, *libwx_gtk2u_core-3.1.so*, 31 hierarchies were correctly reconstructed. 2 hierarchies were underestimated and only 2 were not found during the analysis.

The application *FileZilla* had to be compiled with GCC 4.9 since it requires support for C++14, which is not available for GCC 4.8. It has the worst results of all test cases, as only 3 out of 21 hierarchies were reconstructed correctly. 6 hierarchies were overestimated during reconstruction, 4 underestimated, and 8 not found at all. A manual evaluation of the underestimated and missing hierarchies yields two reasons for these results: First, most of these hierarchies are connected via classes for which no vtable has been emitted by the compiler, which is why *Marx* cannot leverage them. This is due to optimization passes that remove these vtables from the binary. A detailed discussion is given in Section 4.6. Second, *FileZilla* makes heavy use of the *wxWidgets* library (i.e., the shared objects with *libwx_* prefix in Table 4.1). Some underestimated hierarchies are connected via vtables from these shared objects. Despite *Marx*'s inter-modular data flow ability, it was not able to find a connection between all classes of the underestimated hierarchy with the external ones. A manual investigation revealed that not all classes (despite their connection to an external class according to RTTI) execute a library function that overwrites the *vtblptr*—presumably due to compiler optimizations.

4.5.2 Virtual Callsite Targets

With static analysis, it is hard to determine the target function of an indirect call. As noted earlier, for binaries compiled from C++ code, virtual functions are mostly implemented using indirect call instructions. To assist a reverse engineer, our static analysis hence attempts to resolve the target set of a vcall as accurately as possible. To evaluate the correctness of the analysis, we utilize the *VTV* (Virtual Table Verification) GCC pass [149] to generate the ground truth. *VTV* collects class information at compile time and emits code that verifies each virtual call before (potentially) executing it. Verification is performed by checking the object's vtable against a set of allowed vtables. In essence, this performs a check against a specific class hierarchy. For our ground truth, we extract said information and try to match it to the vcall it guards. As test cases, we evaluated the applications used in Section 4.5.1. Unfortunately, we were unable to compile the applications *MySQL Server* and *VBoxManage* with *VTV*. More specifically, the compiler crashed during the compilation of *MySQL Server* and for *VBoxManage* we were not able to pass the configure script.

Table 4.2 shows the results of the vcall target resolution. Remember that non-conservative mode did not require validity of the *thisptr*, but only a dependency on the

Table 4.2: Results of the virtual callsite resolution. # *GT* and # *analysis* give the number of virtual callsites in the ground truth and the framework's results, respectively. # *correct* gives the number of virtual callsites identified correctly. *identified* gives the value in percent of how many virtual callsites of the ground truth are identified. # *resolved* gives the number of resolved virtual callsite targets for the non-conservative and conservative mode (the latter in parentheses). # *matching* gives the number of resolved targets which match completely with the ground truth. # *overestimated* and # *underestimated* give the number of target sets that are overestimated and underestimated, respectively. # *not existing* gives the number of virtual calllsites resolved that do not exist in the ground truth.

	Finding Virtual Callsites				**Resolving Virtual Callsites**				
Program	# *GT*	# *analysis*	# *correct*	*identified*	# *resolved*	# *matching*	# *overestimated*	# *underestimated*	# *not existing*
VboxManage	✗	✗	✗	✗	✗	✗	✗	✗	✗
MySQL Server	✗	✗	✗	✗	✗	✗	✗	✗	✗
MongoDB	14357	13369	12607	87.8%	736 (589)	159 (91)	550 (471)	27 (27)	0 (0)
Node.js	4925	5591	4879	99.0%	798 (754)	166 (142)	629 (611)	1 (0)	2 (1)
FileZilla	2779	2544	2495	89.7%	226 (210)	3 (3)	56 (48)	167 (159)	0 (0)

vtblptr when calculating the target address. As evident from the table, non-conservative mode is able to resolve more vcalls during the analysis. Furthermore, the false positive rate did not increase significantly.

For the application *Node.js*, only 2 vcalls were wrongly detected in non-conservative mode, whereas only 1 was not found in conservative mode. In turn, the non-conservative mode finds 43 vcalls more compared to conservative mode. All in all, for *Node.js*, the analysis was able to identify 4,879 vcalls correctly, which are 99.0% of all virtual callsites.

The worst results were achieved for *FileZilla*. The analysis was only able to resolve 3 vcalls correctly and most of the remaining resolved vcalls were underestimated. This results from the relatively poor results during class hierarchy reconstruction in comparison to the other applications. Due to missing and underestimated hierarchies, *Marx* underestimates the targets of most of the resolved vcalls. However, 2,495 vcalls were identified correctly, which are 89.7% of all virtual callsites.

Overall, *Marx* is able to support an analyst by providing him with potential target addresses for vcalls. Depending on the precision of the class hierarchy reconstruction, the set of target addresses might be underestimated. However, most of the target sets are overestimated such that the analyst does not miss branches during the analysis. On average, 90.5% of all virtual callsites were identified by *Marx* during the analysis. While the results of the call target resolution are helpful for a reverse engineer, more comprehensive target sets can be obtained by combining our static approach with a dynamic profiling phase (such as in Section 4.4.1).

Table 4.3: Evaluation results for our binary vtable protection implementation. For each binary, the table shows (i) *Binary Instrumentation* details, depicting the number of instrumented *vcalls*, written *labels* and *moved* functions; (ii) *Runtime Statistics*, listing the number of *vcalls executed* at runtime, the number of vcalls for which a matching type was found at the target function (*fastpath*), the number of times the *slowpath* was entered, and the number of *unique* paths that require JIT verification; and (iii) *Normalized Runtime*, listing our vtable protection runtime overhead without verification (*hashing only*) and with a synthetic verification timeout of 10ms per unique path (*+verification*).

	Binary Instrumentation			**Runtime Statistics**				**Normalized Runtime**	
Program	*# vcalls*	*# labels*	*# moved*	*# vcalls executed*	*# fastpath*	*# slowpath*	*# unique*	*hashing only*	*+verification*
MySQL	10,864	8,421	28,971	106,330,186	105,035,488	1,294,698	9	1.145	1.155
Node.js	5,905	5,917	26,751	31,491,929	31,491,918	11	6	1.263	1.265
astar	1	1	96	4,595,981,552	4,595,981,552	0	–	1.031	1.031
dealII	1,434	1,428	7,217	96,751,718	96,751,718	0	–	1.012	1.012
namd	2	3	102	2,016	2,016	0	–	0.999	0.999
omnetpp	706	725	1,949	2,061,547,468	2,061,206,142	341,326	361	1.067	1.083
povray	109	111	1,622	4,704,273,295	4,704,273,295	0	–	1.103	1.103
soplex	497	498	873	1,772,890	1,155,673	617,217	661	1.016	1.086
xalancbmk	9,303	9,340	12,808	8,306,798,756	8,306,260,183	538,573	111	1.264	1.272
geomean	342	350	2,318	91,018,910	86,672,172	0	67	1.096	1.108

4.5.3 VTable Protection

We focus the performance evaluation of our vtable protection implementation on two popular Linux C++ servers and the seven C++ applications found in SPEC. Specifically, we evaluated our binary vtable protection with a cross-platform runtime environment for server-side web applications (Node.js 5.10.1, statically compiled with Google's v8 JavaScript engine) and a database server (MySQL 5.7.11). To benchmark Node.js, we configured the Apache benchmark [146] to issue 250,000 requests with 10 concurrent connections and 10 requests per connection for the default page. To benchmark MySQL, we configured the Sysbench OLTP benchmark [90] to issue 10,000 transactions using a read-write workload.

We evaluated our vtable protection instrumentation using the analysis results from *Marx*. To determine the impact on runtime performance, we measured the time to complete the execution of the benchmarks and compared against the baseline—i.e., the original version of the benchmark with no binary instrumentation applied. Table 4.3 details our results.

As shown in the table, it is evident, considering the massive number of executed virtual calls, that our vtable protection performs surprisingly well—10.8% runtime overhead across all the tested applications (geometric mean). Interestingly, there seems to be no

Table 4.4: Evaluation results for our type-safe object reuse implementation. For each binary, the table shows (i) Marx *Statistics*, depicting the number of extracted `new` calls and the number of different `types`; (ii) *Runtime Statistics*, listing the number of used `types`, calls to `malloc`, `new`, and the `new`-array operator during execution, and (iii) *Normalized Runtime*.

	Marx Statistics		**Runtime Statistics**				**Normalized Runtime**
Program	*# new*	*# types*	*# types*	*# malloc*	*# new*	*# new[]*	*overhead*
MySQL	1,017	47	16	2,705,675	82,225	13	1.009
Node.js	4,675	38	14	7,685,562	12,228,927	9,093,605	1.022
astar	11	0	0	1,008,577	108,037	8	0.999
dealII	1,632	11	5	48	144,642,689	6,616,448	1.016
namd	584	2	1	2	2	1,320	0.999
omnetpp	717	9	1	45,950,697	0	221,218,929	1.028
povray	54	7	6	2,414,075	83	176	0.995
soplex	20	6	2	3,718	3	4	0.997
xalancbmk	2,051	167	46	6,854	135,148,541	158	1.046
geomean	350	6	2	56,309	6,032	3,888	1.012

direct correlation between the number of executed virtual calls and the resulting overhead. SPEC binaries *astar* and *povray*, for example, both execute over 4.5 billion virtual calls—all resolved using *Marx*'s analysis results—but yield fairly different runtime overheads: 3% for *astar*, vs 10% for *povray*, a delta that might be caused by CPU caching behavior. We believe that these results are encouraging: they demonstrate that enforcing vtable protection (or CFI) over *likely* (rather than precise) invariants by using a slow path for second-stage verification is feasible in practice.

4.5.4 Type-safe Object Reuse

To evaluate our type-safe object reuse application, implemented on top of *Marx*, we ran experiments on the same set of applications described in Section 4.5.3. Table 4.4 presents our results. The first two columns contain the number of unique `new` (including `new[]`) callsites and types found by *Marx*'s analysis. Next, we present the number of unique types caught by our library, followed by the number of times `malloc`, `new` and `new[]` were called (either typed or untyped) during the benchmark. Finally, we show the overhead from our library.

We observe a slight speedup for *astar*, *povray* and *soplex*. As can be observed from Table 4.4, the latter two are not heavy users of the `new` and `new[]` operators, and although *astar* performs many calls to `new`, we detected no types during its execution. This is caused by the fact that *astar* does not rely on many C++ features [81]: *Marx* recovered *one* vtable which is never written into a heap object in the program. Thus, the `new` operator is never used with a type.

Our results for the real-world applications Node.js and MySQL are much more realistic compared to the SPEC benchmarks: our type-safe object reuse implementation captures a significant fraction of the C++ types as reconstructed by *Marx*. Although both appli-

cations heavily depend on C++ objects, the overhead imposed by the type-safe object reuse application is low. For example, in Node.js, we recorded more than 21 million new objects, while its normalized runtime is 2.2%. We think that these results are encouraging: type-safe object reuse provides significant security invariants, while our experiments report a performance overhead of less than 5% (geometric mean).

4.6 Discussion

In the following, we discuss the effects of compiler optimizations on our analysis and review several ways to optimize our prototype implementation.

4.6.1 Compiler Optimizations and Lost Information

Even though all of our evaluation results are encouraging, we note that the biggest limitations of our approach are due to compiler optimizations and a loss of information on the binary level. Inherently, *Marx* is dependent on vtables (and references to them) emitted by the compiler. Especially for *abstract* base classes, however, such relations may not be revealed by certain vtable usage patterns; the information is simply missing from the binary and we cannot recover this information. This increases the observable *gap* between the formal class hierarchy as set up by the programmer and the results obtained by *Marx*, based on artifacts found in the (optimized) binary itself.

Such a case was encountered during the evaluation of *FileZilla*. A compiler optimization removed vtables of abstract classes from the binary which were the base classes of complete hierarchies. As a result, the overwrite analysis failed to join the smaller hierarchies. Since the vtables did not have other characteristics that allows our approach to find a connection (e.g., via heuristics discussed previously), the reconstructed hierarchies were either not complete or not found at all. Hence, the quality of our results depends on the size of the gap between the formal and the actual class hierarchy as encoded in the binary.

Other than this, we did not encounter any application-specific idiom that affect the accuracy of our results.

4.6.2 Improving Analysis Contexts

Since our static analysis approach focuses on real-world applications, we had to weigh up precision against performance to be able to scale to complex binaries. Hence, we introduced limiting factors such as the call depth restriction and characteristics that we deem as interesting in a basic block. One problem that may arise with these restrictions is that we may miss important information during our analysis. Consider, for example, a function f_{imp} which yields valuable information for our analysis, but is called from a fixed callsite. In the following, we call such a function an *important* function. When our static analysis processes the function's caller and the basic block that calls f_{imp} is not considered interesting, it is highly likely that no path is generated which ends up in f_{imp}. Hence, our analysis misses context that would be provided by the function and its results loose precision.

A naïve approach to tackle this problem is to consider all call instructions as interesting during the path generation (i.e., follow every call). This, however, does not scale to real-world applications due to the path explosion problem. A better solution is to mark those basic blocks with call instructions as interesting that *eventually* reach important functions. In other words, we recognize *importance* of functions as a transitive function which, in turn, impacts the importance of its callers.

However, *Marx* analyzes the functions on a on-demand basis and does not know if the target of a call instruction is important for the analysis process (i.e., it only takes information *local* to the current function into account). In order to add *global* information about the importance of a function into *Marx*'s decision process, we propose to add a preliminary pre-processing step. In essence, we build a static call graph which allows to propagate information about important basic blocks up to its callers. During path generation, this can affect the decision whether or not to follow an (otherwise uninteresting) call. Additionally, this call graph can be enriched at analysis time to include target sets resolved at a vcall. Further, it allows to dynamically adjust the call depth.

4.6.3 Improving Shared Library Results

As shown in Section 4.5.1, the class hierarchy reconstruction of shared libraries is not as precise as for applications. This is due to the fact that shared libraries are written to be used from other applications or shared libraries. Hence, most functions in a shared library are not called from within the very same module. As a result, *Marx* has to analyze these functions without any context given by the caller (e.g., a vcall is using an object that is provided by the caller). This missing information leads to a lower precision in reconstructing the class hierarchy and fewer vcalls are found. One way to tackle this problem is to analyze the shared library in combination with an application that is using it. Once a function inside a shared library is called from the application, the analysis framework has a context that might help improving the results. However, this does not necessarily cover all exported functions of the shared library. Also, an analyst might not always have an application at hand that is using the shared library that he has to analyze.

4.6.4 Reconstructing RTTI

An interesting application of the class hierarchy reconstruction results is the subsequent reconstruction of RTTI associated with vtables. This information can, in turn, be leveraged by other applications, such as analysis programs or protection mechanisms which are able to perform better when provided with RTTI. Notably, this would be an easy way to incorporate our results in potentially closed-source applications which would not require modifications to the programs themselves. However, since *Marx* is not able to recover the class hierarchies with full precision in the general case, the applications have to be able to cope with a certain amount of imprecision.

Furthermore, RTTI holds information about the inheritance direction. More specifically, it only contains a pointer to the RTTI of parent classes. Currently, our analysis approach is not able to extract the direction of the inheritance. Therefore, the recovered RTTI would contain all classes that are in the same hierarchy and therefore overestimate it.

4.6.5 Improving VTable Protection

As shown in Section 4.5.3, the results of *Marx* can be used for a binary-only CFI implementation focusing on vcalls. However, even with a dynamic profiling phase to improve the results of our static analysis, the slow path of our implementation is still required by some applications, which leads to a relatively high performance overhead. In order to tackle this problem, the implementation can be extended to use the technique proposed by Prakash et al. [127]. If our analysis cannot assign a reconstructed class hierarchy to a given vcall, the CFI implementation can allow all functions at the same offset in any known vtable. This way, the implementation would have two different protection granularities: For vcalls with an assigned class hierarchy, the set of allowed functions lies within the class hierarchy. For vcalls without an assigned class hierarchy, the set of allowed functions lies within the known vtables. Hence, the verification at a vcall without an assigned class hierarchy can also be implemented using a simple label check.

4.7 Related Work

We now review related work on the reconstruction of C++ class hierarchies and discuss how *Marx* advances the field. Most similar to our static analysis approach is the work conducted by Jin et al. [83]. Their approach, called *objdigger*, uses symbolic execution and inter-procedural data flow analysis to discover objects of classes, their attributes, and methods. However, their approach does not reconstruct the class hierarchy and only the ideas to recover it are described in the paper (which are similar to our *vtblptr* overwrite analysis). Furthermore, the evaluation is only done on small test cases with up to 10 classes instead of complex binaries.

The approach presented by Fokin et al. [59] focuses on reconstructing the class hierarchies of C++ programs. Their approach recovers the vtables in memory and analyzes them and their corresponding constructors. However, they focus on analyzing the structure of the vtable size and the usage of pure virtual functions to recover the direction of the inheritance. The data-flow through the program is not considered in their work, leading to a certain imprecision.

Katz et al. [85] proposed an approach to support an analyst that reverse engineers C++ binaries based on machine learning. Their approach outputs a probability that indicates what class is used at a given vcall. This is done by using sequences of instructions that can be assigned to a specific class as a training set. The trained model is then used to estimate other vcalls and the used class, with the goal of giving the analyst a hint where the control flow might go next. Unfortunately, their approach ignores polymorphism and is only able to provide one possible branch target. Additionally, their evaluation was done on small applications (largest one has a size of around 1MB) on a machine with 64 CPUs that took several hours. Therefore, their approach is not able to support an analyst on reverse engineering real-world C++ applications.

The binary analysis framework *angr* is presented by Shoshitaishvili et al. [139]. Their work focuses on re-implementing existing techniques for vulnerability identification in order to compare them with each other. The introduced framework has a modular design and provides the possibility to be extended with new analysis techniques. The presented

algorithms of our approach could also be implemented with *angr* instead of writing an own framework. However, *angr* is written in Python and due to its performance, it is likely not efficient for large real-world binaries such as *Node.js* or *MySQL Server*.

Prakash et al. [127] presented *vfGuard*, a binary-only indirect call protection mechanism for C++ binaries. Their approach tries to protect vcalls by creating a whitelist with valid calll targets. If the target address is not within the whitelist, an attack is assumed and the execution is terminated. The whitelist is determined by the offset into the vtable that is used by the vcall. However, they do not try to recover the class hierarchies because of its difficulty and just allow any vtable at a vcall (with some additional filtering). *T-VIP*, proposed by Gawlik et al. [63], is also a binary-only approach to protect virtual callsites from vtable hijacking attacks. However, they do not recover C++ specific structures such as vtables, but reduce the virtual callsite characteristics to two heuristic policies. The first policy restricts the *vtblptr* to point to read-only memory at a vcall. The second policy checks if a random function pointer in the vtable points to memory that is not writable. Both policies narrow down the ability of an attacker to inject a crafted vtable. However, more advanced code-reuse attacks such as proposed by Schuster et al. [135] are not affected by these policies. Gawlik et al. also proposed a third policy to check if the used vtable resides in an allowed set built with the help of the class hierarchy. However, they did not implement this idea because previous existing work did not show a practicable recovery of class hierarchies for real-world programs.

Most similar to our presented application of a type-safe object reuse is Cling, a work presented by Akritidis [6]. Cling is a type-safe memory allocator used to mitigate use-after-free attacks. It modifies the heap allocation process to provide types for each memory allocation that is made in the application. Cling uses the address of the allocation site and size as a type for its pools. Hence, a use-after-free bug only grants access to the remaining data of the same object type. In contrast, our presented application builds types on the base of the reconstructed class hierarchies. Since Cling is C++ agnostic in principle, the class hierarchy as reconstructed by *Marx* can significantly improve it in handling C++ allocations. The benefit is to reduce the number of typed pools (and memory usage) and also avoid expensive instrumentation for deriving the run-time type. In a similar fashion, *VTPin* [133], a vtable hijacking protection for binaries, which is currenlty class-agnostic, could potentially leverage the extracted hierarchies for increasing the accuracy in collecting pinned vtable pointers.

4.8 Conclusion and Future Work

In this chapter, we presented a practical and efficient approach to reconstruct C++ class hierarchies from a given binary application. Our static analysis follows the data flow and tracks objects through multiple paths through the target binary while taking C++ characteristics into account. Hence, we recognize artifacts resulting from the way compilers implement high-level features such as polymorphism and use them to recover information about the relation of classes in the binary.

We presented the design and implementation of a tool called *Marx* capable of performing the outlined approach and evaluated it on several large, real-world applications.

The results are promising: On average, *Marx* precisely reconstructed 84.6% of the class hierarchies of applications and 73.3% of the class hierarchies of shared libraries. The information provided by our analysis can then be used to resolve the sets of potential target functions of virtual callsites and helps an analyst following control flow even across previously unresolvable indirect calls.

Furthermore, we present two applications built atop of the analysis results: First, an improved *vtable protection* mechanism for binary executables capable of verifying the integrity of the control flow. Second, *type-safe object reuse*, which enhances type-safe memory allocators. We show that based on our results, practical defenses that improve security can be developed even in cases where the extracted class hierarchy is reconstructed imperfectly. Our vtable protection treats violations as anomalies and performs more heavyweight checks on a slow path, hence, trading off on performance to compensate for the imprecision. The presented type-safe object reuse application can tolerate type-to-pool mapping mismatches and thus trades off on security to handle the imprecision. In short, we show that it is possible to build *fully conservative* binary-level defense solutions on top of imprecise information.

Even though the evaluation demonstrates that the analysis results can improve binary-level defenses, a remaining imprecision is left. Improving C++ binary-only protections to close the gap between security guarantees source-code based and binary-only defenses provide is subject in the next chapter.

In terms of future work, it would be interesting to explore if the direction of the class relations can be recovered. To this end, the analysis could leverage the order in which constructors and deconstructors overwrite *vtblptrs*. However, it remains to be seen if the order of write operations is sufficient to reconstruct the directions of an entire class hierarchy and if other C++ characteristics exist that contain information about the class relations. Additionally, this advancement would allow reconstructing RTTI in the binary executable, which would be an easy way to make the results usable by other analysis tools.

Chapter 5

Excavating C++ Constructs from Binaries to Protect Dynamic Dispatching

Software implemented in the C++ language is vulnerable to increasingly sophisticated memory corruption attacks [28, 39, 67, 135, 155, 157]. C++ is often the language of choice for complex software because it allows developers to structure software by encapsulating data and functionality in *classes*, simplifying the development process. Unfortunately, the binary-level implementations of C++ features such as polymorphism and inheritance are vulnerable to control-flow hijacking attacks, most notably *vtable hijacking*. This attack technique abuses common binary-level implementations of C++ virtual methods where every object with virtual methods contains a pointer to a *virtual function table* (*vtable*) that stores the addresses of all the class's virtual functions. To call a virtual function, the compiler inserts an indirect call through the corresponding vtable entry (a *virtual callsite*). Using temporal or spatial memory corruption vulnerabilities such as arbitrary write primitives or use-after-free bugs, attackers can overwrite the vtable pointer. Subsequent virtual calls then use addresses in an attacker-controlled alternative vtable, which results in a hijacked control flow. In practice, vtable hijacking is a common exploitation technique widely used in exploits that target complex applications written in C++, such as web browser and server applications [148].

Control-Flow Integrity (CFI) solutions [1, 19, 114, 125, 149, 156, 158] protect indirect calls by verifying that control flow is consistent with a Control-Flow Graph (CFG) derived through static analysis. However, most generic CFI solutions do not take C++ semantics into account and leave the attacker with enough wiggle room to build an exploit [67, 135]. Consequently, approaches that specifically protect virtual callsites in C++ programs have become popular. If source code is available, compiler-level defenses can benefit from the rich class-hierarchy information available at the source level [21, 26, 149, 164]. However, various legacy applications are still in use [115], or proprietary binaries have to be protected, which do not offer access to the source code (e.g., Adobe Flash [4]). Here, binary-level defenses [55, 63, 127, 165] must rely on (automated) binary analysis techniques to reconstruct the information needed to guarantee security and correctness. As Chapter 4 has shown,

these analysis results improve the security of a program significantly, but imprecisions of the analysis still leaves wiggle room for an attacker.

5.1 Introduction

To improve binary-only defenses against vtable-hijacking attacks and to further close the gap between security guarantees source-code based and binary-only defenses provide, we present *VTable Pointer Separation* (VPS). Unlike previous binary-only defenses against vtable-hijacking attacks that restrict the set of vtables permitted for each virtual callsite, we check that the vtable pointer remains unmodified after object creation. Intuitively, VPS checks the vtable pointer's integrity at every callsite. Because the vtable pointer in a legitimate live object never changes and the virtual callsite uses it to determine its target function, VPS effectively prevents vtable-hijacking attacks. In essence, we want to bring a defense as powerful as *CFIXX* [26] (which operates at the source level) to binary-only applications, even though none of the information needed for the defense is available. Our approach is suitable for binaries because, unlike other binary-level solutions, we avoid the inherent inaccuracy in binary-level CFG and class-hierarchy reconstruction. Because VPS allows only the initial virtual pointer(s) of the object ever to exist, we reduce the attack surface even compared to hypothetical implementations of prior approaches that statically find the set of possible vcall targets with perfect accuracy.

Given that binary-level static analysis is challenging and unsound in practice, and may lead to false positives in identifying virtual callsites, we carefully deal with such cases by over-approximating the set of callsites and implementing an (efficient) slow path to handle possible false positives at runtime. Meanwhile, VPS handles all previously verified callsite with highly-optimized fast checks. This approach allows us to prevent false positives from breaking the application as they do in existing work [55, 63, 127, 165]. Additionally, while existing work [83, 85, 86] (as well as *Marx* in Chapter 4) only considers *directly* referenced vtables, compilers also generate code that references vtables *indirectly*, e.g., through the Global Offset Table (GOT). VPS can find all code locations that instantiate objects by writing the vtable, including objects with indirect vtable references.

Our prototype of VPS is precise enough to handle complex, real-world C++ applications such as MongoDB, MySQL server, Node.js, and all C++ applications contained in the SPEC CPU2006 and CPU2017 benchmarks. Compared to the source-code based approach *VTV*, which is part of GCC [149], we can on average correctly identify 97.8% and 97.4% of the virtual callsites in SPEC CPU2006 and SPEC CPU2017, with a precision of 95.6% and 91.1%, respectively. Interestingly, our evaluation also revealed 86 virtual callsites that are *not* protected by *VTV*, even though it has access to the source code. A further investigation with the help of the *VTV* maintainer showed that this is due to a conceptual problem in *VTV*, which requires non-trivial engineering to fix. Compared to the source-code based approach *CFIXX*, VPS shows an accuracy of 99.6% and 99.5% on average for SPEC CPU2006 and CPU2017 with a precision of 97.0% and 96.9%. These comparisons show that VPS's binary-level protection of virtual callsites closely approaches that of source-level solutions. While this still leaves a small attack window, it further

closes the gap between binary-only and source-level approaches making vtable-hijacking attempts mostly impractical.

Compared to state-of-the-art binary-level analysis frameworks like *Marx*, VPS' analysis identifies 26.5% more virtual callsites in SPEC CPU2017 and thus offers improved protection. VPS induces geomean performance overhead of 9% for all C++ applications in SPEC CPU2017 and 11% for SPEC CPU2006, which is slightly more than *Marx* induces but with significantly better protection.

Contributions We provide the following contributions:

- We present VPS, a binary-only defense against vtable-hijacking attacks that sidesteps the imprecision problems of prior work on this topic. The key insight is that vtable pointers only change during initialization and destruction of an object (never in between), a property that VPS can efficiently enforce.
- We develop an instrumentation approach that is capable of handling false positives in the identification of C++ virtual callsites, which would otherwise break the application and which most existing work ignores. Unlike prior work, we also handle indirect vtable references.
- Our evaluation shows that our binary-level instrumentation protects nearly the same number of virtual callsites as the source-level defenses *VTV* and *CFIXX*. In addition, our evaluation uncovered a conceptual problem causing false negatives in *VTV* (part of GCC).

The prototype implementation of VPS and the data we used for the evaluation are available under an open-source license at `https://github.com/RUB-SysSec/VPS`.

Outline This chapter is structured in the following way: Section 5.2 gives an overview of the assumed threat model and how VPS works on a high-level. Section 5.3 describes the analysis and instrumentation approach necessary to protect the binary executable with VPS. The implementation of VPS is described in Section 5.4, whereas Section 5.5 presents the evaluation. We then discuss the presented approach in Section 5.6 and give an overview of related work in Section 5.7. Finally, we conclude this chapter in Section 5.8 and discuss possible topics for future work.

The chapter is based on research published at the Annual Computer Security Applications Conference (ACSAC) 2019 [121]. It was performed together with Victor van der Veen, Dennis Andriesse, Erik van der Kouwe, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos.

5.2 Overview

In this section, we give an overview of VPS. We start by defining the thread model it protects against and, subsequently, we explain on a high-level how VPS works.

5.2.1 Threat Model: VTable Hijacking Attacks

As we explained in Section 2.2.4, virtual callsites use the *vtblptr* to extract the pointer to the called virtual function. Since the object that stores the *vtblptr* is dynamically created during runtime and resides in writable memory, an attacker can overwrite it and hijack the control flow at a virtual callsite.

The attacker has two options to hijack an object, depending on the available vulnerabilities: leveraging a vulnerability to overwrite the object directly in memory, or using a dangling pointer to an already-deleted object by allocating attacker-controlled memory at the same position (e.g., via a use-after-free vulnerability). In the first case, the attacker can directly overwrite the object's *vtblptr* and use it to hijack the control flow at a vcall. In the second case, the attacker does not need to overwrite any memory; instead, the vulnerability causes a virtual callsite to use a still existing pointer to a deleted memory object. The attacker can control the *vtblptr* by allocating new memory at the same address previously occupied by the deleted object.

We assume the attacker has an arbitrary memory read/write primitive, and that the $W \oplus X$ defense is in place as well as the vtables reside in read-only memory. These are standard assumptions in related work [1, 55, 149, 165]. The attacker's goal is to hijack the control flow at a virtual callsite (forward control-flow transfer). Attacks targeting the backward control-flow transfer (e.g., return address overwrites) can be secured, for example, by shadow stacks which are orthogonal to VPS and thus out of scope. Furthermore, data-only attacks are also out of scope.

5.2.2 VTable Pointer Separation

Our approach is based on the observation that the *vtblptr* is only written during object initialization and destruction and cannot legitimately change in between. Therefore, only the *vtblptr* that is written by the constructor (or destructor) is a valid value. If a *vtblptr* changes between the object was created and destroyed, a vtable-hijacking attack is in progress. Since these attacks target virtual callsites, it is sufficient to check at each virtual callsite if the *vtblptr* written originally into the object still resides there.

Figure 5.1 depicts the differences between a traditional application and a VPS-protected application. The traditional application initializes an object and uses a vcall and the created object to call a virtual function. As explained in Section 2.2.4, the application uses the vtable to decide which virtual function to execute. If an attacker is able to corrupt the object between the initialization and vcall, she can place her own vtable in memory and hijack the control flow. In contrast, the VPS-protected application adds two additional functionalities to the executed code. While the object is initialized, it stores the *vtblptr* in a safe memory region. Before a vcall, it checks if the *vtblptr* in the object is still the same as the one stored for the object in the safe memory region. The vcall is only executed when the check succeeds. As a result, the same attacker that is able to corrupt the object in between can no longer hijack the control flow. The same concept holds for *vtblptr*s written in the destructors. The *vtblptr* is written into the object and used for vcalls during its destruction (if it is used at all). Since a VPS-protected application stores the written *vtblptr* into the safe memory region and checks the integrity of the one in the

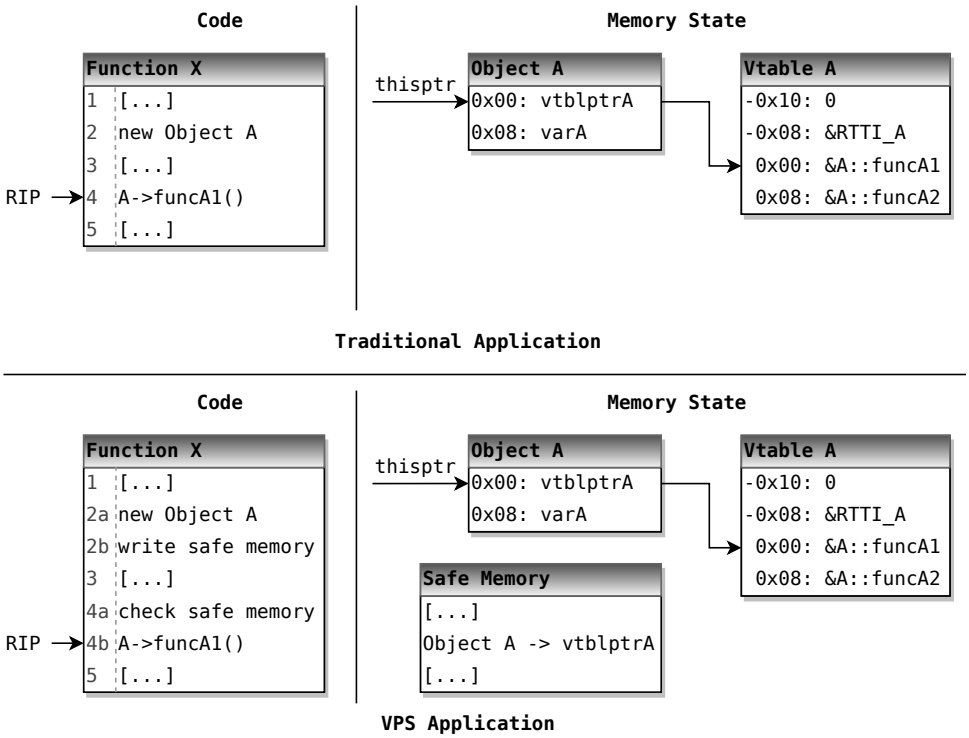


Figure 5.1: High-level overview of the object instantiation and virtual callsite of a traditional application (top) and a VPS protected application (bottom). For both applications the memory state is given while the instruction pointer executes the function call.

object if it is used at a vcall, the approach does not need to differentiate between object initialization and destruction.

In contrast to other binary-only defenses for virtual callsites [55, 63, 127, 165] (as well as *Marx*'s *VTable Protection* presented in Section 4.4.1) that allow a specific overestimated set of classes at a virtual function dispatch, VPS has a direct mapping between an object initialization site and the reachable vcalls.

Even though VPS looks conceptually similar to *CFIXX*, adding this protection at the binary level encounters multiple hurdles. Performing accurate analysis at the binary level is a challenging problem, especially with regards to object creation sites, where false negatives would break the protected application. Our analysis has to take direct and indirect vtable accesses into account, which do not exist on the source level. The virtual callsite identification has to be as precise as possible in order to provide a high level of security and it has to be performed without type information. Any false positive in this result breaks the application, which makes an instrumentation capable of handling these necessary (a problem that other binary-only approaches do not consider).

5.3 Approach

VPS protects binary C++ applications against control-flow hijacking attacks at virtual callsites. To this end, we first analyze the binary to identify C++-specific properties and then apply instrumentation to harden it.

We divide the analysis into three phases: *Vtable Identification*, *Vtable Pointer Write Operations*, and *Virtual Callsite Identification*. At a high-level, our analysis first identifies all vtables in the target binary in the Vtable Identification phase. Subsequently, the identified vtables are used to find all locations in the binary that write *vtblptr*s. Eventually, the identified vtables are also used to identify and verify vcalls in the Virtual Callsite Identification phase. While the Vtable Identification static analysis is an improved and more exact version of the one presented in Section 4.2.1 (finding vtables in `.bss` and GOT, considering indirect referencing of vtables), the other analyses are novel to VPS.

Our approach protects virtual callsites against control-flow hijacking attacks by instrumenting the application using the results from the analysis phase. We instrument two parts of the program: *Object Initialization and Destruction* and *Virtual Callsites*.

In the remainder of this section, we explain the details of our analysis and instrumentation approach. Note that we focus on Linux x86-64 binaries that use the Itanium C++ ABI [60]. However, our analysis approach is conceptually mostly generic and with additional engineering effort can be applied to other architectures and ABIs as well. For architecture-specific steps in our analysis, we describe what to modify to port the step to other architectures.

5.3.1 Analysis: Vtable Identification

To protect *vtblptr*s in objects, we need to know the location of all vtables in the binary. To find these, our static analysis searches through the binary and uses a set of rules to identify vtables. Whenever all rules are satisfied, the algorithm identifies a vtable. Figure 2.3 in Chapter 2 shows a typical vtable structure. The smallest possible vtable in the Itanium C++ ABI [60] consists of three consecutive words (*Offset-to-Top*, *RTTI*, and *Function-Entry*). We use the following five rules to determine the beginning of a vtable:

R-1 In principle, our algorithm searches for vtables in read-only sections such as `.rodata` and `.data.rel.ro`. However, there are exceptions to this. If a class has a base class that resides in another module and the compiler uses copy relocation, the loader will copy the vtable into the `.bss` section [65]. Additionally, vtables from other modules can be referenced through the Global Offset Table (GOT), e.g., in position-independent code [150]. To handle these cases where the vtable data lies outside the main binary, we parse the binary's dynamic symbol table and search for vtables that are either copied to the `.bss` section or referenced through the GOT. Note that we do not rely on debugging symbols, only on symbols that the loader uses, which cannot be stripped.

R-2 Recall that the *vtblptr* points to the first function entry in a class's vtable, and is written into the object at initialization time. Therefore, our algorithm looks for code patterns that reference this first function entry. Again, there are special cases to handle.

The compiler sometimes emits code that does not reference the first function entry of the vtable, but rather the first metadata field at offset `-0x10` (or `-0x18` if virtual inheritance is used). This happens for example in position-independent code. To handle these cases, we additionally look for code patterns that add `0x10` (or `0x18`) to the reference before writing the *vtblptr* into the object, which is necessary to comply with the Itanium C++ ABI [60]. Our algorithm also checks for the special case where vtables are referenced through the GOT instead of directly.

R-3 As depicted in Figure 2.3 in Chapter 2, the *Offset-to-Top* is stored in the first metadata field of the vtable at offset `-0x10`. In most cases this field is `0`, but when multiple inheritance is used, this field gives the distance between the base *vtblptr* and the sub-*vtblptr* in the object (see Section 2.2.2). Our algorithm checks the sanity of this value by allowing a range between `-0xFFFFFF` and `0xFFFFFF`, as proposed by Prakash et al. [127].

R-4 The RTTI field at offset `-0x8` in the vtable, which can hold a pointer to RTTI metadata, is optional and usually omitted by the compiler. If omitted, this field holds `0`; otherwise, it holds a pointer into the `data` section or a relocation entry if the class inherits from another class in a shared object.

R-5 Most of the vtable consists of function entries that hold pointers to virtual functions. Our algorithm deems them valid if they point into any of the `.text`, `.plt`, or `.extern` sections of the binary, or are relocation entries.

Abstract classes are an edge case. For each virtual function without implementation, the vtable points to a special function called *pure_virtual*. Because abstract classes are not meant to be instantiated, calling *pure_virtual* throws an exception. Additionally, the first function entries in a vtable can be `0` if the compiler did not emit the code of the corresponding functions (e.g., for destructor functions). To cope with this, we allowed `0` entries in the beginning of a vtable for *Marx*'s analysis in Section 4.2.1. For VPS, we omit this rule because it can safely ignore the instantiation of abstract classes, given that *vtblptr*s for abstract classes are overwritten shortly after object initialization.

In case of multiple inheritance, we do not distinguish between vtables and sub-vtables. That is, in the example in Figure 2.3 in Chapter 2, our approach identifies *Vtable C* and *Sub-Vtable C* as separate vtables. As discussed later, this does not pose any limitations for our approach given our focus on *vtblptr* write operations (as opposed to methods that couple class hierarchies to virtual callsites).

The combination of multiple inheritance and copy relocation poses another edge case. In copy relocation, the loader copies data residing at the position given by a relocation symbol into the `.bss` section without regards to the type of the data. For classes that use multiple inheritance, the copied data contains a base vtable and sub-vtable(s), but the corresponding relocation symbol holds only information on the beginning and length of the data, not the vtable locations. To ensure that we do not miss any, we identify every `8`-byte aligned address of the copied data as a vtable. For example, if the loader copies a data chunk of `0x40` bytes to the address `0x100`, we identify the addresses `0x100`, `0x108`,

`0x110`, ... up to `0x138` as vtables. While this overestimates the set of vtables, only the correct vtables and sub-vtables are referenced during object initialization.

Note that on other architectures, the assumed size of 8-byte per vtable entry as used by our rules may have to be adjusted. For example, Linux on x86 (32-bit) and ARM would use 4-byte entries, with no conceptual changes.

5.3.2 Analysis: Vtable Pointer Write Operations

The next phase of our static analysis is based on the observation that to create a new object, its *vtblptr* has to be written into the corresponding memory object during the initialization. This is done in the constructor of the class which can be either an explicit function or inlined code. The same holds for object destruction by the corresponding destructor function. Hence, the goal of this analysis step is to identify the exact instruction that writes the *vtblptr* into the memory object. This step is Linux-specific but architecture-agnostic.

First, we search for all references from code to the vtables identified in the previous step. Because vtables are not always referenced directly, the analysis searches for the following different reference methods:

1. A direct reference to the start of the function entries in the vtable. This is the most common case.

2. A reference to the beginning of the metadata fields in the vtable. This is mostly used by applications compiled with position-independent code (e.g., MySQL server which additionally uses virtual inheritance).

3. An indirect reference through the GOT. Here, the address to the vtable is loaded from the GOT.

Starting from the identified references, we track the data flow through the code (using Static Single Assignment (SSA) form [45]) to the instructions that write the *vtblptr*s during object initialization or destruction. We later instrument these instructions, adding code that stores the *vtblptr* in a safe memory region. Our approach is agnostic to the location the C++ object resides in (i.e., heap, stack, or global memory). Furthermore, since we focus on references from code to the vtables, our approach can handle explicit constructor functions as well as inlined constructors and destructors.

During our research, we encountered functions with inlined constructors where the compiler emits code that stores the *vtblptr* temporarily in a stack variable to use it at multiple places in the same function. Therefore, to ensure that we do not miss any *vtblptr* write instructions, our algorithm continues to track the data flow even after a *vtblptr* is written into a stack variable. Because we cannot easily distinguish between a temporary stack variable and an object residing on the stack, our algorithm also assumes that the temporary stack variable is a C++ object. While this overestimates the set of C++ objects, it ensures that we instrument all *vtblptr* write instructions, making this overapproximation comprehensive.

5.3.3 Analysis: Virtual Callsite Identification

Because VPS specifically protects vcalls against control-flow hijacking, we first have to locate them in the target binary. Hence, we have to differentiate between vcalls and normal C-style indirect call instructions. We follow a two-stage approach to make this distinction: we first locate all possible vcall candidates and subsequently verify them. The verification step consists of a static analysis component and a dynamic one. In the following, we explain this analysis in detail.

5.3.3.1 Virtual Callsite Candidates

To find virtual callsite candidates, we use a similar technique as previous work [55, 63, 127, 165]. We search for the vcall pattern described in Section 2.2.4, where the *thisptr* is the first argument (stored in the `RDI` register on Linux x86-64) to the called function and the vcall uses the *vtblptr* to retrieve the call target from the vtable. Note that the *thisptr* is also used to extract the *vtblptr* for the call instruction. A typical vcall looks as follows:

```
mov RDI, thisptr
mov vtblptr, [thisptr]
call [vtblptr + offset]
```

Note that these instructions do not have to be consecutive in the application, but can be interspersed with other instructions. Two patterns can be derived from this sequence: the first argument register always holds the *thisptr*, and the call instruction target can be denoted as `[[`*thisptr* `] + offset]`, where `offset` can be `0` and therefore omitted. This specific dependency between call target and first argument register is rare for non-C++ indirect calls. With the help of the SSA form, our algorithm traces the data flow of the function. If the previously described dependency is satisfied, we consider the indirect call instruction a *vcall candidate*.

Note that the same pattern holds for classes with multiple inheritance. As described in Section 2.2.4, when a virtual function of a sub-vtable is called, the *thisptr* is moved to the position in the object where the sub-vtable resides. Therefore, the first argument holds *thisptr* `+ distance`, and the call target `[[`*thisptr* `+ distance] + offset]`. This still satisfies the aforementioned dependency between first argument and call target. Furthermore, the pattern also applies to Linux ARM, Linux x86, and Windows x86-64 binaries, requiring only a minor modification to account for the specific register or memory location used for the first argument on the platform (`R0` for ARM, the first stack argument for Linux x86, and `RCX` for Windows x86-64).

To effectively protect vcalls, it is crucial to prevent false-positive vcall identifications, as these may break the application during instrumentation. This is also required for related work [55, 63, 127, 165]. While the authors of prior approaches report no false positives with the above vcall identification approach, our research shows that most larger binary programs do indeed contain patterns that result in indirect calls being wrongly classified as virtual callsites.

A possible explanation for the lack of false positives in previous work is that most prior work focuses on Windows x86 [63, 127, 165], where the calling conventions for vcalls and

other call instructions differ. That is, on Windows x86, the *thisptr* is passed to the virtual function via the `ECX` register (*thiscall* calling convention), while other call instructions pass the first argument via the stack (*stdcall* calling convention) [58]. This is not the case for Windows x86-64 and Linux (x86 and x86-64). On these architectures, the *thisptr* is passed as the first argument in the platform's standard calling convention (*Microsoft x64*, *cdecl* and *System V AMD64 ABI*, respectively). While Elsabagh et al. [55], who work on Linux x86, did not report false positives, our evaluation does show false positives in the same application set. We contacted the authors, but they could not help us find an explanation for these differing outcomes and could not give us access to the source code to allow us to reproduce the results.

5.3.3.2 Virtual Callsite Verification

Because a single false positive can break our approach, the next phase in our static analysis verifies the virtual callsite candidates. Basically, we perform a data-flow analysis in which we track whether a *vtblptr* is used at a virtual callsite candidate. If the candidate uses the *vtblptr* to determine the call target, we consider it as verified. However, a data-flow graph alone is not sufficient to verify this connection. The control flow and actual usage of the *vtblptr* have also to be considered. Figure 5.2 depicts an overview of the analysis process. The following describes our analysis in detail.

Data-Flow Graphs First, our analysis tracks the data flow backwards with the help of SSA form starting from all vtable references in the code (which create the *vtblptr*). The data flow is tracked over function boundaries when argument registers or the return value register `RAX` are involved. This means the tracking is done interprocedurally. The same data-flow tracking is done for the call target of each virtual callsite candidate. As Figure 5.2 a) shows, we obtain data-flow graphs showing the source of the data used by the vtable-referencing instructions and the virtual callsite candidates. Whenever a data-flow graph for a virtual callsite candidate has the same data source as a vtable-referencing instruction, we group them together as depicted in Figure 5.2 b).

Control-Flow Path Virtual callsite candidates and vtable-referencing instructions that share the same data source represent a possible connection between a created *vtblptr* and a corresponding vcall. However, this connection alone does not give any information on whether the *vtblptr* is actually used at the virtual callsite candidate. To verify this, we have to check if a control-flow path exists that starts at the data-source instruction, visits the vtable-referencing instruction, and ends at the vcall instruction. For this, our analysis searches all possible data-flow paths through the graph that start at a data-source instruction and end in a vtable-referencing instruction. Additionally, all data-flow paths through the graph are identified that start at a data-source instruction and end at a virtual callsite candidate. Then, they are split into common and unique parts as Figure 5.2 c) depicts.

Next, our analysis tries to transform these data-flow paths into a control-flow path by translating each data-flow node into the basic block that contains the corresponding instruction (see Figure 5.2 d)). With the help of the Control-Flow Graph (CFG), our

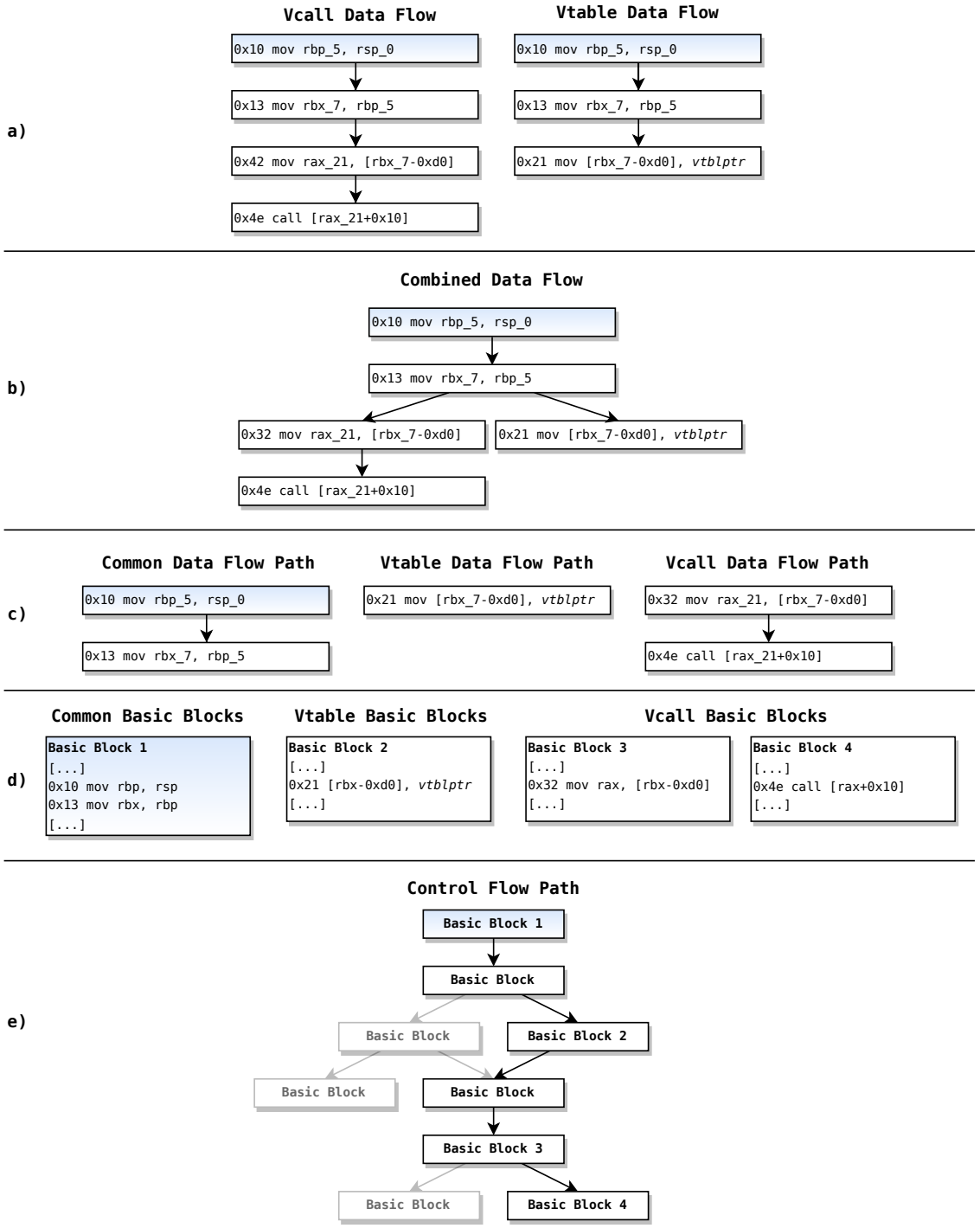


Figure 5.2: Data-flow and control-flow analysis of our vcall verification phase. Step *a)* shows the data-flow graph in SSA form, with the starting node in blue (data source). Step *b)* combines the data-flow graphs of *a)*. Step *c)* divides the paths through the data-flow graph into three components. Step *d)* shows the basic blocks corresponding to the data-flow paths. Step *e)* shows a path through the CFG containing all previously identified basic blocks.

analysis then searches for a path from basic block to basic block until it reaches the final block as Figure 5.2 e) shows. Eventually, if a path exists, the algorithm finds a possible control-flow path that starts from the data-source instruction, visits the vtable-referencing instruction, and ends at the vcall instruction.

Symbolic Execution As a last step, we symbolically execute the obtained control-flow paths to track the flow of the *vtblptr* through the binary. When an instruction writes a vtable into the memory state, we replace that *vtblptr* with a symbolic value. To keep the analysis scalable to large real-world applications, our symbolic execution simply executes basic blocks without checking whether branches can actually be taken in a concrete execution. If a basic block contains a call instruction that is not part of our original data-flow path, we simply execute a return instruction immediately after the call instead of symbolically executing the called function. When the symbolic execution reaches the vcall instruction, we check the obtained memory state to verify that the *vtblptr* is used for the call target. If so, we conclude that the vcall candidate is in fact a vcall and consider it a *verified vcall*.

In addition to explicit vtable-referencing instructions, this analysis phase checks implicit vtable references as well. In case the earlier backward data-flow analysis shows that a vcall target stems from the first argument register, we check whether the calling function is a known virtual function (by checking whether the function resides in any previously identified vtable). If it is, we add a special virtual function node to the data-flow graph. We then search for a path from this virtual function node to the vcall instruction. If a path is found, we apply the steps described previously for transforming the data-flow path to a control-flow path. For such paths, before starting the symbolic execution, we add an artificial memory object containing the *vtblptr* and place the *thisptr* in the first argument register. This way, we simulate an implicit use of the vtable through the initialized object.

We perform the whole vcall verification analysis in an iterative manner. When the data-flow tracking step stops at an indirect call instruction, we repeat it as soon as our analysis has verified the indirect call as a vcall and has therefore found corresponding vtables for resolving the target. The same applies to data-flow tracking that stops at the beginning of a virtual function (because no caller is known). As soon as we can determine a corresponding vcall instruction, we repeat the analysis. The analysis continues until we reach a fixed point where the analysis fails to find any new results.

5.3.3.3 Dynamic Profiling

Our approach includes a dynamic profiling phase that further refines the vcall verification. During this phase, we execute the application with instrumentation code added to all virtual callsite candidates (only the vcall candidates, not the already verified vcalls). Whenever the execution reaches a vcall, the instrumentation code verifies that the first argument contains a valid *thisptr*. To verify this, we check if the first element of the object the *thisptr* points to contains a valid pointer to a known vtable (*vtblptr*). If it does, we consider the vcall verified. Otherwise, we regard the vcall as a false positive of the static analysis and discard it.

Because this phase only instruments vcall candidates identified by the static analysis described in Section 5.3.3.1, it is safe to assume the dependency between first argument and call instruction target. Hence, the above dynamic profiling check is sufficient to remove false positives seen during the profiling run, given that the odds of finding a C-style indirect callsite with such a distinctive pattern that uses C++ objects is extremely unlikely. We did not encounter any such case during our comprehensive evaluation. Also note, that only this dynamic analysis step discards vcall candidates as false positives. Vcalls that could not be verified by the static analysis (or not reached during this dynamic profiling) are still considered vcall candidates since the reason for the failed verification can be missing information (e.g., analysis gaps through indirect control-flow transfers).

5.3.4 Instrumentation: Object Initialization and Destruction

We use the data collected in Section 5.3.2 to instrument object initialization, specifically the instruction that writes the *vtblptr* into the object. When an object is created, the instrumentation code stores a key-value pair that uses the memory address of the object as the *key* and maps it to the *vtblptr*, which is the associated *value*. To prevent tampering with this mapping, we store it in a safe memory region.

Recall that during the creation of a C++ object whose class inherits from another class, the initialization code first writes the *vtblptr* of the base class into the object, which is then overwritten by the *vtblptr* of the derived class. Our approach is agnostic to inheritance and simply overwrites the *vtblptr* in the same order (because each *vtblptr* write instruction is instrumented).

Similarly, our approach is agnostic to multiple inheritance, because object initialization sites use the address where the *vtblptr* is written as the object address. As explained in Section 2.2.4, at a virtual callsite the *thisptr* points to the address of the object the used *vtblptr* resides in. For a sub-vtable, this is not the beginning of the object, but an offset somewhere in the object (in our running example in Figure 2.3 in Chapter 2 offset `0x10`). Because this is exactly the address that our approach uses as the key for the safe memory region, our approach works for multiple inheritance without any special handling.

Since this instrumentation only focuses on *vtblptr* write instructions, it is also agnostic to object initialization and destruction. Hence, we do not have to differentiate between constructor and destructor and can use it for both.

Moreover, despite the fact that we ignore object deletion, our approach does not suffer from consistency problems. This is because, when an object is deleted and its released memory is reused for a new C++ object, the instrumentation code for the initialization of this new object automatically overwrites the old value in the safe memory region with the current *vtblptr*.

5.3.5 Instrumentation: Virtual Callsites Instrumentation

Because a single false-positive virtual callsite can break the application, we designed the vcall instrumentation code such that it can detect false positives and filter them out. In doing so, the vcall instrumentation continuously refines the previous analysis results. The

vcall instrumentation consists of two components, described next: *Analysis Instrumentation* and *Security Instrumentation*.

5.3.5.1 Analysis Instrumentation

We add analysis instrumentation code to all vcall candidates that we were unable to verify during our static vcall verification and dynamic profiling analysis. For verified vcall sites, we only add security instrumentation and omit the analysis code.

Before executing a vcall candidate, the analysis instrumentation performs the same check as the dynamic profiling phase described in Section 5.3.3.3. If the check fails, meaning that this is not a vcall but a regular C-style indirect call, we remove all instrumentation from the call site. If the check succeeds, we replace the analysis instrumentation with the more lightweight security instrumentation for verified virtual callsites described in Section 5.3.5.2, and immediately run the security instrumentation code.

Through our use of adaptive instrumentation, our approach is able to cope with false positives and further refine the analysis results during runtime. By caching the refined results on disk, we can reuse these in later runs of the same application, improving VPS's performance over time. Furthermore, caching also improves the security of our adaptive instrumentation as we discuss in Section 5.6.2.

Because the analysis instrumentation verifies all remaining vcall candidates for false positives at runtime, the static vcall verification from Section 5.3.3.2 and the dynamic profiling from Section 5.3.3.3 can be omitted. Omitting these steps does not affect the correctness of our approach, although we recommend using them for optimal performance.

5.3.5.2 Security Instrumentation

We protect verified vcall sites against control-flow hijacking by adding security instrumentation code that runs before allowing the vcall. The instrumentation uses the *thisptr* in the first argument register to retrieve the *vtblptr* stored for this object in the safe memory region. To decide whether to allow the vcall, the instrumentation code compares the *vtblptr* from the safe memory region with the one stored in the actual object used in the vcall. If they are the same, the instrumentation allows the vcall. If not, we terminate with an alert.

5.4 Implementation

Based on the approach from Section 5.3, we integrated our static analysis into the open source *Marx* framework presented in Chapter 4. This framework provides a basic symbolic execution based on the VEX-IR from the Valgrind project [51] and data structures needed for C++ binary analysis. It is written in C++ and targets Linux x86-64 (amd64) binaries. To support integration of our approach into the *Marx* framework, we added support for SSA and a generic data-flow tracking algorithm.

Because the VEX-IR supports multiple architectures, the framework is easily extendable to these. The same is true for our approach, which is mostly independent from the underlying architecture (Section 5.3). To balance precision and scalability, the symbolic

execution emulates only a subset of the 64-bit VEX instructions that suits our focus on vtable-centered data-flow tracking in real-world applications.

We use IDAPython [79] for vtable identification and CFG extraction. Additionally, we use instruction data provided by IDA Pro [78] to support the SSA transformation, and use Protocol Buffers [70] to export the results in a programming language–agnostic format. We implement dynamic profiling with Pin [104]. We build the runtime component of VPS on top of Dyninst v9.3.2 [17]. Dyninst is responsible for installing *vtblptr* write and (candidate) virtual callsite hooks. We inject these wrappers into the target program's address space by preloading a shared library.

To set up the safe memory region, our preloaded library maps the lower half of the address space as a safe region at load time; this is straightforward for position-independent executables as their segments are mapped exclusively in the upper half of the address space by default. To compute safe addresses, we subtract 64 TB[1] from the addresses used by *vtblptr* writes or virtual calls. To thwart value probing attacks in the safe region, we (i) mark all safe region pages as inaccessible by default and make them accessible on demand, and (ii) use a fixed offset chosen randomly at load time for writes to the safe region. To achieve the latter, we write a random value to the `gs` register and use it as the offset for all accesses to the safe region. To mark pages as readable/writable on demand, we use a custom segfault handler that uses `mprotect` to allow accesses from our library. This means that when a *vtblptr* is written into the safe memory region and the page is not yet accessible, our segfault handler checks if the write access is done by our library and makes the page accessible if it is. Otherwise, a probing attack is detected and execution is stopped. The page remains accessible which speeds up further *vtblptr* writes to it.

We omit an evaluation of potential optimizations already explored in prior work [26, 93], such as avoiding Dyninst's penalties for (re)storing unclobbered live registers or removing trampoline code left over after nopping out analysis instrumentation code. Similarly, we do not implement hash-based safe region compression that would reduce virtual and physical memory usage and allow increased entropy in the safe region, nor do we use Intel MPK [40] to further secure the safe region. Since we focus on the exact analysis of binary applications and the subsequent instrumentation, we consider these optimizations orthogonal to our work.

5.5 Evaluation

In this section, we evaluate VPS in terms of performance and accuracy. We focus our evaluation on MySQL, Node.js, MongoDB, and the fifteen C++ benchmarks found in SPEC CPU2006 and CPU2017 [141, 142]. Even though our approach is able to handle proprietary software, we evaluate it on open-source software since otherwise, we are not able to generate a ground truth to compare against.

[1] Linux x86-64 provides 47 bits for user space mappings, and $2^{47} = 128$ TB.

5.5.1 Virtual Callsite Identification Accuracy

In order to measure the accuracy of the protection of VPS, we evaluate the accuracy of the vcall identification analysis. The results show that VPS, although a binary-only approach, can almost reach the same degree of protection as a source-based approach. Compared to the state-of-the-art binary-only approach *Marx*, it identifies more vcalls with fewer false-positives. As applications for our evaluation, we use the C++ programs of SPEC CPU2006 and SPEC CPU2017 that contain virtual callsites, as well as the MySQL server binary (5.7.21), the Node.js binary (8.10.0), and the MongoDB binary (3.2.4). We used the default optimization levels (`O2` for CPU 2006, `O3` for all others). The analysis was performed on Ubuntu 16.04 LTS running on an Intel Core i7-2600 CPU with 32 GB of RAM.

VTV To gain a ground truth of virtual callsites, we use *VTV* [149] and compare against our analysis results. Since *VTV* leverages source code information, its results are usually used as ground truth for binary-only approaches focusing on C++ virtual callsites. All programs except MongoDB are compiled with GCC 8.1.0. MongoDB crashed during compilation and had to be compiled with the older version GCC 4.9.3. Unfortunately, compiling *450.soplex* results in a crash and it is therefore omitted. Table 5.1 shows the results of our vcall accuracy evaluation.

Overall, we observe that the analysis of VPS is capable of identifying the vast majority of virtual callsites in the binary. This ranges from 91.7% (*510.parest_r*) to all vcalls detected (several benchmarks). Our average recall is 97.8% on SPEC CPU2006 and 97.4% on SPEC CPU2017. With the exception of one outlier (*526.blender_r* with precision 68.3%) we have a low number of false positives, with precision ranging from 87.0% (*447.dealII*) to no false positives at all (several benchmarks). The results are similar for large real-world applications with a recall ranging from 91.8% (*MongoDB*) to 97.6% (*MySQL*) and a precision ranging from 97.2% (*Node.js*) to 99.7% (*MongoDB*). The high recall rate shows that our binary-only approach is able to protect almost as many virtual callsites as *VTV* does and hence provides comparable security as this source-based approach. However, it still misses some vcalls which may leave an attacker with a small room to perform an attack under the right circumstances. The precision rates show that although we have a low false-positive identification rate, we still have some.

To cope with the problem of false-positive identifications, we verify vcalls before we actually instrument them with our security check. The static analysis verification is able to verify 37.9% in the best case (*526.blender_r*) and in the worst case none. On average we verified 20.4% on SPEC CPU2006 and 18.3% on SPEC CPU2017. For large applications, the best verification rate is 12.2% (*Node.js*) and the worst 3.1% (*MongoDB*). Dynamic verification (see Section 5.3.3.3) considerably improves verification performance, verifying 35.1% and 25.9% for SPEC CPU2006 and 2017. Unfortunately, we were not able to execute *510.parest_r*, *MySQL* and *MongoDB* with *VTV*. The applications crashed with an error message stating that *VTV* was unable to verify a vtable pointer (i.e., a false positive). Hence, the only large real-world application with dynamic verification *Node.js* verified 20.2% of the vcalls.

Table 5.1: Results of our vcall accuracy evaluation. For each application this table shows (i) the code size, time needed for the static analysis (hh:mm:ss) and the ground truth generated by *VTV*; (ii) static vcall identification, depicting the number of indirect call instructions identified as vcall that are true positives and false positives as well as recall and precision; (iii) static vcall verification results, listing the number of verified vcall instructions, verified vcalls in percentage and verified false positives; (iv) static and dynamic verification results, showing the number of verified vcall instructions, verified vcalls in percentage, verified false positives, and the number of identified false positives removed. Cases where dynamic verification failed due to *VTV* false positives are in parentheses.

				Static Identification			
Program	*Code Size*	*Time*	*# GT*	*# TP*	*# FP*	*Recall (%)*	*Precision (%)*
447.dealII	4.18 MB	0:02:15	1,558	1,450	215	93.0	87.1
450.soplex	–	–	–	–	–	–	–
453.povray	1.09 MB	0:00:04	102	102	10	100.0	91.1
471.omnetpp	1.17 MB	0:04:00	802	800	0	99.8	100.0
473.astar	0.04 MB	0:00:00	1	1	0	100.0	100.0
483.xalancbmk	7.17 MB	5:54:25	13,440	12,915	17	96.1	99.9
Average [SPEC CPU2006]						*97.8*	*95.6*
510.parest_r	12.69 MB	1:00:00	4,678	4,288	528	91.7	89.0
511.povray_r	1.20 MB	0:00:05	122	122	14	100.0	89.7
520.omnetpp_r	3.60 MB	0:06:57	6,430	6,190	23	96.3	99.6
523.xalancbmk_r	10.34 MB	15:20:40	33,880	33,069	12	97.6	100.0
526.blender_r	11.47 MB	0:03:29	174	172	80	98.9	68.3
541.leela_r	0.33 MB	0:00:01	1	1	0	100.0	100.0
Average [SPEC CPU2017]						*97.4*	*91.1*
MongoDB	48.22 MB	1:57:39	17,836	16,366	44	91.8	99.7
MySQL	35.95 MB	65:57:27	11,876	11,592	179	97.6	98.5
Node.js	38.13 MB	5:16:09	12,643	12,330	353	97.5	97.2

	Static Verification			**Static and Dynamic Verification**			
Program	*#*	*%*	*# FP*	*#*	*%*	*# FP*	*# removed*
447.dealII	379	24.3	7	423	27.2	18	0
450.soplex	–	–	–	–	–	–	–
453.povray	32	31.4	0	55	53.9	0	6
471.omnetpp	245	30.6	0	530	66.1	0	0
473.astar	0	0.0	0	0	0.0	0	0
483.xalancbmk	2,122	15.8	0	3,792	28.2	1	0
Average [SPEC CPU2006]		*20.4*			*35.1*		
510.parest_r	660	14.1	13	(660)	(14.1)	(13)	–
511.povray_r	33	27.1	0	62	50.8	0	6
520.omnetpp_r	1,585	24.7	0	2,286	35.6	6	0
523.xalancbmk_r	1,948	5.8	0	4,961	14.6	0	0
526.blender_r	66	37.9	0	70	40.2	0	49
541.leela_r	0	0.0	0	0	0.0	0	0
Average [SPEC CPU2017]		*18.3*			*25.9*		
MongoDB	552	3.1	0	(552)	(3.1)	(0)	–
MySQL	1,330	11.2	3	(1,330)	(11.2)	(3)	–
Node.js	1,538	12.2	10	2,559	20.2	45	118

```
92      /**
93      * Destroy the object pointed to by a pointer type.
94      */
95      template<typename _Tp>
96      inline void
97      _Destroy(_Tp* __pointer)
98      { __pointer->~_Tp(); }
```

(a) Snippet from `stl_construct.h`.

```
2545    Vector<double> us[dim];
2546    for (unsigned int i=0; i<dim; ++i)
2547        us[i].reinit (dof_handler.n_dofs());
```

(b) Snippet from `grid_generator.cc`.

Figure 5.3: Two source code snippets where *VTV* fails to identify a virtual callsite.

A manual analysis of the missed virtual callsites (false negatives) reveals two possibilities for a miss: the data flow was too complex to be handled correctly by our implementation, or the described pattern in Section 5.3.3.1 was not used. The former can be fixed by improving the implemented algorithm that is used for finding the described pattern. In the latter, the *vtblptr* is extracted from the object, however, a newly-created stack object is used as *thisptr* for the virtual callsite which does not follow a typical C++ callsite pattern. This could be addressed by considering additional vcall patterns, at the risk of adding false positives. Given our already high recall rates, we believe this would not be a favorable trade-off.

We also verified 86 cases which *VTV* did not recognize as virtual callsite instructions. A manual verification of all cases show that these are indeed vcall instructions and hence missed virtual callsites by *VTV*. For example, Figure 5.3a depicts the relevant code for 34 of these cases that are linked to the compiler provided file `stl_construct.h`. Line 98 provides the missed vcall instruction that calls the destructor of the provided object. Since the destructor of a class is also a virtual function, it is invoked with the help of a virtual callsite. Another example is given in Figure 5.3b for *510.parest_r*. Here, a vector is created and the function `reinit()` is invoked on line 2547. However, since the class `dealii::Vector<double>` is provided by the application and `reinit()` is a virtual function of this class, this function call is translated into a virtual callsite. We contacted the *VTV* authors about this issue and they confirmed that this happens because the compiler accesses the memory of the objects directly when calling the virtual function in the internal intermediate representation. Usually, the compiler accesses them while going through an internal *vtblptr* field. Unfortunately, to fix this issue in *VTV* would require a lot of non-trivial work since the analysis has to be enhanced.

CFIXX Since *CFIXX* performs the enforcement in a similar way, we also evaluated our binary-only approach against this source code based method. Hence, we compiled the applications with *CFIXX* which is based on LLVM and extracted the protected virtual

Table 5.2: Results of our comparison against *CFIXX*. For each application this table shows (i) the ground truth generated by *CFIXX*; (ii) static vcall identification, depicting the number of indirect call instructions identified as vcall that are true positives and false positives as well as recall and precision.

		Static Identification			
Program	*# GT*	*# TP*	*# FP*	*Recall (%)*	*Precision (%)*
447.dealII	–	–	–	–	–
450.soplex	553	553	10	100.0	98.2
453.povray	110	110	11	100.0	90.9
471.omnetpp	943	942	0	99.9	100.0
473.astar	1	1	0	100.0	100.0
483.xalancbmk	12,670	12,427	527	98.0	95.9
Average [SPEC CPU2006]				99.6	97.0
510.parest_r	7,288	7,194	265	98.7	96.5
511.povray_r	119	119	11	100.0	91.5
520.omnetpp_r	6,037	6,032	71	99.9	98.8
523.xalancbmk_r	23,661	26,407	528	98.9	97.8
526.blender_r	–	–	–	–	–
541.leela_r	2	2	0	100.0	100.0
Average [SPEC CPU2017]				99.5	96.9
MongoDB	20,873	20,716	448	99.3	97.9
MySQL	13,035	12,921	380	99.1	97.1
Node.js	13,013	12,982	491	99.8	96.4

Table 5.3: Results of *Marx*'s vcall accuracy evaluation. For each application this table shows (i) the ground truth generated by *VTV*; (ii) static vcall identification, depicting the number of indirect call instructions identified as vcall that are true positives and false positives as well as recall and precision.

		Static Identification			
Program	*# GT*	*# TP*	*# FP*	*Recall (%)*	*Precision (%)*
447.dealII	1,558	1,307	122	83.9	91.5
450.soplex	–	–	–	–	–
453.povray	102	98	10	96.1	90.7
471.omnetpp	802	701	3	87.4	99.6
473.astar	1	1	0	100.0	100.0
483.xalancbmk	–	–	–	–	–
Average [SPEC CPU2006]				91.8	95.4
510.parest_r	4,678	3,673	295	78.5	92.6
511.povray_r	122	115	11	94.3	91.3
520.omnetpp_r	6,430	5,465	22	85.0	99.6
523.xalancbmk_r	33,880	23,541	33	69.4	99.9
526.blender_r	174	171	1,347	98.3	11.3
541.leela_r	1	0	0	0.0	0.0
Average [SPEC CPU2017]				70.9	65.8
MongoDB	17,836	12,437	1,249	69.7	90.9
MySQL	11,876	10,867	1,214	81.3	88.8
Node.js	12,643	10,648	1,095	84.2	90.7

callsites as ground truth for our comparison. Table 5.2 shows the results of this evaluation. Unfortunately, we were not able to compile *447.dealII* and *526.blender_r* with *CFIXX*. As the table shows, VPS can identify on average 99.6% of all SPEC CPU2006 and 99.5% of SPEC CPU2017 virtual callsites that are also protected by *CFIXX*. Furthermore, VPS also yields a high precision with 97.0% for SPEC CPU2006 and 96.9% for SPEC CPU2017 on average. For large real-world applications, the recall and precision rates are similar with a recall of 99.1% for *MySQL* and 99.8% for *Node.js* and a precision of 97.1% and 96.4% respectively. A manual analysis of the missed virtual callsites (false negatives) showed the same two reasons for a miss that also occurred for *VTV*.

Marx A direct comparison of the accuracy with other binary-only approaches is difficult since different test sets are used to evaluate it. For example, vfGuard evaluates the accuracy of their approach against only two applications, while *T-VIP* is only evaluated against one. *VTint* states absolute numbers without any comparison with a ground truth. *VCI* evaluates their approach against SPEC CPU2006, but the numbers given for the ground truth created with *VTV* differ completely from ours (e.g., 9,201 vs. 13,440 vcalls for *483.xalancbmk*) which makes a comparison difficult. Additionally, the paper reports no false positives during their analysis which we encounter in the same application set with a similar identification technique. Unfortunately, as discussed in Section 5.3.3.1, we were not able to determine the reason for this. Furthermore, most approaches target different

Table 5.4: Object creation and destruction accuracy results, showing the number of vtable references in the code as found in the ground truth and as identified or missed by our analysis.

Program	*# GT*	*# identified*	*# missed*
447.dealII	–	–	–
450.soplex	102	228	0
453.povray	103	226	0
471.omnetpp	372	871	0
473.astar	0	8	0
483.xalancbmk	2,918	6,530	0
510.parest_r	12,482	25,804	0
511.povray_r	103	224	0
520.omnetpp_r	1,381	3,280	0
523.xalancbmk_r	2,790	6,323	0
526.blender_r	–	–	–
541.leela_r	87	180	0
MongoDB	8,054	11,401	0
MySQL	8,532	11,524	0
Node.js	7,816	19,204	0

platforms than VPS (Windows x86 and Linux x86) and are not open source. Since *Marx* is the only open-source approach that targets the same platform, we analyzed our evaluation set with it. In order to create as few false positives as possible, we used its conservative mode. Unfortunately, *Marx* crashed during the analysis of *483.xalancbmk*. The results of the analysis can be seen in Table 5.3. Compared to *Marx*, VPS' analysis has considerably higher recall with better precision. Averaged over the CPU2006 benchmarks supported by *Marx*, VPS achieves 98.2% recall (91.8% for *Marx*) and on CPU2017 97.4% versus 70.9%, respectively. This does not come at the cost of more false positives, as our precision is similar on CPU2006 (94.5% vs. 95.4%) and much better on CPU2017 (91.1% vs. 65.8%). For large real-world applications like *MySQL* and *MongoDB*, VPS identifies 16.3% and 28.1% more virtual callsites with better precision (98.5% vs. 88.8% for *MySQL* and 99.7% vs. 90.9% for *MongoDB*).

Overall, our analysis shows that VPS is precise enough to provide an application with protection against control-flow hijacking attacks at virtual callsites. The evaluation showed that on average only 2.5% when comparing against *VTV* and 0.5% comparing against *CFIXX* of the vcalls were missed. Since binary analysis is a hard problem, the results are very promising in showing that a sophisticated analysis can almost reach the same degree of protection as a source-based approach. In addition, it shows that even source-code approaches such as *VTV* do not find all virtual callsite instructions and can benefit from binary-only approaches such as VPS. Furthermore, the number of false positives show the sensibility of our approach to handle them during instrumentation rather than assume their absence.

5.5.2 Object Initialization/Destruction Accuracy

To avoid breaking applications, VPS must instrument all valid object initialization and destruction sites. To ensure that this is the case, we compare the number of vtable-referencing instructions found by VPS to a ground truth. We generate the ground truth with an LLVM 4.0.0 pass that instruments Clang's internal function `CodeGenFunction::InitializeVTablePointer()`, which Clang uses for all vtable pointer initialization.

Table 5.4 shows the results for the same set of applications we used in Section 5.5.1. We omit results for *447.dealII* from SPEC CPU 2006 and *526.blender_r* from SPEC CPU 2017 because these benchmarks fail to compile with LLVM 4.0.0. The results for the remaining applications show that our analysis finds all vtable-referencing instructions. It conservatively overestimates the set of vtable-referencing instructions, ensuring the security and correctness of VPS at the cost of a slight performance degradation due to the overestimated instruction set.

5.5.3 Performance

This section evaluates the runtime performance of VPS by measuring the time it takes to run each C++ benchmark in SPEC CPU2006 and CPU2017. We compare VPS-protected runtimes against the baseline of original benchmarks without any instrumentation. We compile all test cases as position-indepedent executables with GCC 6.3.0. For each benchmark, we report the median runtime over 11 runs on a Xeon E5-2630 with 64 GB RAM, running CentOS Linux 7.4 64-bit. We use a single additional run with more logging enabled to obtain statistics such as the number of executed virtual calls. Table 5.5 details our results.

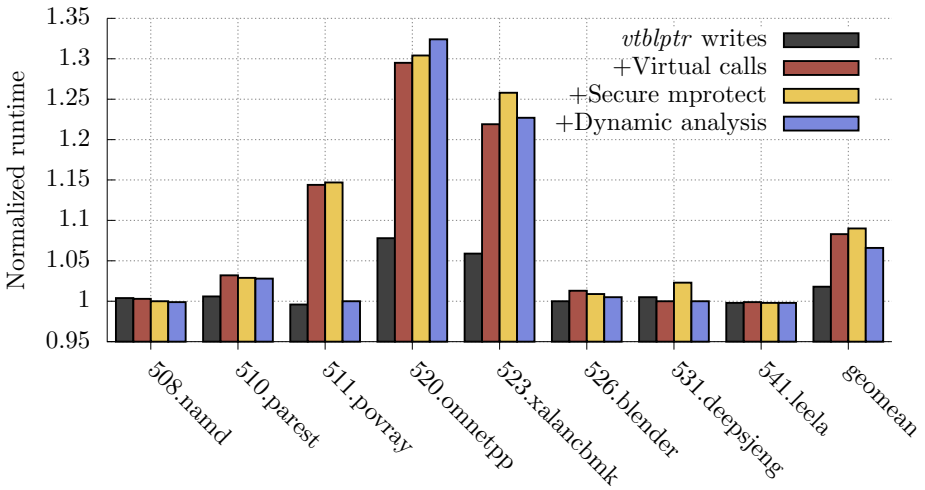
Our results show the variety in properties of C++ applications. Some programs make little to no use of virtual dispatching, e.g., *444.namd*, *508.namd_r*, *531.deepsjeng_r*, and *473.astar*. Others contain thousands of *vtblptr* writes and virtual callsites, e.g., *510.parest_r* with over 12,000 *vtblptr* writes, or *483.xalancbmk* in CPU2006 with more than 1,300 verified virtual callsites. Further details are shown in the first group in Table 5.5.

The comparison of verified virtual calls (true positive) and regular indirect calls (false positive) shows the accuracy of our analysis. Almost all vcall candidates turn out to be real vcalls. Furthermore, with absolute numbers of executed virtual calls and *vtblptr* writes in the billions, it is clear that our instrumentation must be lightweight. The second group in Table 5.5 depicts the exact numbers.

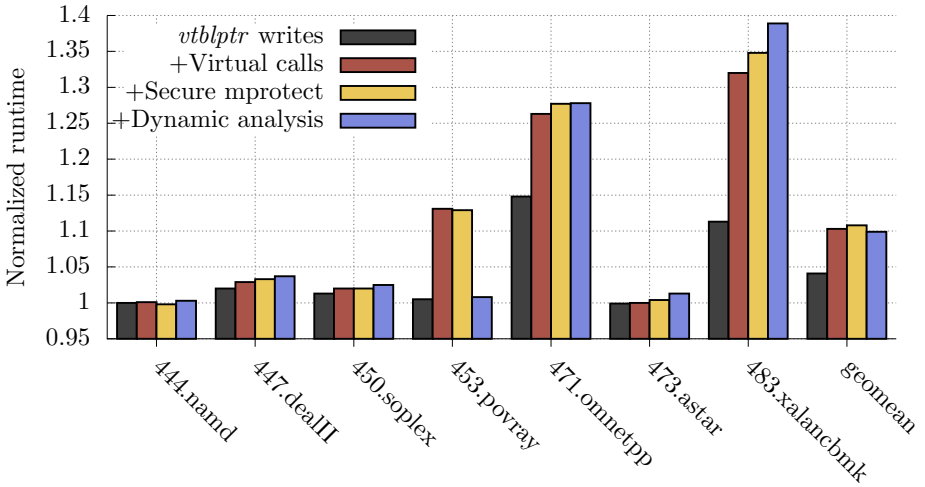
The runtime overhead of our instrumentation varies from 0% for programs with little to no virtual dispatch code to 35% for the worst-case scenario (*483.xalancbmk*). In almost all cases, we see a correlation between increased overhead and number of instrumentation points (*vtblptr* writes and virtual calls). An exception is *511.povray_r*, which shows a 15% performance decrease despite a relatively low number of vcalls and *vtblptr* writes. Further inspection shows that this is caused by the 6 false positives candidate vcalls; if we disable hot-patching, our vcall instrumentation code is called over 18 billion times. While we remove instrumentation hooks for the majority of these cases, which are not real vcalls,

Table 5.5: VPS performance results and runtime statistics. For each binary, this table shows (i) **binary instrumentation** details, depicting the number of instrumented *vtblptr* writes (*#vtblptr*), positive virtual calls (*#positive*), and candidate vcalls (*#candidates*); (ii) **runtime statistics**, listing the number of true positive (*#TP*) and false positive (*#FP*) virtual calls, and the total number of virtual calls (*#vcalls*) and *vtblptr* writes (*#vtblptr*); and (iii) **runtime overhead**, listing runtime overhead (VPS) compared to the baseline (*base*) in seconds.

	Binary instrumentation			**Runtime statistics**				**Runtime overhead**	
Program	*# vtblptr*	*# positive*	*# candidates*	*# TP*	*# FP*	*# vcalls*	*# vtblptr*	*base*	*VPS*
444.namd	6	0	2	0	0	0	2,018	343.5	342.9 (+ 0%)
447.dealII	4,283	161	1,459	47	0	97m	21m	289.7	299.2 (+ 3%)
450.soplex	120	195	364	48	0	1,665,968	40	215.8	220.2 (+ 2%)
453.povray	98	21	91	21	6	101,743	162	135.8	153.3 (+13%)
471.omnetpp	507	117	677	327	0	1,585m	2,156m	290.0	370.2 (+28%)
473.astar	0	0	1	0	0	0	0	350.3	351.6 (+ 0%)
483.xalancbmk	4,554	1,348	11,623	1,639	0	3,822m	2,316m	185.0	249.4 (+35%)
Geometric mean [SPEC CPU2006]									+ 11%
508.namd_r	48	0	0	0	0	0	21	271.8	271.8 (+ 0%)
510.parest_r	12,206	243	4,539	350	4	2,625m	119m	586.3	603.1 (+ 3%)
511.povray_r	113	19	121	21	6	4,577	183	498.7	572.0 (+15%)
520.omnetpp_r	2,591	447	5,310	751	0	7,958m	2,070m	507.4	661.7 (+30%)
523.xalancbmk_r	4,512	801	30,771	2,844	0	4,873m	2,314m	366.8	461.5 (+26%)
526.blender_r	43	37	174	4	46	11	3	325.8	328.6 (+ 1%)
531.deepsjeng_r	0	0	0	0	0	0	0	345.1	353.1 (+ 2%)
541.leela_r	177	0	2	0	0	0	404,208	535.5	534.6 (+ 0%)
Geometric mean [SPEC CPU2017]									+ 9%



(a) Microbenchmarks for SPEC CPU2017



(b) Microbenchmarks for SPEC CPU2006

Figure 5.4: Normalized runtime for C++ programs in SPEC CPU2006 and CPU2017, with cumulative configurations: (i) only instrument *vtblptr writes*; (ii) also instrument *virtual call instructions*; (iii) *secure the safe region* by marking all pages unwritable, and only selectively `mprotect`-ing them if they are accessed from our own instrumentation code; and (iv) include offline *dynamic analysis* results, reducing the need for hot-patching.

our current implementation does not remove the Dyninst trampolines. These trampolines are the source of the unexpected overhead. The numbers depicting the comparison of the uninstrumented baseline runs to VPS-protected runs are shown in the third group in Table 5.5.

To better understand the overhead of VPS, we gathered detailed statistics for both SPEC CPU2006 and SPEC CPU2017 in varying configurations. We first run SPEC with only instrumentation for *vtblptr* writes enabled. In this run, the entire safe region is read/writable and the instrumentation only (i) computes the address in the safe region to store the vtable pointer at, and (ii) copies the vtable pointer there. In the second configuration, we additionally instrument virtual calls. We check whether candidates are actual vcalls by testing the call's first argument and, if it can be dereferenced, looking this value up in the list of known vtables. We then either patch verified vcalls to enable the fast path, or remove instrumentation for false positives. The fast path fetches the vtable pointer by dereferencing the first argument, and then compares it against the value stored in the safe region. The third configuration additionally makes the safe region read-only and uses a segfault handler to mark pages writable on demand. Finally, the fourth configuration includes dynamic analysis results, removing the need to hot-patch previously verified vcalls at runtime. The results show that the majority of VPS's overhead stems from (i) *vtblptr* writes, and (ii) virtual callsite instrumentation. Figure 5.4 details the numbers of this evaluation.

Overall, with a geometric mean performance overhead of 11% for SPEC CPU2006 and 9% for SPEC CPU2017, VPS shows a moderate performance impact. As expected, it does not perform as well as a source-based approach such as *VTV* with reported 4% geometric mean for SPEC CPU2006 [149]. However, it outperforms comparable previous work (*VCI* with 14% [55] and *T-VIP* with 25% [63]) and performs slightly worse than *Marx*'s *VTable Protection* with a reported 8% geometric mean for SPEC CPU2006, however, with better accuracy and additional type integrity.

5.6 Discussion

This section first discusses the susceptibility of VPS to COOP attacks [135]. Next, we discuss the limitations of VPS.

5.6.1 Counterfeit Object-oriented Programming

CFI approaches targeting C++ must cope with advanced attackers using Counterfeit Object-oriented Programming (COOP) attacks [44, 135]. This attack class thwarts defenses that do not accurately model C++ semantics. As we argue below, VPS reduces the attack surface sufficiently that practical COOP attacks are infeasible.

For a successful COOP attack, an attacker must control a container filled with objects, with a loop invoking a virtual function on each object. The loop may be an actual loop, called a *main loop gadget*, or can be achieved through recursion, called a *recursion gadget*. We refer to both types as *loop gadget*. The attacker places counterfeit objects in the container, allowing them to hijack control flow when the loop executes each object's virtual function. To pass data between objects, the attacker can overlap the objects' fields.

The first restriction VPS imposes on an attacker is to prevent filling the container with counterfeit objects; because the objects were not created at legitimate object creation sites, the safe memory does not contain stored *vtblptr*s for them. An attacker has only two options to craft a container of counterfeit objects under VPS: either the program allows attackers to arbitrarily invoke constructors and create objects, or the attacker can coax the program into creating all objects needed on their behalf. The former occurs (in restricted form) only in programs with scripting capabilities. The latter scenario, besides requiring a cooperative victim program, hinges on the attacker's ability to scan data memory to find all needed objects without crashing the program (hence losing the created objects) and filling the container with pointers to these.

The second restriction is prohibiting overlapping objects (used for data transfer in COOP), since objects can only be created by legitimate constructors. As a result, a COOP attack would have to pass data via argument registers or scratch memory instead. Data passing via argument registers works only if the loop gadget does not modify the argument registers between invocations. Moreover, the virtual functions called must leave their results in the correct argument registers when they return. Passing data via scratch memory limits the attack to the use of virtual functions that work on memory areas. The pointer to the scratch memory area must then be passed to the virtual function gadgets either via an argument register (subject to the earlier limitations), or via a field in the object. To use a field in the object as a pointer to scratch memory, the attacker must overwrite that field prior to the attack, which could lead to a crash if the application tries to use the modified object.

As a third restriction, VPS's checks of the *vtblptr* at each vcall instruction limit the virtual functions attackers can use at a *loop gadget*. Only the virtual function at the specific vtable offset used by the vcall is allowed; attackers cannot "shift" vtables to invoke alternative entries. This security policy is comparable to *vfGuard* [127].

To summarize, VPS restricts three crucial COOP components: object creation, data transfer, and *loop gadget* selection. Because all proof of concept exploits by Schuster et al. [135] rely on object overlapping as a means of transferring data, VPS successfully prevents them. Moreover, Schuster et al. recognize *vfGuard* as a significant constraint for an attacker performing a COOP attack. Given that VPS raises the bar even more than *vfGuard*, we argue that VPS makes currently existing COOP attacks infeasible.

We found that multiple of the virtual callsites missed by *VTV* (as shown in Section 5.5.1) reside in a loop in a destructor function (similar to the *main loop gadget* example used by Schuster et al. [135]). Because the loop iterates over a container of objects and uses a virtual call on each object, COOP attacks can leverage these missed callsites as a *main loop gadget* even with *VTV* enabled. This demonstrates the need for defense-in-depth, with multiple hurdles for an attacker to cross in case of inaccuracies in the analysis.

5.6.2 Limitations

At the moment, our proof of concept implementation of the instrumentation ignores object deletion because it does not affect the consistency of the safe memory. As a result, when an object is deleted, its old *vtblptr* is still stored in safe memory. If an attacker manages to control the memory of the deleted object, they can craft a new object that uses the

same vtable as the original object. Because the *vtblptr* remains unchanged, this attack is analogous to corrupting an object's fields and does not allow the attacker to hijack control. Thus, while our approach does not completely prevent use-after-free, it forces an attacker to re-use the type of the object previously stored in the attacked memory.

Another limitation of our approach lies in the runtime verification of candidate vcall sites. If an attacker uses an unverified vcall instruction, they can force the analysis instrumentation to detect a "false positive" vcall and remove the security instrumentation for this instruction, leaving the vcall unprotected. Because we cache analysis results, this attack only works for vcall sites that are unverified in the static analysis and have never been executed before in any run of the program (since otherwise only the security check is performed), leading to a race condition between the analysis instrumentation and the attacker. The only way to mitigate this issue is by improving coverage during the dynamic profiling analysis and therefore reducing the number of unverified vcalls. This is possible by running test cases for the protected program or through techniques such as fuzzing [77, 130]. Note also that this attack requires specific knowledge of an unverified vcall; if the attacker guesses wrong and attacks a known vcall, we detect and log the attack.

VPS inherits some limitations from Dyninst, such as Dyninst's inability to instrument functions that catch or throw C++ exceptions and Dyninst's inability to instrument functions for which it fails to reconstruct a CFG. These limitations are not fundamental to VPS and can be resolved with additional engineering effort.

Finally, we note that our safe memory region implementation can be enhanced to provide stronger protection against probing attacks [68, 116]. For example, this can be done by using hardware features such as Memory Protection Keys (MPK) [40]. In the current implementation, an adversary might still be able to overwrite values in the safe memory region under the right circumstances. However, because the safe region is merely a building block for VPS, we consider improvements to safe memory an orthogonal topic (e.g., [89]) and do not explore it further in this work.

5.7 Related Work

A lot of work has been presented on C++ defenses in the past. In the following, we discuss the ones closely related to ours. Although VPS is a binary-only defense, we also describe source-based related work.

5.7.1 Binary-Only Defenses

Marx presented in Chapter 4 reconstructs class hierarchies from binaries for *VTable Protection* and *Type-safe Object Reuse*. VTable Protection verifies at each vcall whether the *vtblptr* resides in the reconstructed class hierarchy. However, the analysis is incomplete and the instrumentation falls back to *PathArmor* [156] for missing results. *Marx*'s Type-safe Object Reuse prevents memory reuse between different class hierarchies, reducing the damage that can be done with use-after-free. However, this approach leaves considerable wiggle room for attackers for large class hierarchies. In contrast, VPS does not rely on

Table 5.6: C++ binary-only mitigation mechanisms

Defense	Binary-only	Protects vcalls	Protects type	Protects dangl. ptrs	Tolerates FP vcalls
Marx (VTable)	✓	✓	✗	✓	✓
Marx (Type-safe)	✓	✗	✗	✓	n.a.
vfGuard [127]	✓	✓	✗	✓	✗
T-VIP [63]	✓	✓	✗	✓	✗
VTint [165]	✓	✓	✗	✓	✗
VCI [55]	✓	✓	✗	✓	✗
VTPin [133]	needs RTTI	✗	✗	✓	n.a.
VPS	✓	✓	✓	✓	✓

Defense	Security Strategy
Marx (VTable)	*vtblptr* in reconstructed class hierarchy (fallback *PathArmor* [156]).
Marx (Type-safe)	Memory allocator uses class hierarchy as type.
vfGuard [127]	Call target resides in at least one vtable at correct offset.
T-VIP [63]	*vtblptr* and random vtable entry must point to read-only memory.
VTint [165]	Verifies vtable ID, vtable must be in read-only memory.
VCI [55]	*vtblptr* must be statically found, in class hierarchy, or *vfGuard*-allowed.
VTPin [133]	Overwrites *vtblptr* when object freed.
VPS	Check at vcall if object was created at a legitimate object creation site.

class hierarchy information and provides stronger security by only allowing exactly correct types. Moreover, *Marx* only protects the heap whereas VPS protects all objects.

VTint [165] instruments vtables with IDs to check their validity, but unlike VPS allows exchanging the original *vtblptr* with a new pointer to an existing vtable. Moreover, *VTint* breaks the binary in case of false positives.

VTPin [133] overwrites the *vtblptr* whenever an object is freed to protect against use-after-free, but it requires RTTI and does not prevent *vtblptr* overwrites in general.

vfGuard [127] identifies vtables and builds a mapping of valid target functions at each vtable offset. At vcalls, it checks the target and calling convention. Unlike VPS, *vfGuard* allows fake vtables as long as each entry appears in a valid vtable at the same offset. Further, *vfGuard* may break the binary in case of false positives.

T-VIP [63] protects vcalls against fake vtables, but breaks the binary when vtables reside in writable memory (e.g., in `.bss`). Moreover, unlike VPS, *T-VIP* uses potentially bypassable heuristics.

VCI [55] only allows a specific set of vtables at each vcall, mimicking *VTV* [149]. When the analysis cannot rebuild the sets precisely, *VCI* falls back to *vfGuard*. Moreover, false positive virtual callsites in *VCI* break the application, as may incomplete class hierarchies (e.g., due to abstract classes as described in Chapter 4). In contrast, VPS allows calls through any legitimately created object. Moreover, even in the hypothetical case of a perfect *VCI* analysis, *VCI* allows changing the *vtblptr* to another one in the set, unlike VPS.

Table 5.6 shows a summary of the comparison of our design against other binary-only C++ defenses. Overall, existing defenses targeting vtable-hijacking attacks assign a set of allowed target functions to each virtual callsite (*Marx*'s *VTable Protection* presented in Section 4.4.1, *vfGuard* [127], *T-VIP* [63], *VTint* [165] and *VCI* [55]). The inaccuracy of

binary analysis forces them to overestimate the target set, leaving room for attacks [135]. In contrast, VPS enforces that vtable pointers remain unmodified after object construction, ensuring that only validly created objects can be used at virtual callsites and reducing the attack surface even compared to a hypothetical defense with a perfect set of allowed targets.

5.7.2 Source-Based Defenses

VTV [149] is a GCC compiler pass that only allows a statically determined set of vtables at each vcall, like most binary-only approaches [55, 63, 127] (as well as *Marx*'s *VTable Protection* presented in Section 4.4.1).

CFIXX [26] is the state-of-the-art source-based C++ defense. Like VPS, it stores *vtblptr*s in safe memory and fetches them at each callsite. Given the lack of comparison against the *vtblptr* as stored in the object, *CFIXX* prevents but does not detect vtable hijacking. As an LLVM compiler extension, *CFIXX* cannot protect applications for which no source code (and LLVM compilation) is available. Therefore, proprietary legacy applications cannot be protected afterwards. While *CFIXX* and VPS offer similar security, our binary-level analysis is completely novel. Unlike source-level analysis, our analysis must consider both direct and indirect vtable accesses. Moreover, identifying the virtual callsites for subsequent security instrumentation is challenging given the lack of type information.

VTrust [164] is a source code CFI implementation focusing on C++ applications which is divided into two components. The first component of this approach creates hashes of function types during the compilation process and emits them in front of each function At each virtual callsite, the emitted hash is checked against the type of the vcall. If a mismatch is found, an attack is identified. The second component replaces the *vtblptr* of an object with an index. This index is then used to lookup the corresponding vtable in a global table. In contrast to VPS and *CFIXX*, this table is only used as a lookup table for vtable addresses. Hence, it is basically just a way to encode the *vtblptr*. Therefore, an attacker is still able to replace the index of an object (which is just an encoded *vtblptr*) with another valid index (which is an encoded *vtblptr* to another vtable). However, the first component of the approach limits the set of valid vtables.

The approach by Bounov et al. [21] is a compiler pass that reorders the memory layout of vtable hierarchies and interleaves them. As a result, at each virtual callsite a simple range and alignment check is sufficient to verify if a vtable from the correct hierarchy is used. However, in order to work correctly the application has to be linked statically.

5.8 Conclusion and Future Work

In this chapter, we presented VPS, a practical binary-level defense mechanism against C++ vtable hijacking attacks. Unlike prior work that restricts the target set of virtual callsites, our approach protects objects *at creation time* and restricts their usage to virtual calls that are reachable by the object. This sidesteps accuracy problems faced by prior work while simultaneously extending the threat model to include use-after-free attacks. Moreover, VPS provides improved correctness guarantees by handling false positives at vcall verification

time. Our evaluation shows that VPS precisely protects applications from modern C++ code-reuse attacks, including whole-function reuse.

Our analysis for detecting virtual callsites in binaries uncovered inaccuracies in *VTV*, a source-based approach that is commonly used as ground truth in this research area and broadly considered among the state-of-the-art for C++-based defenses. We reported these issues to the *VTV* maintainers.

A topic for future work is to improve the binary instrumentation of VPS. At the moment, the safe memory region of our proof of concept implementation is vulnerable to sophisticated probing attacks. Enhancing it with hardware features such as MPK or other techniques can significantly improve the security of our VPS implementation. Furthermore, the evaluation showed an overestimation of object creation and destruction sites. Although, as a result of this overestimation, no vtable write was missed, it led to a performance overhead. Improving this analysis could make VPS faster.

Chapter 6

Towards Automated Application-Specific Software Stacks

Software is getting larger and more complex [11, 30, 95, 97, 107]. One way to handle this complexity during the development process is the choice of the programming language (e.g., C++ to use object-oriented programming). Another way to handle complexity during development is to reuse existing code in the form of shared libraries. This gives developers the possibility to focus solely on the user-facing application rather than re-implementing common functionality such as memory management or string processing functions over and over again. However, since not all code of a given shared library is used in a given program, the downside of this concept is that unnecessary code is loaded into memory: a recent study finds that only 5% of the *libc*, the standard library for the C programming language, is used on average across 2,016 applications of the Ubuntu Desktop environment [129].

From an attacker's perspective, the typical way to exploit an existing vulnerability is to reuse existing code (e.g., ret2libc [152] or return-oriented programming [136] (ROP)) to execute shellcode and bypass existing mitigation systems such as $W \oplus X$ and address space layout randomization (ASLR). Since shared libraries offer a plethora of (mostly) unused code, the attacker has a large variety of existing functions or code parts to choose from.

The same holds for applications written in interpreted languages, such as PHP, Python, or Ruby: the interpreter is a complex piece of software and offers more functionality than the application requires [128]. Hence, an attacker that can inject her own script code into the application can leverage these provided but unused methods to execute her exploit.

One way to remove the unused code of a shared library is to link it against the target application statically using Link Time Optimization (LTO). This allows the linker to remove the unnecessary code and thus reduce the availability of code snippets an attacker can choose from for a code-reuse attack. However, this increases the complexity of managing software updates: Since each application has to be compiled statically linked with all used shared libraries, each application has to be updated if a vulnerability is found in *one* of these libraries. To tackle this problem, Quach et al. [129] presented the concept of *piece-*

wise compilation and loading. It allows to compile an application and shared libraries with additional metadata to have a customized loader only load the needed code into memory. Unfortunately, the concept of this approach only works with shared libraries and does not apply to applications written in interpreted languages. As a result, in cases of interpreted applications, their approach only focuses on a single part of the software stack, the shared libraries, instead of considering the entire stack encompassing the interpreter and shared libraries.

6.1 Introduction

In this chapter, we present a first step towards *automatic application-specific software stacks*. Our goal is to customize the software stack for a given application (e.g., a web application or server application) such that only the required library code and underlying execution environment is contained within the software stack, hence *debloating* the software stack. To achieve this goal, we introduce a compiler extension capable of removing unused code from shared libraries, written in C. With information about which exported functions the target application uses, the compiler pass can omit functions at compile time from the shared library that are not used by the application or library itself. As a result, a shared library tailored specifically to the target application is created. To enhance usability, our approach can create shared libraries that are tailored to more than one application (e.g., a script interpreter and a web server). In contrast to a statically linked library, tailoring to a group of applications provides the same flexibility as a dynamically linked shared library given that only the shared library has to be re-compiled if a vulnerability in its code was discovered. When deployed with other existing defenses, such as Control-Flow Integrity (CFI) [1], an application-specific software stack further restricts the wiggle room an attacker can exploit to perform a successful attack.

Moreover, we show that—with the help of domain knowledge—this approach is also capable of removing unused functionalities in script interpreters when targeting an application written in an interpreted language (such as PHP or Ruby). Consider, for example, a Wordpress installation. With our approach, a PHP interpreter can be tailored to the concrete Wordpress web application. Since all unused functionalities are removed from the interpreter, an attacker that is able to inject script code (e.g., by uploading a script file) is no longer able to leverage them for their attack. Moreover, instead of removing unused functionalities in the interpreter, our approach allows us to replace them with *booby traps* [43], i.e., dormant code that, when executed, triggers an alarm. This way, an ongoing attack can be detected when a functionality that was removed is executed. Note that the shared libraries used by the Wordpress-specific PHP interpreter and the web server can also be compiled with our debloating approach, leading to an application-specific software stack. Regarding the recent trend to separate services into container (such as Docker [52]) to provide better security in case of a vulnerability, this makes tailoring shared libraries to specific server applications real-world deployable.

An application-specific script interpreter also allows reducing the attack surface significantly in environments in which untrusted scripts are executed (such as Google App Engine [71]). Usually, unwanted functionalities are disabled in configuration files. However,

since the code that provides these functionalities is still available in the script interpreter, an attacker might be able to bypass the restrictions and escape the interpreter's internal sandbox [109, 118]. When compiling the script interpreter in an application-specific way, the code for the unneeded functionalities is completely removed, which prevents an attacker from using them entirely.

We evaluated our prototype compiler pass for LLVM by tailoring two *libc* implementations (*musl-libc* and *uClibc*) to a diverse set of applications. The results show that, on average, the code for the *musl-libc* tailored to an application is reduced by 71.3%. A previous study on *libc* utilization [129] concluded that only 5% of code on average is used in the library. However, their evaluation set consists of mostly small applications, which explains this notable difference in comparison to our results. Additionally, we show that by using domain knowledge, our prototype is able to mitigate possible attacks on web applications: starting from *seven* security-critical PHP functions that might be used for *remote command execution* (according to the RIPS code analyzer [131]) in the interpreter, a PHP interpreter tailored to *OpenConf* or *FluxBB* only contains *one* sensitive PHP function. This significantly raises the bar for an attacker able to execute her own PHP code since using a removed PHP functionality triggers a booby trap and hence raises an alarm. In fact, in the case of *OpenConf*, our approach removes the possibility to execute shell commands from the interpreter in most system configurations due to the nature of the remaining sensitive PHP function. Additionally, we show the real-world applicability of our approach by creating a Docker container consisting of an application-specific software stack for a *Wordpress* installation. Our evaluation shows that the code of the *libc* used by the web server and PHP interpreter in this container is reduced by 65.1% in total, hence demonstrating that our debloating approach removes a significant fraction of unused code.

Contributions In summary, we provide the following contributions in this chapter:

- We present the design and implementation of an LLVM compiler pass capable of removing unused code from shared libraries and script interpreters written in C that effectively reduces the available code snippets for reuse attacks by debloating the software stack used by a given application.
- Our evaluation shows that, on average, 71.3% of the code in the *musl-libc* is removed when tailoring it to a target application. Moreover, when applying our approach to the PHP interpreter by targeting specific web applications, it is capable of eliminating entire vulnerability classes, such as *command execution*.

Outline This chapter is structured in the following way: Section 6.2 gives an overview of application-specific software stacks. Section 6.3 explains our analysis approach to tailor a shared library or script interpreter to an application. The implementation of our compiler extension is explained in Section 6.4, whereas the evaluation is presented in Section 6.5. We then discuss application-specific software stacks in Section 6.6 and give an overview of related work in Section 6.7. Section 6.8 ends this chapter with a conclusion and discussion of possible topics for future advancements.

The chapter results from a publication at the European Symposium on Research in Computer Security (ESORICS) 2019 [48] and was conducted together with Nicolai Davidsson and Thorsten Holz. Furthermore, it contains parts of the technical report [49] published 2019.

6.2 Overview

In this section, we give an overview of application-specific software stacks. First, we give a short description of static and dynamic linking and their differences. Next, we introduce our idea of application-specific software stacks.

6.2.1 Static and Dynamic Linking

Shared libraries offer developers a way to reuse already implemented functionalities in their program. These functionalities can either be *code* in the form of functions or *data* (e.g., global variables). For example, *libc* provides the developer with a variety of low-level functionalities (e.g., memory allocation and string processing). During compilation, there are two ways to couple the external functionalities with the own application: *static linking* and *dynamic linking*. In case of static linking, the external functionalities are resolved and plainly copied into the application during compilation. This means that no shared library is needed to execute the application since all library-provided functionalities are part of the program itself and hence available in memory. In case of dynamic linking, the external functionalities are replaced with a symbol which is resolved during the execution of the program. Hence, the shared libraries that provide the functionalities have to be present in memory to execute the application.

In practice, dynamic linking is used in most deployment scenarios. This allows the system to use the same shared library for multiple applications. Furthermore, having only one copy of the shared library improves usability during software patching: if a vulnerability is found in a function offered by a shared library, the user only needs to update the corresponding shared library. Since all dependent applications use this shared library, the vulnerability is fixed for all of them. In case of static linking, all applications using this functionality have to be updated to fix the vulnerability. As explained earlier, the main downside of using dynamic linking is the fact that this approach increases the amount of unused code that is mapped into the memory of the application. Therefore, sensible operations in functionalities not used by the application itself are also present in memory.

6.2.2 Application-Specific Software Stacks

The idea behind application-specific software stacks is based on the observation that applications do not use every functionality provided by their underlying software stack (e.g., interpreters or libraries). Therefore, it is safe to remove code of these unused functionalities to debloat the application without affecting it. Furthermore, by removing code snippets or whole functions that can potentially be used by an attacker in code-reuse attacks narrows down the options an attacker has. This also holds for scripting languages,

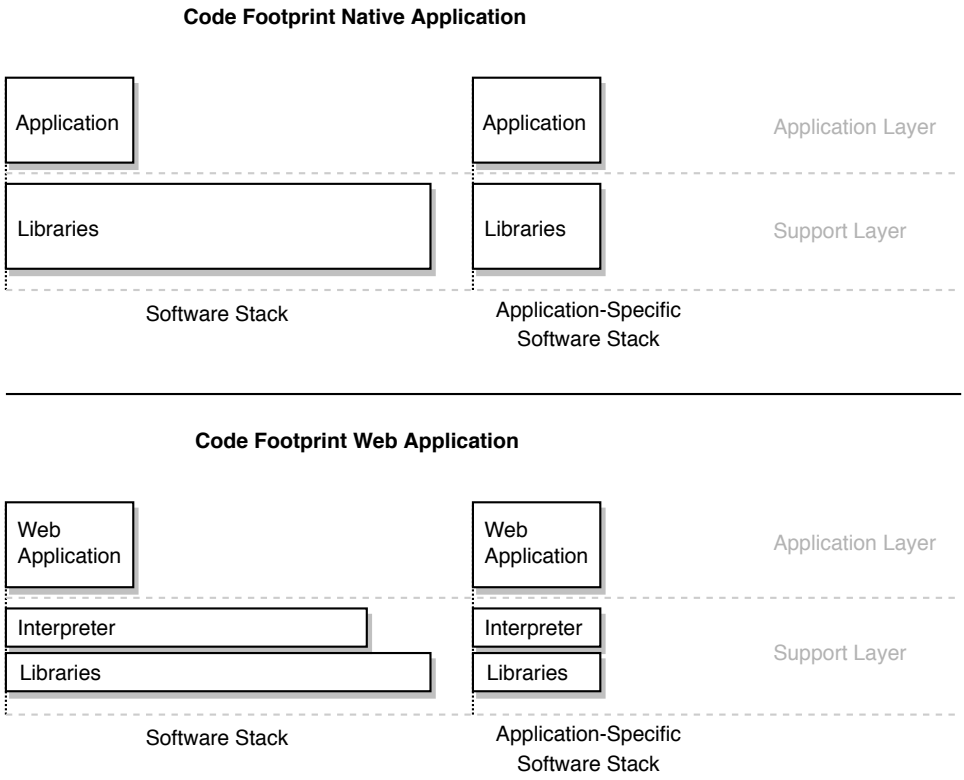


Figure 6.1: Schematic overview of the difference of the code footprint between a traditional software stack and an application-specific one showing the average code reduction of around 70%. On the top it is shown for a native code application, on the bottom for an interpreted application such as a web application. Even though the interpreter is part of the support layer, it also does not use the complete code provided by the shared libraries.

for example, in a web application context: the script interpreter offers more functionality than the web application uses. Stripping the interpreter from these functionalities debloats the interpreter, but does not interfere with the given web application. Moreover, in cases an attacker is able to insert her own script code (e.g., by uploading a script file to a web server), she is limited in the interpreter functionalities she can use. Figure 6.1 shows the difference between a normal software stack and an application-specific one graphically.

We define two layers for a software stack: the *application layer* and the *support layer*. The application itself resides on the application layer. This can either be a native code application or an application written in an interpreted language (e.g., web application). In a web application context, the application layer also includes the web framework the application uses. The libraries and script interpreter are located on the support layer. This layer provides functionalities that are used by the application. However, it also contains additional code and functionalities that are not used by the application. Underneath the support layer resides the operating system (OS). Functionalities provided by the OS are usually accessed via the support layer through low-level libraries such as the *libc*.

Our goal is to debloat the software stack by removing unneeded code from the support layer. This is done by analyzing the application and retrieving control transfers from the application layer into the support layer. This information is then used to recompile the support layer without the unused code. The result is a software stack tailored to the application. However, this approach is not limited to tailoring the support layer to only one application, thus increasing its usability. Consider for example a Wordpress installation. The libraries used by the web server and PHP interpreter can be specifically tailored to support both. Moreover, the PHP interpreter can be customized to only contain functionalities used by Wordpress. Hence, the debloating is achieved throughout the whole software stack by preserving the usability of shared libraries.

In the case of native code applications, the same code reduction can be achieved by using static linking with LTO during the compilation and linking process. As a result, the functionalities provided by the libraries and used by the application are directly inserted into the code of the program. This moves parts of the support layer directly into the application layer. However, this also means that the advantages of sharing libraries between multiple applications are also lost. As a result, as soon as a vulnerability is discovered in a library functionality, all applications using this library have to be updated. Application-specific software stacks, on the other hand, still provide the advantages of shared libraries. It is possible to group different applications to use one shared library tailored to them (as in our example a web server and script interpreter). Hence, our approach offers a middle ground between code reduction and usability.

6.3 Approach

In this section, we describe our approach for application-specific software stacks. We start by describing the basic method of our LLVM pass and refining it step-by-step throughout this section until each challenge encountered is tackled. The final goal in this chapter is to create a Wordpress installation with a tailored PHP interpreter and a *libc* implementation application-specific to the interpreter and web server. Hence, the described method focuses

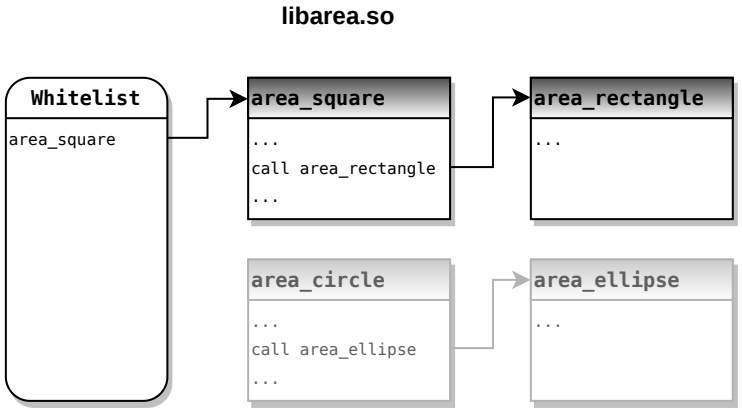


Figure 6.2: Example of the basic idea of the analysis. A target application uses the function `area_square`. Hence, the function `area_rectangle` is also added to the whitelist.

only on tailoring the libraries to a target application first. Afterwards, domain knowledge is used to enhance our approach to also support specific script interpreters, more specifically, the PHP and Ruby interpreter. Finally, we show the complete algorithm capable of handling shared libraries, PHP and Ruby interpreter.

6.3.1 Libraries

The control-flow transfer from the application layer into the support layer can be performed in multiple ways. In the easiest form, it is a direct call of a function. However, more complicated constructs such as indirect calls via function pointers are also possible. An analysis tailoring libraries to a specific application at compile time must not miss any of these, since one missing functionality leads to an uncompilable library in the best case, and a broken application in the worst. Next, we describe a method for LLVM capable of handling all these cases.

Base Method We start with a whitelist of functions, which initially contains all exported functions of the library used by the target application. The exported functions can be obtained by reading the metadata of the target application (e.g., with the help of the binutils tool `readelf`). Consider the example shown in Figure 6.2. The target application uses the function `area_square` of the library. During compilation, each currently processed function is checked if it resides in the whitelist. If the function `area_square` is processed, all direct control-flow transfers are also explored. Each new function that is reachable by the direct control-flow transfer is added to the whitelist and further explored. In this example, the function `area_rectangle` is added to the whitelist. This phase of searching for new reachable functions is called *function exploration*. Since this phase uses a depth-first search (DFS) approach, it is guaranteed to visit all functions that are reachable by an initial given function. Hence, all functions in the whitelist after the analysis is finished

are necessary for the application to work. All other functions can be safely removed. In the given example, `area_circle` and `area_ellipse` are dismissed.

Indirect Control-Flow Transfers Unfortunately, the compiler cannot always determine the target of a control-flow transfer. Often control-flow transfers are handled with the help of function pointers, i.e., through indirect call instructions. Therefore, we have to consider them during our analysis. Hence, we have to extend our approach to work with instructions handling function pointers. We found that the following LLVM intermediate representation (IR) instructions are capable of handling function pointers:

- `store`: storing data in a variable.
- `return`: returning data at the end of a function.
- `select`: chooses between two distinct values depending on a boolean condition.
- `phi`: merging multiple variables into a single variable for Single Static Assignment (SSA) form [45].

Since all these instructions can work with a function pointer, our analysis has to be able to process them. Therefore, we extend the *function exploration* phase to extract the data handled by these instructions to find all indirect control-flow transfers. If the extracted data is a function pointer, we continue the exploration at the pointer target.

This refined method handles all possible function pointers that are set inside the used code. However, since the semantics of the code are not considered, this analysis can overestimate the actually used functions. Consider for example a `select` instruction that chooses between two function pointers. When the boolean condition evaluates always to true, then only one function is ever reached by this code construct. Yet, our analysis considers both functions as reachable and thus overestimates the actually used functions. Note that this conservative overestimation guarantees us to not break the application.

Global Variables Although function pointers set directly in the code are already handled by our analysis, function pointers can also reside in global variables. Consider the code snippet from *musl-libc* shown in Figure 6.3a. The function `__stdout_write` is stored as function pointer in the global variable `stdout`. This variable is a `FILE` struct used for I/O operations. Figure 6.3b shows the function `fwrite` that uses a pointer to a `FILE` struct to invoke the *write* function stored there. Since the current form of our algorithm is not able to find the `__stdout_write` function pointer in the global variable, a valid call to `fwrite` with the global variable `stdout` as argument would break the application.

To handle global variables, we add a *global exploration* phase to our analysis. In this phase, all global variables are processed and checked for function pointers. If they contain a function pointer, the target is added to the whitelist as well. The *global exploration* phase is executed before the *function exploration* phase to guarantee that the newly whitelisted functions are also explored. A discussion about limitations of our function pointer analysis is given in Section 6.6.

```
static FILE f = {
    .write = __stdout_write,
    //...
};
FILE *const stdout = &f;
```

(a) Definition of a function pointer in a global variable from `src/stdio/stdout.c`.

```
size_t __fwritex(const unsigned char *restrict s, size_t l, FILE *restrict f)
{
    //...
    size_t n = f->write(f, s, i);
    //...
}

size_t fwrite(const void *restrict src, size_t size, size_t nmemb, FILE
    *restrict f)
{
    //...
    k = __fwritex(src, l, f);
    //...
}
```

(b) Possible usage of a function pointer from `src/stdio/fwrite.c`.

Figure 6.3: Code snippets from *musl-libc* that show the usage of function pointers in global variables.

6.3.2 Script Interpreters

Often applications written in scripting languages like PHP, Ruby, or Python are not translated into native code, but interpreted by the corresponding script interpreter. As a result, the interpreter itself is a part of the support layer for these applications. However, in contrast to the method described in Section 6.3.1 for native code libraries, the analysis cannot just remove code from the interpreter since it cannot distinguish which code belongs to a certain interpreter functionality. Hence, to build an application-specific interpreter, our analysis has to leverage domain knowledge about the internals of the target interpreter. More specifically, the analysis has to know the mapping of script functions to native code functions. To achieve our goal of running a Wordpress installation with an application-specific interpreter, we modify our analysis to work with the PHP interpreter in the following. To show that our approach is not limited to PHP, we further extend our algorithm to work with the Ruby interpreter.

PHP PHP stores information for each registered PHP function in global *function entries*, which are basically a map of structs [126]. The structs contain, among others, the pointer to the native code function and the name of the PHP function. During execution, they are used to handle the transition from PHP to native code. The interpreter uses these function entries to look up the native code function that is eventually executed to perform the application's desired functionality. Hence, modifying these function entries during the compilation of the PHP interpreter to remove the code from it is the best way to keep our approach as generic as possible. Since the function entries are part of the architecture of PHP, they are less likely to change between different PHP versions and hence our approach should be compatible with upcoming PHP releases.

To enable our analysis to remove PHP functionalities from the interpreter at compile time, we introduce a whitelist of PHP functions and modify the *global exploration* phase. The modification extracts the PHP function names from the PHP global *function entries* and checks if they are on the PHP whitelist. If they are, the corresponding native function is stored for processing during the *function exploration* phase. As a result, the native code corresponding to the functionality only remains in the interpreter when it is on the PHP whitelist.

PHP supports the paradigm of object-oriented programming, i.e., functions can be associated to classes. An example of a class and its member function directly provided by the PHP interpreter is the `Directory` class and its function `read` [147]. However, the PHP function name does not contain any information about the associated class. Hence, if multiple classes register a PHP function with the same name, our analysis is not able to distinguish between them. Consider an example where classes `A` and `B` both register a function with the name `read`. If the application only uses `A::read`, our analysis will still whitelist the `read` function of both classes. This loss in precision results in the PHP interpreter still containing functionality that is not needed, however, it guarantees to not break the application.

Ruby In contrast to PHP, Ruby does not register its functions through global function tables. Instead, Ruby functions are added by calling internal register functions at runtime.

```
void Init_IO(void)
{
    //...
    rb_define_global_function("syscall",
    rb_f_syscall,
    -1);
    rb_define_global_function("open",
    rb_f_open,
    -1);
    rb_define_global_function("printf",
    rb_f_printf,
    -1);
    //...
}
```

Figure 6.4: Ruby function registration example from `io.c`.

Listing 6.1: A list of Ruby's internal registration functions.

```
1  rb_define_protected_method
2  rb_define_private_method
3  rb_define_singleton_method
4  rb_define_method
5  rb_define_method_id
6  rb_define_module_function
7  rb_define_global_function
8  rb_define_alloc_func
9  rb_define_virtual_variable
10 rb_define_hooked_variable
```

Figure 6.4 shows how various I/O functions are registered in the Ruby source code. The arguments to the internal register function contain the name of the Ruby function and a pointer to its native code pendant. A list of all internal register functions can be found in Listing 6.1. To remove the functionality provided by Ruby functions, we aim to remove these registering function calls from the code. Again, our approach tries to focus on the architecture of the interpreter since this is less likely to change between versions and hence our approach should be compatible with upcoming Ruby releases.

To enable our analysis to remove Ruby functionalities from the interpreter at compile time, we added a whitelist for Ruby functions and modified the *function exploration* phase. Basically, the *function exploration* phase now checks if a call instruction calls an internal register function to register a Ruby functionality. If it does and the Ruby function name of the registration is whitelisted, the corresponding native code function is further explored. Otherwise, the corresponding call instruction is deleted from the code. As a result, only the code corresponding to whitelisted Ruby functions is part of the compiled

Ruby interpreter and since the call instruction is removed, the Ruby interpreter does not register the functionality at runtime and preserves its integrity.

This method works with Ruby functions registered directly in the code. However, dynamically registered Ruby functions are not detected. But since we did not encounter any dynamically registered Ruby functions in the core functionalities, we did not pursue it further.

6.3.3 Algorithm

The complete algorithm of our automated application-specific software stack approach as described in this section is given in Algorithms 1, 2, and 3. The LLVM compiler pass executes Algorithm 1 for each module that is compiled. This algorithm then uses Algorithm 2 for the *global exploration* to handle global variables and Algorithm 3 for the *function exploration* phase to process functions.

Algorithm 1 Starting part of our algorithm that is executed for each module that is compiled. The algorithm whitelists all needed functions during compilation. It is capable of handling shared libraries, the PHP interpreter and the Ruby interpreter.

```
global set whitelist
global set php_whitelist
global set ruby_whitelist

function VISIT_MODULE(module)
    // Start Global Exploration
    for all global in module.globals do
        EXPLORE_GLOBAL(global)                          // See Algorithm 2
    end for

    // Start Function Exploration
    for all func in module.funcs do
        if func in whitelist then
            EXPLORE_FUNCTION(func)                      // See Algorithm 3
        end if
    end for
end function
```

Algorithm 2 *Global exploration* phase of our algorithm that is executed to process global variables. The algorithm whitelists all needed functions during compilation. It is capable of handling shared libraries, the PHP interpreter and the Ruby interpreter.

```
function EXPLORE_GLOBAL(global)
    if is php and global is php function table then
        for all (func_name, func_ptr) in global do
            if func_name not in php_whitelist then
                global.delete(func_name, func_ptr)
            else
                whitelist.insert(func_ptr)
            end if
        end for
    else if global is struct then
        for all member in global do
            EXPLORE_GLOBAL(member)
        end for
    else if global is function pointer then
        whitelist.insert(global)
    end if
end function
```

Algorithm 3 *Function exploration* phase of our algorithm that is executed to process functions. The algorithm whitelists all needed functions during compilation. It is capable of handling shared libraries, the PHP interpreter and the Ruby interpreter.

```
function EXPLORE_FUNCTION(func)
    whitelist.insert(func)
    set targets

    for all instr in func do
        if instr.type == call then
            targets.insert(instr.target)
            if is ruby then
                if instr.target is ruby register function then
                    (func_name, func_ptr) = instr.args
                    if func_name in ruby_whitelist then
                        targets.insert(func_ptr)
                    else
                        delete instruction instr
                    end if
                end if
            end if
        else if instr.type == store then
            targets.insert(instr.store_value)
        else if instr.type == return then
            targets.insert(instr.return_value)
        else if instr.type == select then
            targets.insert(instr.true_value)
            targets.insert(instr.false_value)
        else if instr.type == phi then
            for all value in instr.incoming_values do
                targets.insert(value)
            end for
        end if
    end for

    for all target in targets do
        if target is function then
            if target not in whitelist then
                EXPLORE_FUNCTION(target)
            end if
        end if
    end for
end function
```

6.4 Implementation

Our prototype implementation resides in the compiler itself since it has to be able to modify the code and data structures directly (e.g., for the PHP interpreter). Hence, to build a tailored software stack for a given software, the whole support layer has to be recompiled using our compiler pass. The support layer consists of the libraries (and script interpreter) of the target application, and all libraries used by the libraries. Eventually, an application-specific software stack is created for the given software. For native code applications, the used exported functions have to be extracted as initial information for the compiler pass (e.g., with the help of the binutils tool `readelf`). For applications using script languages, the analysis to get all used interpreter functionalities has to be done by external tools like Parse [41] for PHP.

We built the prototype of our approach as compiler pass for LLVM 5.0.1. In total, our implementation consists of around 1,000 lines of C++ and 100 lines of Python code. To prevent possible dependency issues, each created module by LLVM is merged into one. This gives our compiler pass a global view of all existing code and data. Since our pass works on the LLVM IR, it is completely architecture and platform independent. Hence, each architecture that is supported by LLVM is also supported by our approach (e.g., ARM or MIPS).

In order to integrate it into the build process of an application as seamlessly as possible, we created a compiler wrapper script. This script is used as compiler for the application and handles all steps needed to perform our analysis.

6.4.1 Manual Configuration

Although our approach aims to automate the process in creating an application-specific software stack, a user might want to preserve certain functionalities in the libraries. This can have various reasons, e.g., using the same library by multiple applications. Hence, the user is able to modify the configuration file for the library and add additional function names to the whitelist. Furthermore, a library could need an additional whitelisted function which is not referenced directly from the application. This is the case for C entry functions (e.g., `_start`) which are directly called by the loader during load time.

Since LLVM does not lift assembly instructions into its IR, control-flow transfers to functions done in assembly are not detected by our analysis. Figure 6.5 shows an example encountered while compiling *musl-libc* for the target application. The control-flow transfer to function `__sigsetjmp_tail` is not detected by our analysis. We encountered five such cases in which assembly instructions in the code call a function not referenced in the rest of the code base (three in *musl-libc* and two in *uClibc*). Since we did not encounter any cases outside of the *libc*, we believe such cases more common in libraries providing low-level functionalities such as memory management and hence an exception.

Another case for manual configuration are functions that are resolved dynamically via loader functionalities such as `dlsym`. Since these functions do not have a reference in the code (either a direct reference or an indirect via a function pointer), our current prototype is not able to detect them. However, since we only encountered one case of dynamically resolved functions during our evaluation (`__dls3` in *musl-libc*), we believe this

```
sigsetjmp:
__sigsetjmp:
;...
.hidden __sigsetjmp_tail
jmp __sigsetjmp_tail
;...
```

Figure 6.5: Assembler instructions initiating a control-flow transfer found in `src/signal/x86_64/sigsetjmp.s`

feature to be rarely used in practice. Furthermore, this function was not resolved by loader functionalities, but by a self-implemented version of `dlsym` inside the *musl-libc*. This shows further how difficult it is to fully automate the process of creating an application-specific software stack and the reason for allowing manual configuration. A detailed discussion on how to address these cases in an automated way is given in Section 6.6.

6.4.2 Booby Trapping Script Interpreters

Most scripting languages offer ways to list all registered functions. An attacker able to execute script commands is therefore able to use this functionality as information leak to circumvent a removed functionality. For example, the PHP function `get_defined_functions` returns all functions registered to the interpreter. To thwart these attempts, our approach is not only able to remove functionality from the script interpreter, but to replace its native code implementation with a booby trap [43]. A booby trap contains code that when executed warns from an attack. Since this code lies dormant in memory and is never executed by the benign application, an execution of this code detects an altered control flow and hence an ongoing attack. When the native code implementation of a script function is replaced by this code, an attacker executing interpreter functionality that is not used by the application otherwise is detected. Furthermore, this removes any leak regarding the information about functions registered to the interpreter. If the attacker does not have access to the source code of the application (e.g., a proprietary application), this removes the possibility to circumvent booby traps.

6.5 Evaluation

As a target for our applications, we use Linux on the Intel x86-64 architecture because of its popularity as a server system. In this section, we first evaluate the effect of an application-specific software stack on the used shared libraries, afterwards a PHP interpreter tailored to specific web applications is measured. Subsequently, we study the code reduction of our approach on our running example: an application-specific software stack for a Wordpress installation. Finally, we perform a security evaluation of our approach on the basis of several CVEs and discuss the performance overhead.

6.5.1 Libraries

To evaluate the effect of our approach on native code applications, we compile different *libc* versions as an application-specific software stack. Unfortunately, the most common implementation *glibc* is written in GNU C, an extension of the C programming language which is not supported by LLVM [110]. Therefore, we resort to two other popular *libc* implementations: *musl-libc* (1.1.18) and *uClibc* (0.9.34). The *musl-libc* focuses on speed, feature-completeness, and simplicity [111]. It is used, for instance, by the Alpine Linux distribution, which is the distribution used for official Docker containers [34]. The *uClibc* implementation targets microcontrollers and therefore focuses mainly on size [8] (e.g., it is used by the buildroot project [25]). We compile both *libc* implementations without any changes by our transformation to have a complete shared library to compare against as an upper boundary. As a lower boundary, we compile both implementations using our approach with a minimal configuration which contains the least amount of functions necessary in the initial whitelist to compile the library (5 functions for *musl-libc* and 12 functions for *uClibc*).

To show the effect of an application-specific shared library, we compile the *libc* implementation for different applications: *Micro-Lisp*, *Nginx* (1.13.8), *Lighttpd* (1.4.48), *Busybox* (1.28), *PHP* (7.3.0-dev) for different web applications, and *Miniruby* (2.6.0-dev). To have a small basic PHP interpreter that supports all base features of our used web applications, we enabled support for Mysqli and zlib and disabled support for XML, iconv, PEAR, and DOM. Additionally, the PHP interpreter is also compiled in a minimal configuration (the least amount of functions necessary to run it) and in a complete configuration to better show the impact of an application-specific library. The *Ruby* interpreter has the option to build a smaller version of itself called *Miniruby*. This interpreter only contains the core functionalities (YARV instruction set [134]) of the *Ruby* interpreter. Since the difference between a complete *Miniruby* interpreter and a minimal *Miniruby* are smaller, it is more suited to show the impact of our approach than the full-fledged *Ruby* interpreter. For *Busybox*, we had to disable the coreutil functionalities: `date`, `echo`, `ls`, `mknod`, `mktemp`, `nl`, `stat`, `sync`, `test` and `usleep`. We were not able to compile *uClibc* with LLVM when these features were activated because of the dependency on buildroot. Hence, we had to modify the toolchain for *uClibc* to work without buildroot.

Code Reduction Table 6.1 depicts the results of our measurements. As evident from the table, the complete *musl-libc* has 2,603 functions, whereas a minimal configuration only needs 358 functions (13.8%) to be compilable. These configurations provide an upper and lower boundary of the code reduction that is possible for a target application. When tailoring the *musl-libc* to a specific application, *Micro-lisp* needs the fewest functions from the library with 14.1% remaining. In fact, this configuration needs only eight functions more than the minimal configuration which is necessary to compile the library. A complete *PHP* interpreter needs the most with 39.0%. On average, 30.3% of the functions remain in the *musl-libc* when tailored to an application. Since *uClibc* focuses on being as small as possible to work on microcontrollers, it does not have all features that the *libc* provides. Therefore, only *Busybox* and *Micro-Lisp* of our evaluation set work with this library. The complete library has 891 functions, whereas the minimal configuration only has 164

Table 6.1: Results of the remaining code for *musl-libc* and *uClibc*. On top for each library, the table shows the number of functions and code size for the complete and minimal library. The minimal library shows the remaining code for a configuration which contains the minimal number of functions to compile the library. Following the same metrics for the library tailored to a specific application.

Program	*# Funcs*	*%*	*Code Size*	*%*	**Program**	*# Funcs*	*%*	*Code Size*	*%*
musl-libc (complete)	2,603		1,007 kB		uClibc (complete)	891		450 kB	
musl-libc (minimal)	358	13.8	116 kB	11.5	uClibc (minimal)	164	18.4	108 kB	23.9
Micro-lisp	366	14.1	118 kB	11.7	*Micro-lisp*	168	18.9	115 kB	25.5
Busybox	893	34.3	345 kB	34.2	*Busybox*	388	43.6	329 kB	73.2
Nginx	762	29.3	276 kB	27.4					
Lighttpd	745	28.6	260 kB	25.9					
PHP (Complete)	1,014	39.0	390 kB	38.8					
PHP (FluxBB)	817	31.4	296 kB	29.4					
PHP (OpenConf)	839	32.2	326 kB	32.3					
PHP (Wordpress)	874	33.6	336 kB	33.4					
PHP (Minimal)	768	29.5	280 kB	27.8					
Miniruby (Complete)	907	34.8	325 kB	32.3					
Miniruby (Minimal)	684	26.3	221 kB	21.9					

(18.4%). A *uClibc* tailored to *Micro-lisp* has 168, which are 18.9% of all functions and only four functions more than the minimal configuration possible. The *Busybox* configuration has 43.6% functions remaining after its compilation. This shows that even a library focusing on being as small as possible can be further reduced by our approach. The code size confirms that the libraries did not only lose small wrapper-like functions, but that the code is reduced in a proportional way to the number of functions present.

Removing PHP functionalities from the interpreter also influences the code required in the underlying *libc*. A complete PHP interpreter has 39.0% of the functions available in the *musl-libc* remaining, whereas a minimal PHP interpreter only needs 29.5% of the functions in the library. A PHP interpreter tailored to the *Wordpress* web application, the largest web application of our evaluation set, needs only 33.6% of the functions of the *musl-libc*. On average, a PHP interpreter tailored to a web application needs only 32.4% of the functions. This shows that for software debloating it is imperative to not only focus on the shared libraries itself, but to take into account the actual application running when an interpreted language is used.

Code-Reuse Attacks A modern way for an attacker to exploit a vulnerability in an application is to reuse existing code. One way for an attacker is to transfer the control flow to an existing function in a library with crafted arguments and therefore execute the behavior the attacker desires (e.g., ret2libc attack [152]). However, since the number of existing functions in the library is significantly reduced, an attacker may not be able to find a function that executes the behavior she needs. For example, in all configurations listed in Table 6.1, except for *Busybox* for *uClibc*, the function `system` which is usually used to execute shell commands in an exploit is removed from the code.

Another way to reuse existing code for an attack is called return-oriented programming (ROP) [136]. For this exploiting technique, small code snippets called *gadgets* are com-

Table 6.2: Results of our gadget evaluation for *musl-libc* and *uClibc*. On top for each library, the table shows the number of unique ROP gadgets for the complete and minimal library. The minimal library shows the remaining gadgets for a configuration which contains the minimal number of functions to compile the library. Following the same metrics for the library tailored to a specific application.

Application	*# unique*	*%*	*# JOP*	*%*	*# COP*	*%*	*# CP*	*%*	*syscall*	*%*
musl-libc (complete)	9,692		332		324		581		157	
musl-libc (minimal)	1,578	16.3	40	12.1	106	32.7	108	18.6	81	51.6
Micro-lisp	1,581	16.3	36	10.8	113	34.9	110	18.9	81	51.6
Busybox	3,203	33.1	152	45.8	204	62.7	252	43.4	103	65.6
Nginx	3,196	33.0	105	31.6	166	51.2	209	36.0	106	67.5
Lighttpd	2,694	27.8	97	29.2	163	50.3	224	38.6	101	64.3
PHP (Complete)	4,012	41.4	130	39.2	235	72.5	281	48.4	106	67.5
PHP (FluxBB)	2,950	30.4	99	29.8	210	64.8	222	38.2	100	63.7
PHP (OpenConf)	3,387	35.0	101	30.4	201	62.0	226	38.9	97	61.8
PHP (Wordpress)	3,518	36.3	133	40.1	184	56.8	223	38.4	97	61.8
PHP (Minimal)	2,794	28.8	85	25.6	187	57.7	195	33.6	96	61.2
Miniruby (Complete)	3,533	36.5	97	29.2	181	55.9	237	40.8	112	71.3
Miniruby (Minimal)	2,578	26.6	59	17.8	176	54.3	181	31.2	104	66.2
uClibc (complete)	6,101		663		285		546		733	
uClibc (minimal)	1,736	28.5	87	13.1	75	26.3	142	26.0	150	20.5
Micro-lisp	1,724	28.3	82	12.4	77	27.0	146	26.7	150	20.5
Busybox	3,896	63.9	315	47.5	129	45.3	312	57.1	325	44.3

bined by the attacker to build the shellcode. Since an attacker needs a variety of different ROP gadgets to obtain the shellcode she needs, we measured the reduction of gadgets in the library with the tool *ROPgadget* [132] in version 5.6. While a tailored software stack alone does not prevent code-reuse attacks, this metric gives an estimate on the limitation an application-specific software stack imposes on ROP attacks. Besides measuring the number of unique ROP gadgets remaining, we also measured security-sensitive gadgets such as jump-oriented programming (JOP) [20], call-oriented programming (COP) [29], call-preceding gadgets (CP) [29], and syscall gadgets [136].

A minimal configuration of *musl-libc* and *uClibc* has only 16.3% and 28.5% of the unique ROP gadgets the complete library has. A tailored *musl-libc* has in the worst case 41.4% of unique ROP gadgets remaining for the complete PHP interpreter and in the best case 16.3% for *Micro-lisp*. For a tailored *uClibc*, 28.3% of the unique ROP gadgets remain for *Micro-lisp* and 63.9% for *Busybox*. Since *uClibc* is already optimized in regard to code size, the gadget reduction was to be expected less than the one for *musl-libc*. A full overview of all remaining gadgets is given in Table 6.2.

Overall, our evaluation shows that an application-specific library loses most of its code. The code size reduces proportionally to the number of functions removed. Furthermore, the number of unique ROP gadgets is reduced significantly, which narrows down the choices an attacker has when exploiting a vulnerability. While an application-specific software stack alone does not prevent code-reuse attacks, the combination of a tailored

Table 6.3: Results for PHP. The categories show the number of sensitive functions remaining in the PHP interpreter for each configuration. The special configurations *complete* and *minimal* give the numbers of sensitive functions for an unmodified PHP interpreter and a PHP interpreter containing the least number of functions to be executable.

	Base Interpreter		**Application-Specific Interpreter**		
	Complete	Minimal	FluxBB	OpenConf	Wordpress
Code Execution	5	0	3	2	3
Command Execution	7	0	1	1	4

software stack with other defenses (e.g., CFI) might restrict an attacker sufficiently to prevent exploitation.

6.5.2 Web Applications

To show the applicability of an application-specific software stack for applications using a script interpreter, we measure the impact of our approach on web applications, namely FluxBB (version 1.5.10, 21,295 LOC), OpenConf (version 6.80, 21,232 LOC), and Wordpress (version 4.9.1, 183,820 LOC). We focus on web applications for PHP and use the same interpreter as compiled for the evaluation in Section 6.5.1. To give a realistic overview, we have chosen web applications of different categories and sizes. To generate the initial whitelist of PHP functions as described in Section 6.3.2, we use the static analysis tool Parse [41]. Unfortunately, Parse does not support the paradigm of object-oriented programming, which leads to the necessity to add two additional functions to the initial whitelist for *FluxBB* (`dir` and `read`) and one for *Wordpress* (`mysqli_connect`).

Although modern web applications often provide a way to install additional plugins, we only evaluate our approach on the basic web applications to give a base line of removable functionalities. If someone wants to use specific plugins, these plugins only have to be included into the extraction of PHP functions for the initial whitelist to work with the resulting customized interpreter.

To evaluate the quality of the removed code, we measure the number of remaining sensitive functions in the script interpreter. We use the categories provided by the open-source version of RIPS, a static PHP security scanner [47]. Since the goal of an application-specific script interpreter is to reduce the impact of an attacker executing arbitrary PHP code (e.g., by uploading an attacker controlled script file), we focus on the categories *Code Execution* and *Command Execution*. *Code Execution* contains all functions that allow an attacker to execute arbitrary PHP functionality and *Command Execution* contains all functions that allow an attacker to execute shell commands on the host. Table 6.3 shows the full results, in the following we provide a high-level overview.

The base interpreter without any functions removed has five PHP functions in the *Code Execution* category (`assert`, `create_function`, `preg_filter`, `preg_replace`, and `preg_replace_callback`). In contrast, a minimal configuration of the interpreter (least amount of PHP functions necessary to run the interpreter itself) does not have any such

function. This shows that it is possible to remove this functionality completely from the interpreter as long as the target web application does not use one of the sensitive functions. Unfortunately, all projects use some *Code Execution* functionality and hence our approach is not able to remove it completely from the script interpreter with *FluxBB* using three different PHP functions, *OpenConf* two, and *Wordpress* three.

PHP functions that provide the ability to execute arbitrary shell commands on the host system are in the category *Command Execution*. A complete PHP interpreter provides seven such functions (`exec`, `passthru`, `popen`, `proc_open`, `shell_exec`, `system`, and `mail`) and a minimal configuration none. Unfortunately, each of the web applications of our evaluation set again uses at least one sensitive function from the category. For a *FluxBB* installation, the only PHP function allowing arbitrary shell command execution remaining is `exec`. However, since `exec` is only used to display the system's uptime in the administration control panel, removing it from the code would allow to remove the ability to execute shell commands completely from the script interpreter. Hence, an attacker that is able to upload her own script file to a web server is no longer able to execute shell commands. An *OpenConf* configuration has also only one PHP function remaining in the *Command Execution* category, the function `mail`. However, there are multiple limiting factors to consider before an attacker is able to execute shell commands with the help of `mail` which we discuss in Section 6.5.4 in detail. Hence, a tailored script interpreter for *OpenConf* removes the attack vector of *Command Execution* in most cases completely. A configuration for *Wordpress* has still four PHP functions that allow shell command execution. Here, the functionality still remains in the script interpreter and a malicious usage is only mitigated by the insertion of booby traps as explained in Section 6.4.2. An attacker not knowing about the tailored PHP interpreter that gains arbitrary PHP function execution could trigger a booby trap by executing a removed functionality.

In summary, an application-specific script interpreter reduces the available options for executing code or shell commands. Furthermore, it is also able to remove certain functionalities altogether and leave the attacker with no possibility to perform such an attack. In cases where the functionality still remains in the interpreter, it mitigates its malicious effects by inserting booby traps (which are especially effective in case of proprietary web applications) that can be triggered by an attacker using a removed functionality.

6.5.3 Use Case: Wordpress Container

To evaluate the debloating effect for a real-world scenario, we created a Docker container for our running example, an application-specific *Wordpress* installation. This container comprises of a PHP interpreter tailored to *Wordpress*, as well as a *musl-libc* tailored to the *Nginx* web server and PHP interpreter. Since the web server has to interact with the interpreter directly, PHP is additionally compiled with the FastCGI Process Manager (FPM). This scenario comprises a setting for which our approach was designed. One shared library tailored to multiple applications to keep the usability benefits of dynamic linking and a script interpreter customized for a web application.

The code reduction for the script interpreter is as discussed in Section 6.5.2. However, the reduction in the library is different since it is now tailored to two applications. The code of the *musl-libc* is reduced to 351 kB (34.9% of its original size). To put things in perspective, the *musl-libc* tailored solely to a *Wordpress* customized PHP interpreter has only 33.4% of its code remaining and a *Nginx*-specific library 27.4%. This suggests that

most of the library functions are shared by PHP and *Nginx*. Only 2.958 unique ROP gadgets were found (41.2% of the original amount). Even when comparing to a library specific to a complete PHP interpreter, this shared *musl-libc* setup results in a smaller library with less code.

In summary, this real-world setting shows a significant code reduction even with a library tailored to multiple applications. Since this code reduction restricts the options for an attacker performing an attack (e.g., whole function reuse, ROP, or PHP code execution), it is an important additional piece for a security-in-depth environment already providing other forms of defenses (e.g., CFI).

6.5.4 Security Evaluation

OpenConf 5.30 had multiple vulnerabilities that could be chained together to gain remote code execution [46]. This was achieved by injecting PHP code into an uploaded file and executing it. In an application-specific script interpreter for *OpenConf*, the attacker's possibilities are limited after gaining PHP code execution. The only remaining way to execute shell commands is by using the `mail` function which allows control over the arguments passed to the underlying sendmail command. However, before the arguments are passed to sendmail by the PHP interpreter, they are escaped internally. As a result, it is exploited by creating a file that can be abused as PHP shell and thus gain PHP code execution [69]. However, again the only remaining way for the attacker to execute shell commands with her created PHP shell is with the `mail` function. Hence, it is not possible for the attacker to execute any shell commands with the tailored PHP interpreter. The only exception is a system that uses the Exim mail server which allows a direct shell command execution with the `mail` function. Therefore, depending on the system configuration, an application-specific script interpreter would mitigate such an attack.

CVE-2016-5771 and CVE-2016-5773 in the PHP interpreter were found for Pornhub's bug bounty program in 2016 [74]. The penetration testers used it to exploit the `unserialize` function and gain remote code execution on the server. In their ROP shellcode, they used the function `zend_eval_string` to interpret a given string as PHP code. Although an application-specific PHP interpreter would not have eliminated this vulnerability (since the code was used by the web application), the exploiting could be made more difficult with it. For example, the native code function `zend_eval_string` is not present in any of our tailored interpreter instances (except the complete PHP interpreter). Additionally, when interpreting a string as PHP code, it might use a removed functionality and thus trigger a booby trap. Hence, depending on the used web application, the range of suitable candidates to use for an exploit can be limited.

6.5.5 Performance

Since our approach only removes unnecessary code from the support layer of the target application, it does not induce a performance penalty. However, it does not have a performance gain either, because only code is removed that is not executed by the application anyways. The memory consumption of an application-specific library is smaller than the consumption of the complete library, since code is removed from the binary and therefore

not loaded into memory. Nonetheless, since each group of applications need their own tailored library, the overall memory consumption of the system is increased. However, since using containers for each service (which also increase the memory consumption for each used library) gains more popularity, we deem it acceptable for practical deployments.

6.6 Discussion

Scripting languages often offer the possibility to dynamically evaluate code (such as `eval` in PHP). When used by the application, it makes the initial analysis to gather all necessary interpreter functionalities much harder. Our approach relies on the accuracy of specialized analysis tools for this. However, if the analysis tool is not able to provide accurate data, the tailored interpreter could break the application. Furthermore, if a user-provided input is directly passed to an evaluation function, stripping down the interpreter becomes impossible since the user can provide any programming construct she likes. However, such flawed code constructs allow direct access to the system anyway and trying to prevent it can be regarded as a losing battle.

As evident from our evaluation, an application-specific interpreter reduces the options an attacker has if she is able to execute own code in a targeted web application. Furthermore, it is able to remove certain vulnerability classes completely. However, if a web application uses a certain interpreter functionality that can also be used for an attack, our approach is not able to thwart this. To be more precise, if a web application relies on the PHP function `exec` to execute commands directly on the system (like in the case of *FluxBB*), our approach cannot remove it. To mitigate attacks using this functionality, approaches to monitor such remaining functions can be deployed additionally [143].

We showed that the concept of our approach is capable of working with script interpreters such as PHP and Ruby. However, as script interpreters have different internal structures, our approach cannot be used directly with another interpreter such as Python. To support it, domain knowledge of the interpreter's internal workings has to be integrated (i.e., the mapping of script functions to native code functions). As this merely means that additional engineering effort is needed to support other interpreters, it does not constitute a limitation of the general concept of our approach.

Another limitation is that each application needs its own customized libraries. As a result, when running multiple services like a web application in combination with a database server, both need their own tailored *libc* (or combine their analysis results to create one *libc* for both applications). On first glance, this seems infeasible for a real-world scenario. However, the recent trend to separate each part of a service into a container, such as Docker [52] (which uses Alpine Linux with *musl-libc* for official containers), makes our approach applicable for real-world scenarios. When running a web application, one container can contain the web server as well as a script interpreter (e.g., PHP) with a shared application-specific software stack and another container the database server with its own tailored software stack. Thus, enhancing the security mechanism of separating services with reduced options for an attacker to reuse existing code.

As the evaluation in Section 6.5 has shown, minor manual configuration is still necessary in some cases. For web applications these were cases where the used static analysis tool

Parse was not able to process object-oriented programming constructs. However, this is not a shortcoming of our approach, but just a limitation of the used analysis tool. Using a different analysis tool that is capable of handling object-oriented programming like RIPS [131] solves this problem. Minor manual configuration was also necessary for both tested *libc* versions. These were either cases that LLVM could not handle due to assembly, functions that are called by the loader, or functions that were resolved dynamically during runtime by the loader as explained in Section 6.4.1. These cases require more engineering work and do not constitute conceptual limitations of our approach. Assembly directly used in the source code can either be lifted to LLVM IR with tools such as McSema [151] or processed separately. Entry point functions called directly by the loader can be whitelisted initially by just adding the names of the C specific starting functions (e.g., `_start`). We did not do this to have a complete evaluation. Dynamically resolved functions can be addressed by integrating a data-flow analysis which ends in the corresponding library functions (e.g., `dlsym`). However, solving this in general is hard since the only case we encountered used a self-implemented function of the `dlsym` functionality to resolve the function pointer. Hence, our approach can be seen as a first step to an automated way to create application-specific software stacks.

An additional use case for our approach are restricted script interpreter environments that execute user provided untrusted scripts such as Google App Engine [71]. These script interpreters prevent internally the usage of specific sensitive functions from being executed. However, Park et al. [118] presented an attack with a restricted attacker model that is able to rewrite bytecode of functions to execute these sensitive functions and therefore bypassing the restriction. By applying our approach for an application-specific script interpreter, these restricted functionalities are completely removed from the interpreter and hence such an attack cannot use them.

Our current prototype focuses on removing unused code from shared libraries and script interpreters written in C. However, the compiler extension does not support shared libraries and script interpreters written in C++. To support C++, our approach has to be able to handle virtual function tables (vtables), which are used on a low-level to implement polymorphism. A naive approach would be to whitelist all functions that are part of a vtable. However, this would decrease the precision of the code debloating and heavily overestimates the used functions. A better way would be to improve the static analysis to only keep functions in the vtable that are actually used. For this to work correctly, our approach has to track the data flow of vtables precisely to identify all used functions and must be able to modify entries in the vtables to remove unused ones.

Our approach uses a flow-insensitive analysis to find function pointer targets with which we did not encounter any misses during our evaluation. However, the C programming language allows constructs that do not provide sufficient meaningful information in LLVM to determine the possible targets. In these edge cases, a more sophisticated points-to analysis has to be implemented like the one developed by Emami et al. [56].

6.7 Related Work

Debloating software is an appealing approach to thwart attacks and we now discuss works closely related to ours. Based on the observation that an application only uses a small part of the code provided by a shared library, Quach et al. [129] presented a debloating approach. They developed a compiler extension that adds metadata to an ELF binary (application and shared libraries) about the location of functions and their dependencies. On execution of an application, the loader writes the shared library into memory and then removes all functions that are not used by the application by overwriting them. However, though the analysis is similar to the presented one, their approach is only applicable to native code applications and does not work with applications written for a script interpreter.

JRed [82] is an automated approach to remove unused code from Java applications. It analyzes the bytecode of an application and removes unused code in the application itself and core libraries of the JRE. However, it is only capable of handling Java bytecode and ignores native code libraries during its analysis. Since JRed only targets Java bytecode, it does not tackle challenges like indirect control-flow transfers through function pointers as done by our approach. Landsborough et al. [94] presented an approach to remove unwanted functionalities from binary code by using a genetic algorithm. Since it works on traces obtained via dynamic analysis, it needs test cases that execute every functionality the target application should keep. If the set of test cases is not complete, the code corresponding to a needed but not tested functionality is removed and thus breaks the application. Additionally, it does not scale and did not even terminate when removing a feature from the *echo* application of coreutils. *Chisel* [76] aims to support programmers to debloat programs. It needs the source code and a high-level specification of its functionalities to remove unwanted features with the help of delta debugging. A similar goal is pursued by Sharif et al. [137] and their prototype implementation *TRIMMER*, a LLVM compiler extension. With the help of a user-provided manifest about the desired features, it tries to remove unwanted functionalities to debloat the application. A binary-only approach targeting specifically applications using a client-server architecture is presented by Chen et al. [32]. Their approach uses binary-rewriting techniques and a user-provided list of features with corresponding test cases to execute those to customize the target application. *BinRec* [91] also aims at debloating already compiled applications. It is based on LLVM and needs to lift the target binary into the LLVM IR before it can perform its transformations. Since automatically removing features from an application on the binary level is prone to errors, *BinRec* also provides a fallback mechanism to use removed code from the original binary. In contrast to our approach, these approaches focus on removing features from a target application itself, while we aim to remove unused functionalities from libraries and script interpreters.

An approach to debloat the Linux kernel was presented by Kurmus et al. [92]. Their approach focuses on optimizing the configuration for the Linux kernel to remove unnecessary features at compile time. This work is orthogonal to ours and can further improve the security of the system by not only tailoring the userspace software stack in an application-specific way, but also optimizing the Linux kernel to target a specific application.

6.8 Conclusion and Future Work

In this chapter, we presented an approach to compile shared libraries tailored to a specific application by removing unused code from them. Since large real-world applications, such as the PHP interpreter, do not even use half of the provided functions in a shared library, we showed that debloating reduces the choices an attacker has for code-reuse attacks significantly. Furthermore, we demonstrated that with the help of domain knowledge, our approach is also capable of tailoring a script interpreter to a script application (i.e., a web application).

We evaluated our prototype implementation of the proposed approach on large, real-world native applications using the *libc* and on large web applications using the PHP interpreter. We demonstrated an application-specific software stack tailored to a *Wordpress* installation (customized PHP interpreter, *libc* tailored to web server and interpreter), and showed a significant code reduction.

In terms of future work, it would be interesting to add C++ support to our compiler extension. This requires to support object-oriented programming constructs as well as exception handling. Since popular applications, such as the JavaScript run-time environment Node.js, are written in C++, this extension would allow debloating of a broader set of programs.

Chapter 7

Conclusion

In this thesis, we explored new automated program analysis approaches that can help to secure software systems. To this end, we developed four new analysis techniques that identify constructs that are relevant for hardening the program.

First, we presented the design of an analysis to find PRNG and CHF implementations in binary executables reliably. This analysis approach works generically without relying on magic constants or other implementation-specific artifacts. To this end, we investigated various PRNG and CHF implementations in popular cryptographic shared libraries and created an interaction model encompassing both algorithm types. Based on these insights, we implemented a prototype called *TropyHunter*. We thoroughly evaluated *TropyHunter* on a diverse set of open- and closed-source programs demonstrating its accuracy. This work showed that it is possible to identify PRNG and CHF implementations fully automated in a binary program and, thus, automates the initial step of a security assessment. Since our approach works in a generic way, it also can detect future algorithms without modifications.

One limitation encountered during the evaluation of *TropyHunter* was that analysis approaches working mostly in a static way have problems handling function pointers. Since C++ code utilizing polymorphism is implemented with function pointers on the binary level, we developed an analysis approach reconstructing C++ class hierarchies in a practical and efficient way. To this end, we created a way to find vtables in binary programs which contain the function pointers used by polymorphic constructs. With the help of these vtables, we developed a static analysis technique tracking the data flow of objects through the binary while taking C++ characteristics for polymorphism into account. Based on these characteristics, we were able to deduce the relationship of classes and hence recover their hierarchies. We implemented a prototype called *Marx* targeting large applications based on these analysis techniques. The evaluation showed the precision of the class-hierarchy reconstruction and applicability to real-world applications (e.g., MySQL Server). To demonstrate the analysis results are precise enough to build security applications on top of it, we created two defenses utilizing them: *vtable protection* and *type-safe object reuse*. Overall, this work showed an automated way to reconstruct C++ constructs that makes the analysis process of binary executables more involved for an

analyst and even impossible for most analysis tools. Hence, this work provides a building block for future analysis techniques to build upon.

Binary-only defenses that target to restrict virtual callsites of C++ programs to a set of valid targets still leave wiggle room for an attacker. Although the evaluation of *Marx* showed an improvement for these valid target sets, an imprecision remains. Therefore, we developed VPS, a binary-only defense mechanism against C++ vtable-hijacking attacks. In contrast to other related work constraining virtual callsites to a set of valid targets, VPS restricts virtual callsites to only use objects created at valid creation sites (i.e., constructors). This sidesteps the accuracy problems of binary analyses and by protecting objects at creation time generates a direct mapping between object initialization sites and reachable virtual callsites. To extract the information needed for this defense from binary executables, we refined *Marx*'s vtable identification approach. Additionally, we created novel analysis techniques to identify virtual callsites precisely. While related work assumes the absence of false-positive virtual callsites that would break the application, we developed a dynamic instrumentation approach capable of handling false positives by analyzing not yet verified virtual callsites. The evaluation demonstrated the accuracy of VPS and, thus, the high level of protection it provides. Furthermore, it uncovered imprecisions in the source-based approach *VTV* which is considered among the state-of-the-art for C++ defenses. Overall, this work showed that it is possible for binary-only approaches capable of handling C++ low-level constructs to almost reach the degree of protection that a source-based approach has. Additionally, it demonstrated the importance of handling false-positive analysis results to not break the application.

Finally, we described a novel approach capable of reducing unused code in shared libraries and script interpreters. While the previously presented defense approaches were focusing on preventing the exploitation of a vulnerability via vtable-hijacking attacks, this approach aims to remove the available code an attacker can leverage for code-reuse attacks. To this end, we created a compiler pass capable of removing unused code from shared libraries during compilation time without resorting to static linking. Moreover, with the help of domain knowledge, this compiler pass can also remove unused functionalities from script interpreters. The evaluation demonstrated the effectiveness of the code reduction: large applications such as the PHP interpreter do not even use half of the provided functions in the *libc*. Furthermore, the evaluation showed that in some instances entire classes of functionalities in an interpreter, such as executing shell commands, can be removed completely. Overall, this work provides a method of combining code reduction and the manageability of software updates. It further demonstrated that it is possible to remove unused code from an interpreter.

Publications

During the development on this thesis the author contributed to the following publications:

Peer-reviewed Publications

- Towards Automated Application-Specific Software Stacks
  Nicolai Davidsson, Andre Pawlowski, Thorsten Holz
  In *European Symposium on Research in Computer Security (ESORICS)*, 2019

- VPS: Excavating High-Level C++ Constructs from Low-Level Binaries to Protect Dynamic Dispatching
  Andre Pawlowski, Victor van der Veen, Dennis Andriesse, Erik van der Kouwe, Thorsten Holz, Cristiano Giuffrida, Herbert Bos
  In *Annual Computer Security Applications Conference (ACSAC)*, 2019

- On the Weakness of Function Table Randomization
  Moritz Contag, Robert Gawlik, Andre Pawlowski, Thorsten Holz
  In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2018

- How They Did It: An Analysis of Emission Defeat Devices in Modern Automobiles
  Moritz Contag, Guo Li, Andre Pawlowski, Felix Domke, Kirill Levchenko, Thorsten Holz, Stefan Savage
  In *IEEE Symposium on Security and Privacy (S&P)*, 2017

- MARX: Uncovering Class Hierarchies in C++ Programs
  Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, Cristiano Giuffrida
  In *Network & Distributed System Security Symposium (NDSS)*, 2017

- A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level
  Victor van der Veen, Enes Goktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, Cristiano Giuffrida
  In *IEEE Symposium on Security and Privacy (S&P)*, 2016

- Detile: Fine-Grained Information Leak Detection in Script Engines
  Robert Gawlik, Philipp Koppe, Benjamin Kollenda, Andre Pawlowski, Behrad Garmany, Thorsten Holz
  In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016

- Probfuscation: An Obfuscation Approach using Probabilistic Control Flows
  Andre Pawlowski, Moritz Contag, Thorsten Holz
  In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016

- A Practical Investigation of Identity Theft Vulnerabilities in Eduroam
  Sebastian Brenza, Andre Pawlowski, Christina Pöpper
  In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2015

Technical Reports

- Towards Automated Application-Specific Software Stacks
  Nicolai Davidsson, Andre Pawlowski, Thorsten Holz
  Technical Report, arXiv:1907.01933, 2019

- Technical Report: Detile: Fine-Grained Information Leak Detection in Script Engines
  Robert Gawlik, Philipp Koppe, Benjamin Kollenda, Andre Pawlowski, Behrad Garmany, Thorsten Holz.
  Ruhr-Universität Bochum, Horst Gŕtz Institute (HGI), 2016

- Technical Report: Probfuscation: An Obfuscation Approach using Probabilistic Control Flows
  Andre Pawlowski, Moritz Contag, Thorsten Holz
  Ruhr-Universität Bochum, Horst Gŕtz Institute (HGI), 2016

Bibliography

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] Ali Abbasi, Thorsten Holz, Emmanuele Zambon, and Sandro Etalle. ECFI: Asynchronous Control Flow Integrity for Programmable Logic Controllers. In *Annual Computer Security Applications Conference (ACSAC)*, 2017.

[3] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. Challenges in Designing Exploit Mitigations for Deeply Embedded Systems. In *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2019.

[4] Adobe. Adobe Flash Player. `https://get.adobe.com/de/flashplayer/`, Accessed: September 16, 2019; 2019.

[5] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers - Principles, Techniques & Tools*. Pearson Addison Wesley, 2007.

[6] Periklis Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *USENIX Security Symposium*, 2010.

[7] Frances E Allen. Control Flow Analysis. In *ACM Sigplan Notices*, 1970.

[8] Erik Andersen. uClibc. `https://www.uclibc.org`, Accessed: 13th August, 2018; 2018.

[9] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Security Symposium*, 2016.

[10] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *USENIX Security Symposium*, 2016.

[11] Vard Antinyan, Anna B Sandberg, and Miroslaw Staron. A Pragmatic View on Code Complexity Management. *Computer*, 52(2):14–22, 2019.

[12] ARM. C++ ABI for the ARM Architecture. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0041e/IHI0041E_cppabi.pdf`, Accessed: 16th September, 2019; 2015.

[13] Arm. Arm® Architecture Reference Manual. *Armv8, for Armv8-A architecture profile*, 2019.

[14] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[15] Elaine Barker and John Kelsey. NIST SP 800-90A – Recommendation for Random Number Generation Using Deterministic Random Bit Generators. Technical report, National Institute of Standards and Technology, 2015.

[16] David Berard. IDA pro plugin to find crypto constants (and more). `https://github.com/polymorf/findcrypt-yara`, Accessed: July 16, 2019; 2017.

[17] Andrew R. Bernat and Barton P. Miller. Anywhere, Any-Time Binary Instrumentation. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2011.

[18] Daniel J Bernstein. ChaCha, a variant of Salsa20. In *Workshop Record of the State of the Art of Stream Ciphers (SACS)*, 2008.

[19] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating Code-reuse Attacks with Control-Flow Locking. In *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[20] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[21] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[22] Bruno O Brachtl, Don Coppersmith, Myrna M Hyden, Stephen M Matyas Jr, Carl H. W. Meyer, Jonathan Oseas, Shaiy Pilpel, and Michael Schilling. Data authentication using modification detection codes based on a public one way encryption function, March 13 1990. US Patent 4,908,861.

[23] Sebastian Brenza, Andre Pawlowski, and Christina Pöpper. A Practical Investigation of Identity Theft Vulnerabilities in Eduroam. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, 2015.

[24] Robert G. Brown. Dieharder: A Random Number Test Suite. `https://webhome.phy.duke.edu/~rgb/General/dieharder.php`, Accessed: March 21, 2019; 2019.

[25] Buildroot. Buildroot - Making Embedded Linux Easy. `https://buildroot.org`, Accessed: 13th August, 2018; 2018.

[26] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. CFIXX: Object Type Integrity for C++ Virtual Dispatch. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[27] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[28] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security Symposium*, 2015.

[29] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *USENIX Security Symposium*, 2014.

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

[31] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[32] Yurong Chen, Shaowen Sun, Tian Lan, and Guru Venkataramani. TOSS: Tailoring Online Server Systems through Binary Feature Customization. In *Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2018.

[33] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for In-Vivo Multi-Path Analysis of Software Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[34] Brian Christner. Docker Official Images are Moving to Alpine Linux. `https://www.brianchristner.io/docker-is-moving-to-alpine-linux/`, Accessed: 16th September, 2019; 2016.

[35] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting Bare-Metal Embedded Systems With Privilege Overlays. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[36] Shaanan N Cohney, Matthew D Green, and Nadia Heninger. Practical state recovery attacks against legacy RNG implementations. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[37] Moritz Contag, Robert Gawlik, Andre Pawlowski, and Thorsten Holz. On the Weaknesses of Function Table Randomization. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2018.

[38] Moritz Contag, Guo Li, Andre Pawlowski, Felix Domke, Kirill Levchenko, Thorsten Holz, and Stefan Savage. How They Did It: An Analysis of Emission Defeat Devices in Modern Automobiles. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[39] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[40] Jonathan Corbet. Memory protection keys. `https://lwn.net/Articles/643797/`, Accessed: September 16, 2019; 2015.

[41] Chris Cornutt. Parse: A PHP Security Scanner. `https://github.com/psecio/parse`, Accessed: 3rd July, 2018; 2018.

[42] 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.

[43] Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. Booby Trapping Software. In *ACM New Security Paradigms Workshop (NSPW)*, 2013.

[44] Stephen J 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 *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[45] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1991.

[46] Johannes Dahse. OpenConf 5.30 - Multi-Step Remote Command Execution. `https://blog.ripstech.com/2016/openconf-multi-step-remote-command-execution/`, Accessed: 16th September, 2019; 2016.

[47] Johannes Dahse. RIPS Sensitive Sinks. `https://github.com/ripsscanner/rips/blob/master/config/sinks.php`, Accessed: 15th August, 2018; 2018.

[48] Nicolai Davidsson, Andre Pawlowski, and Thorsten Holz. Towards Automated Application-Specific Software Stacks. In *European Symposium on Research in Computer Security (ESORICS)*, 2019.

[49] Nicolai Davidsson, Andre Pawlowski, and Thorsten Holz. Towards Automated Application-Specific Software Stacks. In *arXiv preprint arXiv:1907.01933*, 2019.

[50] Theo de Raadt. OpenBSD 3.3. `http://www.openbsd.org/33.html`, Accessed: November 7, 2019; 2003.

[51] Valgrind Developers. Valgrind. `http://www.valgrind.org/`, Accessed: July 17, 2018; 2018.

[52] Docker Inc. Docker - Build, Ship, and Run Any App, Anywhere. `https://www.docker.com/`, Accessed: 16th September, 2019; 2019.

[53] Morris J. Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. Technical report, National Institute of Standards and Technology, 2015.

[54] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons, 2011.

[55] Mohamed Elsabagh, Dan Fleck, and Angelos Stavrou. Strict Virtual Call Integrity Checking for C++ Binaries. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2017.

[56] Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[57] Max Fillinger and Marc Stevens. Reverse-Engineering of the Cryptanalytic Attack Used in the Flame Super-Malware. In *Advances in Cryptology—ASIACRYPT*, 2015.

[58] Agner Fog. Calling conventions for different C++ compilers and operating systems. `http://agner.org/optimize/calling_conventions.pdf`, Accessed: August 7, 2018; 2018.

[59] Alexander Fokin, Katerina Troshina, and Alexander Chernov. Reconstruction of Class Hierarchies for Decompilation of C++ Programs. In *European Conference on Software Maintenance and Reengineering (CSMR)*, 2010.

[60] Linux Foundation. Itanium C++ ABI. `http://refspecs.linuxbase.org/cxxabi-1.83.html`, Accessed: 16th September, 2019; 2019.

[61] Gael H. Find out about Intel's new RdRand Instruction. `https://software.intel.com/en-us/blogs/2011/06/22/find-out-about-intels-new-rdrand-instruction`, Accessed: August 12, 2019; 2011.

[62] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, Heyuan Shi, and Jiaguang Sun. VulseekerPro: Enhanced Semantic Learning Based Binary Vulnerability Seeker with Emulation. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018.

[63] 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.

[64] Robert Gawlik, Philipp Koppe, Benjamin Kollenda, Andre Pawlowski, Behrad Garmany, and Thorsten Holz. Detile: Fine-Grained Information Leak Detection in

Script Engines. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.

[65] Xinyang Ge, Mathias Payer, and Trent Jaeger. An Evil Copy: how the Loader Betrays You. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[66] Patrice Godefroid. Micro execution. In *International Conference on Software Engineering (ICSE)*, 2014.

[67] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[68] Enes Goktas, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermining Information Hiding (And What to do About it). In *USENIX Security Symposium*, 2016.

[69] Dawid Golunski. Pwning PHP mail() function For Fun And RCE - New Exploitation Techniques And Vectors - Release 1.0. `https://exploitbox.io/paper/Pwning-PHP-Mail-Function-For-Fun-And-RCE.html`, Accessed: 16th September, 2019; 2017.

[70] Google. Protocol Buffers. `https://developers.google.com/protocol-buffers/`, Accessed: September 16, 2019; 2018.

[71] Google Cloud. Google Cloud: App Engine - Build Scalable Web & Mobile Backends in Any Language. `https://cloud.google.com/appengine/`, Accessed: 16th September, 2019; 2019.

[72] Jan Gray. C++: Under the Hood. `http://www.openrce.org/articles/files/jangrayhood.pdf`, Accessed: 16th September, 2019; 1994.

[73] Felix Gröbert, Carsten Willems, and Thorsten Holz. Automated Identification of Cryptographic Primitives in Binary Programs. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2011.

[74] Ruslan Habalov. How we broke PHP, hacked Pornhub and earned $20,000. `https://www.evonide.com/how-we-broke-php-hacked-pornhub-and-earned-20000-dollar/`, Accessed: 16th September, 2019; 2016.

[75] Istvan Haller, Enes Göktaş, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. Shrinkwrap: VTable protection without loose ends. In *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[76] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective Program Debloating via Reinforcement Learning. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[77] Jesse Hertz. Project Triforce: Run AFL on Everything! `https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/`, Accessed: August 7, 2018; 2018.

[78] Hex Rays. IDA: About. `https://www.hex-rays.com/products/ida/`, Accessed: July 11, 2019; 2019.

[79] IDAPython. IDAPython. `https://github.com/idapython`, Accessed: March 21, 2019; 2019.

[80] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual. *Volume 2: Instruction Set Reference, A-Z*, 2016.

[81] Ciji Isen and Lizy John. On the Object Orientedness of C++ programs in SPEC CPU 2006. In *SPEC Benchmark Workshop*. Citeseer, 2008.

[82] Yufei Jiang, Dinghao Wu, and Peng Liu. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *Computer Software and Applications Conference (COMPSAC)*, 2016.

[83] Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya Narasimhan. Recovering C++ Objects From Binaries Using Inter-Procedural Data-Flow Analysis. In *ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*, 2014.

[84] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC PRESS, 2007.

[85] Omer Katz, Ran El-Yaniv, and Eran Yahav. Estimating Types in Binaries using Predictive Modeling. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2016.

[86] Omer Katz, Noam Rinetzky, and Eran Yahav. Statistical reconstruction of class hierarchies in binaries. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[87] John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator. In *International Workshop on Selected Areas in Cryptography (SAC)*, 1999.

[88] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[89] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *European Conference on Computer Systems (EuroSys)*, 2017.

[90] Alexey Kopytov. SysBench. `https://github.com/akopytov/sysbench`, Accessed: 16th September, 2019; 2019.

[91] Taddeus Kroes, Anil Altinay, Joseph Nash, Yeoul Na, Stijn Volckaert, Herbert Bos, Michael Franz, and Cristiano Giuffrida. BinRec: Attack Surface Reduction Through Dynamic Binary Recovery. In *Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2018.

[92] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Symposium on Network and Distributed System Security (NDSS)*, 2013.

[93] 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.

[94] Jason Landsborough, Stephen Harding, and Sunny Fugate. Removing the Kitchen Sink from Software. In *Conference on Genetic and Evolutionary Computation (GECCO)*, 2015.

[95] Immo Landwerth. .NET Core 3.0 concludes the .NET Framework API porting project. `https://github.com/dotnet/announcements/issues/130`, Accessed: October 16, 2019; 2019.

[96] 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.

[97] Meir M Lehman. On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle. *Journal of Systems and Software*, 1979.

[98] Thomas Lendacky. x86/CPU/AMD: Clear RDRAND CPUID bit on AMD family 15h/16h. `https://lore.kernel.org/patchwork/patch/1115413/`, Accessed: August 20, 2019; 2019.

[99] Pierre Lestringant, Frédéric Guihéry, and Pierre-Alain Fouque. Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.

[100] Julian Lettner, Benjamin Kollenda, Andrei Homescu, Per Larsen, Felix Schuster, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, and Michael Franz. Subversive-C: Abusing and Protecting Dynamic Message Dispatch. In *USENIX Annual Technical Conference*, 2016.

[101] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[102] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2010.

[103] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[104] 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.

[105] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 1998.

[106] Robert McEvoy, James Curran, Paul Cotter, and Colin Murphy. Fortuna: Cryptographically Secure Pseudo-Random Number Generation In Software And Hardware. In *IET Irish Signals and Systems Conference (ISSC)*, 2006.

[107] Tom Mens. On the Complexity of Software Systems. *Computer*, 45(8):79–81, 2012.

[108] Microsoft. Data Execution Prevention. `https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention?redirectedfrom=MSDN`, Accessed: November 7, 2019; 2018.

[109] mm0r1. PHP 7.0-7.3 disable_functions bypass. `https://github.com/mm0r1/exploits/tree/master/php7-gc-bypass`, Accessed: October 9, 2019; 2019.

[110] Marius Muench, Fabio Pagani, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna, and Davide Balzarotti. Taming Transactions: Towards Hardware-Assisted Control Flow Integrity using Transactional Memory. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2016.

[111] musl libc. musl libc. `https://www.musl-libc.org`, Accessed: 13th August, 2018; 2018.

[112] Robert Muth and Saumya Debray. On the Complexity of Function Pointer May-Alias Analysis. In *Theory and Practice of Software Development (TAPSOFT)*, 1997.

[113] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2004.

[114] Ben Niu and Gang Tan. Modular Control-Flow Integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[115] United States Government Accountability Office. Federal Agencies Need to Address Aging Legacy Systems. `https://www.gao.gov/assets/680/677436.pdf`, Accessed: September 16, 2019; 2016.

[116] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking Holes in Information Hiding. In *USENIX Security Symposium*, 2016.

[117] Christof Paar and Jan Pelzl. *Understanding Cryptography – A Textbook for Students and Practitioners*, chapter 11, pages 296–303. Springer, 2011.

[118] Taemin Park, Julian Lettner, Yeoul Na, Stijn Volckaert, and Michael Franz. Bytecode Corruption Attacks Are Real—And How to Defend Against Them. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2018.

[119] Andre Pawlowski, Moritz Contag, and Thorsten Holz. Probfuscation: An Obfuscation Approach using Probabilistic Control Flows. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.

[120] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. MARX: Uncovering Class Hierarchies in C++ Programs. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[121] Andre Pawlowski, Victor van der Veen, Dennis Andriesse, Erik van der Kouwe, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. VPS: Excavating High-Level C++ Constructs from Low-Level Binaries to Protect Dynamic Dispatching. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[122] PaX Team. Address Space Layout Randomization. `https://pax.grsecurity.net/docs/aslr.txt`, Accessed: August 26, 2019; 2001.

[123] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. X-Force: Force-Executing Binary Programs for Security Applications. In *USENIX Security Symposium*, 2014.

[124] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[125] Pieter Philippaerts, Yves Younan, Stijn Muylle, Frank Piessens, Sven Lachmund, and Thomas Walter. Code Pointer Masking: Hardening Applications against Code Injection Attacks. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2011.

[126] PHP Internals Book. Registering and using PHP functions. `http://www.phpinternalsbook.com/php7/extensions_design/php_functions.html`, Accessed: 3rd July, 2018; 2018.

[127] 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.

[128] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. A Multi-OS Cross-Layer Study of Bloating in User Programs, Kernel and Managed Execution Environments. In *Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2017.

[129] Anh Quach, Aravind Prakash, and Lok Kwong Yan. Debloating Software through Piece-Wise Compilation and Loading. In *USENIX Security Symposium*, 2018.

[130] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[131] RIPS Technologies GmbH. PHP Security Analysis - RIPS. `https://www.ripstech.com/`, Accessed: 3rd July, 2018; 2018.

[132] Jonathan Salwan. ROPgadget. `https://github.com/JonathanSalwan/ROPgadget`, Accessed: 13th August, 2018; 2018.

[133] Pawel Sarbinowski, Vasileios P. Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. VTPin: Practical VTable Hijacking Protection for Binaries. In *Annual Computer Security Applications Conference (ACSAC)*, 2016.

[134] Koichi Sasada. YARV: Yet Another RubyVM: Innovating the Ruby Interpreter. In *Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[135] 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.

[136] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[137] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. TRIMMER: Application Specialization for Code Debloating. In *International Conference on Automated Software Engineering (ASE)*, 2018.

[138] S. Shen and X. Lee. SM3 Hash function. `https://tools.ietf.org/html/draft-shen-sm3-hash-00`, Accessed: June 19, 2019; 2011.

[139] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[140] Jonas Skeppstedt. *An Introduction to the Theory of Optimizing Compilers*. Skeppberg, 2012.

[141] SPEC. SPEC CPU2006. `https://www.spec.org/cpu2006`, Accessed: July 6, 2018; 2018.

[142] SPEC. SPEC CPU2017. `https://www.spec.org/cpu2017`, Accessed: July 6, 2018; 2018.

[143] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. Synode: Understanding and Automatically Preventing Injection Attacks on Node.js. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[144] Bjarne Stroustrup. *The C++ Programming Language*. Pearson Education, 2013.

[145] Bjarne Stroustrup. C++ Applications. `http://www.stroustrup.com/applications.html`, Accessed: September 16, 2019; 2019.

[146] The Apache Software Foundation. Apache benchmark. `http://httpd.apache.org/docs/2.0/programs/ab.html`, Accessed: 16th September, 2019; 2019.

[147] The PHP Group. Directory::read. `http://php.net/manual/en/directory.read.php`, Accessed: 4th July, 2018; 2018.

[148] Caroline Tice. Improving Function Pointer Security for Virtual Method Dispatches. In *GNU Tools Cauldron Workshop*, 2012.

[149] 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.

[150] Tool Interface Standards (TIS). Executable and Linkable Format (ELF). `https://www.cs.cmu.edu/afs/cs/academic/class/15213-s00/doc/elf.pdf`, Accessed: August 7, 2018; 2018.

[151] Trail of Bits. McSema. `https://github.com/trailofbits/mcsema`, Accessed: 16th September, 2019; 2019.

[152] 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.

[153] Alan Mathison Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 1937.

[154] Unicorn. Unicorn - The ultimate CPU emulator. `https://www.unicorn-engine.org`, Accessed: March 21, 2019; 2019.

[155] Victor van der Veen. Trends in Memory Errors. `https://vvdveen.com/memory-errors/`, Accessed: September 16, 2019; 2017.

[156] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[157] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrida. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[158] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[159] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. ReFormat: Automatic Reverse Engineering of Encrypted Messages. In *European Symposium on Research in Computer Security (ESORICS)*, 2009.

[160] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[161] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When Private Keys are Public: Results from the 2008 Debian OpenSSL Vulnerability. In *Internet Measurement Conference (IMC)*, 2009.

[162] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

[163] Michal Zalewski. american fuzzy lop. `http://lcamtuf.coredump.cx/afl/`, Accessed: 13th August, 2019; 2019.

[164] Chao Zhang, Scott A Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, and Dawn Song. VTrust: Regaining Trust on Virtual Calls. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[165] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Protecting Virtual Function Tables' Integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[166] Yuliang Zheng, Josef Pieprzyk, and Jennifer Seberry. HAVAL–a one-way hashing algorithm with variable length of output. In *International Workshop on the Theory and Application of Cryptographic Techniques (AUSCRYPT)*, 1992.