Full-Stack Static Security Analysis for the Java Platform

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von

Dipl.-Inform. Benjamin Hermann

geboren in Langen (Hessen).

Referent:	Prof. Dr.-Ing. Mira Mezini
Korreferent:	Prof. Dr. Awais Rashid
Datum der Einreichung:	29. September 2016
Datum der mündlichen Prüfung:	16. November 2016

Erscheinungsjahr 2016

Darmstädter Dissertationen
D17

Preface

I did not decide right away that in my career as a computer scientist pursuing a doctorate would be something I would invest time into. After my diploma thesis I wanted to venture out and learn through experience. Designing and developing software for a living is something that I enjoyed very much. However, after five years I felt that I needed something more in my life. Coincidentally, a particular problem I encountered during my professional career sparked my scientific interest.

I had the opportunity and the pleasure of exploring this idea – which I present in this thesis – during the last years. During these year I met very bright people that helped me on the way. Following you find an attempt to praise all of them.

First, I would like to thank my supervisor Prof. Mira Mezini, who took the chance of advising someone who has been out of the scientific workflow for some time. It is a venture I recently begun to fully understand and which I am deeply grateful for.

I also thank Prof. Awais Rashid for being the second examiner of my thesis and his time and effort of carefully reviewing it.

Next, I thank Prof. Eric Bodden for all the support during and beyond the projects we collaborated on. I learned an immense amount about the scientific process just by discussing with him.

I also would like to thank the quality assurance inspector of a pharmaceutical company that I had the pleasure of convincing of the quality of a piece software in a prior engagement. During his systematic questioning of our process he let me stumble upon a very interesting issue that I could not solve with state-of-the-art technology. By this, he unknowingly sparked my interest for research. This issue – after many iterations and reformulations – ultimately led me to my research question presented in this thesis.

One of the many things I can look back on is my research project PEAKS that was funded by the German Ministry of Education and Research (BMBF). It was a great honor to receive funding at this early career stage. It enabled me to lead a research team and back my work up with working prototypes, which we released along with the publications. The motivated people involved in this project were two PhD students called Leonid Glanz and Michael Reif, the undergraduate students Lawrence Dean, Tim Kranz, Florian Kübler, Patrick Müller, and Moritz Tiedje, and our brilliant intern Ben Setzer. We grew as a team over the course of the project and still collaborate on new topics.

In this very project, I had the chance to meet Christian Gengenbach from Software AG. He was my mentor over the course of the project and the time after the project ended. I am very grateful for his time and advice during that time. Our regular meetings provided me with strength, confidence and insight to tackle all obstacles on the path to this thesis.

I would like to thank the people that have provided their help in proof reading this thesis. It is their merit, that you can follow my line of thought and, well, might even enjoy reading the next pages. They are Sven Amann, Michael Eichberg, Johannes Lerch, Jan Sinschek, and Michael Reif. Also, I would like to thank the anonymous reviewers of my successful as well as my less successful papers. I took their criticism to heart and tried to include it into my work. To be honest, it is a system that I became to love and loathe at the same time. It is never easy to accept criticism, but once I saw how helpful it was, I immediately became grateful even for the harshest comments.

I would also include the students I had the pleasure to supervise in the last years. These are Bastian Gorholt, Tim Kranz, Dominic Helm, Henriette Röger, Florian Kübler, Florian Wendel and Johann Wiedmeier.

A day in the office would not be complete without the great conversations with my brilliant colleagues Sven Amann, Andi Bejleri, Oliver Bracevac, Marcel Bruch, Ervina Cergani, Joscha Drechsler, Michael Eichberg, Matthias Eichholz, Sebastian Erdweg, Leonid Glanz, Sylvia Grewe, Sven Keidl, Edlira Kuci, Johannes Lerch, Ingo Maier, Ralf Mitschke, Ragnar Mogk, Sarah Nadi, Sebastian Proksch, Guido Salvaneschi, Andreas Sewe, Jan Sinschek Michael Reif, Manuel Weiel, Pascal Weisenburger, and Jurgen van Ham.

No dedication section would be complete without the mentioning of Gudrun Harris. She may be the most important person in the endeavor of venturing out for a doctorate degree in Mira's group. She constantly keeps track that we are all funded properly, fulfill regulations and in general will not lose our nerves in process. I cannot event begin to mention the importance of all of these things in the process of my doctorate studies.

And of course I could not have made it without the support of my partner, my friends and my family. I am deeply grateful for everyone of them, that still talk to me and like me, even though I missed their birthdays, celebrations, or invitations. They forgave me for replying late to their emails and missing important dates. Apparently, pursuing a doctorate degree requires to have a very understanding personal environment and I am glad to say that I have that.

Editorial notice: Throughout this thesis I use the term "we" and "us" to describe my work. It is meant to underline that research is always a cooperative effort and that I would have much less (if something at all) to present here, if other people had not took the time off of their own work to review and discuss mine. I am deeply grateful for their effort.

Also, I use pronouns in the feminine gender in this thesis. This is a deliberate decision to even out the general use of male gender in literature. By using these pronouns I mean to include persons of any gender identity. Written language is, however, not perfectly suited to represent this intent, so please keep the intent in mind when reading this thesis.

Abstract

We have to face a simple, yet, disturbing fact: current computing is inherently insecure. The systems we develop and maintain have outgrown our capacity to prove them secure in every instance. Moreover, we became increasingly dependent on these systems. From small firmware running in cars and household appliances to smart phones and large-scale banking systems, software systems permeate our every day life. We rely on the safety and security of these systems, yet, we encounter threats to these properties every day.

Therefore, systems have be secure by construction and not by maintenance. The principles to achieve this are well known. The Principle of Least Privilege [SS75] has been published in 1975, yet, software systems do not generally apply it. We argue that new, lightweight methods based on sound theory have to be put forward so that developers can efficiently check that their software is secure in their domain context.

In this thesis, we present three analysis techniques that help programmers develop more secure software by informing them about the current state of unsafe operation usage, extensive capability use in third-party components, and suspicious dead software paths that point to programming errors that could lead to insecurity. These three analyses inspect the full stack of a Java program from the application code over library and base-library code down to the native code level. If programmers use the information provided by the analyses, they are able to reduce the attack surface of their applications and provide more safe and secure systems to their users.

Furthermore, we contribute two concepts for automated isolation. While the first concept reduces the attack surface by slicing third-party components to their necessary parts, the second concept is more fundamental and aims at achieving a fine-grained privilege separation.

We believe that the software engineering discipline needs more research on these language-based approaches that tackle the problem of software security at its root cause: defective implementation. Using formal methods to construct these tools is necessary, yet, software developers cannot be overburdened with new requirements to their work process. Automated tools need to derive security properties from program code by themselves with as little input required from the programmer as possible. By these means software can be developed reliably secure in an efficient fashion.

Zusammenfassung

Der stetige Strom an Exploit-Meldungen scheint es zu bestätigen: Aktuelle Software-Systeme sind unsicher. Die Größe der Systeme, die wir entwickeln und warten hat unsere Möglichkeiten überstiegen ihre Sicherheit nachzuweisen. Dazu sind wir sehr viel abhängiger von diesen Systemen geworden, als noch vor zehn Jahren. Software-Systeme durchziehen unser tägliches Leben angefangen von Firmware in Automobilen und Haushaltsgeräten über Smartphones bis zu großen Banktransaktions-Systemen. Wir verlassen uns auf die Sicherheit dieser Systeme, jedoch erfahren wir tagtäglich die Gefahr, die von diesen Systemen ausgeht.

Daher meinen wir, dass Software-Systeme durch Konstruktion sicher sein müssen und nicht durch Wartung. Die dafür notwendigen Prinzipien sind bekannt. Das "Principle of Least Privilege" [SS75] wurde 1975 publiziert, jedoch wird es in aktuellen Systemen immer noch nicht konsequent umgesetzt. Wir vertreten die Ansicht, dass neue, leichtgewichtige Methoden, welche auf tragfähigen theoretischen Ansätzen beruhen, eingesetzt werden müssen, damit Entwickler die Sicherheit der von ihnen entwickelten Software in ihrem Kontext effizient überprüfen können.

In dieser Arbeit präsentieren wir drei Analyse-Techniken, die es Programmierern ermöglichen sichere Software zu entwickeln, in dem sie über den aktuellen Zustand der für das System relevanten Sicherheitkriterien informiert werden. Dies sind im Einzelnen die Nutzung unsicherer Programmiermethoden, die übermäßige Nutzung von Betriebsystem-Ressourcen in Drittanbieter-Komponenten und verdächtige tote Softwarepfade, die auf Programmierfehler hindeuten, welche auch Sicherheitslücken beinhalten können. Diese drei Analysen decken alle Schichten einer Java-Applikation vom Applikationscode über Bibliotheks- und Basis-Bibliothekscode bis zur nativen Codeebene ab. Programmierer sind in der Lage die Angriffsfläche ihrer Applikationen signifikant zu reduzieren, in dem sie die Informationen aus diesen Analysen nutzen, und somit sicherere Software für ihre Endanwender bereit zu stellen.

Weiterhin zeigen wir zwei neuartige Konzepte für automatisierte Isolation vor. Während das erste Konzept die Angriffsfläche einer Anwendung dadurch reduziert, in dem es Drittanbieter-Komponenten auf den tatsächlich genutzten Programmcode reduziert, stellt das zweite Konzept einen fundamentaleren Eingriff dar und hilft dabei eine fein-granulare Berechtigungskontrolle herzustellen.

Wir sind der Meinung, dass Softwareentwicklung weitere Forschung in diesem Bereich programmiersprachlich-basierter Ansätze für Sicherheit benötigt, um die Probleme an ihrem Kern anzugehen: fehlerhafte Implementierung. Während der Einsatz formaler Methoden für die Entwicklung von Werkzeugen hierfür notwendig ist, dürfen Softwareentwickler nicht mit weiteren Anforderungen für die tägliche Arbeit überfordert werden. Automatisierte Werkzeuge müssen Sicherheitsattribute selbstständig aus dem Pro-

grammcode ableiten können und sollten dabei auf möglichst wenig zusätzliche Eingaben des Softwareentwicklers bauen. Durch diese Techniken wird es möglich sein nachweisbar sichere Software in effizienter Weise zu entwickeln.

Contents

1. Introduction

Computer systems permeate our every-day life. They have become an integral part of the fabric of society. From the small mobile device in our pockets over household appliances to traffic management and industrial plants, hardware and software of increasingly distributed computer systems are driving our world. A flaw in the confidentiality, integrity, or availability (CIA) of these systems will not only be a minor disturbance, but will indeed cause a major disruption, when mitigation strategies are missing or fail. Already industrial sites are known to have been compromised and infrastructure, such as hospitals, have fallen prey to extortion by criminals using their computer systems.

Even our economic system has become dependent on the proper operation of computer systems. Stock markets, logistics, manufacturing, and communications would not be able to function without them. Attackers controlling algorithmic trading systems, for instance, can have disastrous effects on the stock market. Especially when considering the interconnectedness of computer systems, they need to live up to high CIA standards, as even industrial control systems (ICS) are frequently connected to the Internet to enable remote access and control. Security flaws can have catastrophic consequences, including the loss of human life.

Moreover, computer systems increasingly rely on software, as it can be developed, tested, and deployed more rapidly than new hardware. Hardware becomes more configurable (e.g., software-defined radio) and more dependent on software running it. This puts a large obligation on the quality of software.

Fortunately, software systems can be protected by a multitude of protection mechanisms. For example, software that interacts with networks can be protected from unauthorized or malicious usage by firewalls, intrusion detection systems, or access control systems. In addition, software can be hardened against possible attacks by inspecting it either statically or dynamically to find security flaws during its development or in a deployed state.

In the following sections, we will discuss the relevant terms for the construction of secure systems. Using these terms, we derive a problem statement and present the contributions of this thesis.

1.1. The Security of a System

A general principle for the construction of secure software is the *Principle of Least Privilege* [SS75]. This principle states that every program in a computer system should operate under the smallest set of privileges that is necessary to fulfill its purpose. While this does not prevent implementation flaws, it minimizes the impact of a corrupted program to these privileges a program needs for normal operation. For example, an administrative program used for filesystem management needs the privilege to read and modify the entire filesystem, but a program that displays some file will just be privileged to read that file.

The security of a system can be characterized using the relationship between three terms: attacks, defenses, and policies [Sch11]. This relationship can be expressed in sentences like "Defenses of type D enforce policies in class P despite attacks from class A". Defenses, policies and attacks can also be combined to express more complex security systems. These relationships have to be treated as hypotheses, which can be proven or disproven.

Policies are the guarantees that a system can still give despite attacks. Conversely, we may state that some attack may break one or more policies. A customer relationship management system, for instance, enforces the policy that customer data is only disclosed to authenticated and authorized users of the system. Thus, the policy of the system expresses a confidentiality property on the customer data. Policies can express properties in either dimension of confidentiality, integrity, and availability.

Attacks are directed against a system's interface. When taking a common Android smartphone as an example, this can be the ability for users to input data via the touchscreen interface, the network connection, or even the temperature of the device [MS13]. The complete set of all attackable interface is called the *attack surface*. The goal of an attack is always the infringement of at least one policy of a system. For instance, the infamous Distributed-Denial-of-Service (DDoS) attacks flooding Internet servers mostly target the availability of a system. However, even though the loss of availability is the most significant effect of such an attack, it might also be intended to break authentication systems to gain access to data and, thus, breaking the confidentiality of the system.

Defenses protect policies on systems from attacks. For instance, a firewall can protect a system from an unauthorized external access, while allowing access from inside the same network.

At the heart of all defenses there are three basic mechanisms: isolation, monitoring, and obfuscation. *Isolation* describes the physical or logical separation of a (sub)system's interfaces from each other or the environment. For example, a virtual machine is somewhat isolated from other virtual machines running on the same physical machine. *Monitoring* describes the observation of a system's interfaces and state in order to enforce policies as soon as a possible violation is detected or predicted. A firewall, for instance, observes the incoming and outgoing traffic of a network and, depending on the traffic's contents and the firewall's ruleset, decides to allow or block network traffic. *Obfuscation* protects data by making it unreadable without the knowledge of a secret. For instance, an encryption algorithm protects data from being understood without the knowledge of a key. Therefore, an arbitrary amount of data can be protected digitally with a small piece of data – the key – which has to be protected physically.

In the design of defenses and policies for systems the Principle of Least Privilege should be the guiding factor. When systems are provided with policies and the defensive mechanisms to enforce them that limit the privilege of the system to the necessary minimum, attacks are less likely to succeed or to be harmful. These defensive mechanisms have to be carefully considered to enforce the policies in an effective way and can use isolation, monitoring, or obfuscation techniques or a combination of either of them.

1.2. Problem Statement

Software development has changed dramatically since 1975 when Saltzer and Schröder postulated the Principle of Least Privilege. Software became less monolithic and consists of various (reusable) parts. Library-based reuse is widely accepted and became an exigence for efficient software production [Boe99, SE13, HDG+11]. But this means developers introduce third-party code into the security context of their applications. We, therefore, have modularity in the construction of software, but no separation of principal[1]. However, we believe that library code must also be limited to its necessary privileges to avoid exploitation. This evolved situation needs to influence the Principle of Least Privilege, which we reformulate as follows (changes in bold typeface):

> Every program, **every part of the program**, and every user of the system should operate using the least set of privileges necessary to complete the job.

Therefore, we need methods to express and enforce policies for program libraries. Application developers have to be made aware of possible vulnerabilities in libraries and the impact they might have, when they want to include the libraries into their programs. Aside from the apprehension of vulnerabilities, developers also need mechanisms to deactivate or remove parts of libraries they deem too risky for their applications.

During the development of software, various techniques from program analysis and rewriting can be used to establish defenses before a program is released into the wild. This class of defenses is called *language-based security* and can also be used for program libraries. Static analysis can be used to infer properties of a program from its static representation including, but not limited to, running time, space requirements, power consumption, and correctness w.r.t. bugs or vulnerabilities. Formal methods, such as information flow analysis, can be used to infer non-interference properties of a program. Compilers and type systems can be used to produce safe low-level code that can be easily verified and safely executed. A large-scale example for the latter is the Java compiler, which produces verifiable Bytecode that can be compiled to assembly code by the runtime engine.

[1]an authenticated entity in a computer system

1.3. Contributions of this Thesis

This thesis makes contributions to the field of *analytical security defenses* applied during development of software. The techniques presented in this thesis make security sensitive areas or security flaws obvious to a developer, who may then take action to remove these flaws or add checks to sensitive areas. Thus, they can be used to monitor the security of a software system in the development process. Other techniques presented in this thesis help developers in reducing the attack surface of a program by removing unnecessary parts or allowing a fine-grained object communication control. Therefore, these techniques provide isolation mechanisms to software developers.

Detection of Unsafe Programming Practices In our first contribution in the field of monitoring, we identify and detect programming practices that can lead to vulnerable code and evaluate our approach on the native part of the Java Class Library (JCL), which is written in C and C++. These languages, unlike languages such as Java, allow for the direct manipulation of memory. Safeguards, such as array-bounds checks, are not in place here. This makes some programming practices unsafe, but also unavoidable. For instance, an array access in C is basically a pointer-arithmetic operation with a pointer to the start of the array and the index as an offset from this pointer to the requested array element. Therefore, user supplied array indices always have to be checked to prevent arbitrary read or write operations on memory areas. A well-known example of a vulnerability involving this programming practice is the Heartbleed bug [DKA$^+$14]. Here, a remote attacker could read the complete memory of a server, opening the door for other attacks with more severe consequences. The cause of this vulnerability was a function that takes a string (which is an array of characters) and a number, which should coincide with the length of the string, and echoes that string back to the client. As the length parameter was not checked for correctness, a client could instruct the function to read beyond the bounds of the string.

A Capability Model for the Java Platform Our second contribution in the field of monitoring is an analysis for system capability usage of Java libraries. Usually programmers use a large set of libraries for various purposes to develop applications. As these libraries run in the same security context as the code developed by the application programmer, they can also use the complete set of privileges given to the application, even though it is not necessary for the operations the library offers. We consider this excessive endowment of permission as a violation to the Principle of Least Privilege. In order to alleviate the situation we contribute a static analysis that detects the use of system capabilities in Java Bytecode. This helps understanding which privileges libraries actually might use and allows a policy system to limit the libraries privileges to the permissions that are actually necessary.

Detection of Bugs through Dead Code Paths Our third contribution in the field of monitoring is an analysis to detect code paths that can be proven to be unreachable.

Assuming that well-written code will never contain such paths, we use this method to point to a large variety of bug patterns, including bugs that can lead to critical security vulnerabilities. We evaluate this method using it on the JCL, the very large basic library shipped with the Java Runtime, and find 258 previously unknown bugs.

Library Slicing Programmers frequently use only parts of the functionality of a library. For instance, the popular log4j logging library offers various logging targets to use. In most scenarios developers only use one of them. Based on this observation, we contribute a method of slicing libraries down to the actually used parts. This way, we further limit the necessary permission set for the respective library and prevent possible abuse of shipped but unused code. Thus, we allow developers to effectively isolate their applications from unused code.

Fine-Grained Capability Control The effective isolation of systems is hard to guarantee in object-oriented systems as authority can be transfered implicitly between components. The Object-Capability Model [MS03, Mil06] provides a set of rules that only allow explicit transfer of authority between objects. Therefore, we inspected capability security more closely. Java's guarantees for memory safety already fulfill the first assumption of the Object-Capability Model that references must be an unforgeable entity. Although Java as a language is quite close to the requirements of the Object-Capability Model, it has three fundamental violations of this principle: Intrusive Reflection, Static References, and Native Bindings. We use a static analysis to find instances of these violations. We performed a preliminary experiment that suggests a strong correlation between these instances and known security flaws. Furthermore, we present a design for a study as future work. Based on a positive outcome of this study, we would be able to enumerate these violations and suggest ideas to remove them allowing developers full object-capability-based control over their complete application. This is a necessary step towards provable and effective, yet flexible isolation.

1.4. Generality of Contributions

In the contributions of this thesis, we use the Java language and platform as the example to evaluate our analyses and concepts. With the exemption of the detection of unsafe programming practices the implementations of the analyses target programs written in Java. However, the concepts behind the analyses can be adapted to other programming languages and platforms.

For example, a capability model such as the one we constructed for Java can be developed for different platforms, e.g. the .NET platform. Many capabilities regarding the underlying operating system will remain the same while capabilities such as class loading or use of debugging facilities will have a different meaning depending on the respective platform. The algorithms will be adaptable since they depend on a call graph for the specific platform API, which can be constructed for most if not all platforms. Our method of library slicing is build on top of the capability model analysis and should, therefore, be directly applicable to the new platform. However, the method of deployment of this new platform needs to be regarded when repackaging the sliced libary. For example, in case of the .NET framework the emission of CIL code and the packaging to assemblies needs to be handled.

Also the detection of dead code paths can be adapted to other programming languages by adapting the underlying control-flow and data-flow analysis to the language specifics. Since it is based on abstract interpretation, the adaption to the new language should be straightfoward to implement.

Our detection of unsafe programming practices is very specific to the practices used in C/C++. Yet, the analysis is not limited to our case study on the native part of the Java platform, but can be directly applied to other codebases that are being used as platforms such as the Linux kernel or the native part of the .NET framework. It is highly likely to find similar results in a case study inspecting these platforms. We use the example of Java in this thesis to provide a coherent, full-stack picture for the Java platform, yet, the method is directly applicable to other platforms.

Our observations on the Object-Capability Model are indeed specific to Java. To retrieve a similar result for a different language the method would have to be repeated. Our findings for Java cannot be generalized.

1.5. Structure of this Thesis

This thesis is organized as follows. In Part I, we present the conceptual foundations of secure software engineering in Chapter 2. Subsequently, we present applications of these concepts, attacks and defenses in Chapter 3 which motivate our work and present open challenges we address in this thesis.

To foster a systematic discussion of the contributions of this thesis, we organize it using the notions of monitoring and isolation. In Part II of this thesis, we introduce three analyses that aid the monitoring of programs under development. First, in Chapter 4, we look into unsafe programming practices, stemming from the fact that low-level languages like C or C++ trade type safety and memory safety for performance. We identify unsafe programming practices and construct ways to detect these practices. Second, we construct a course-grained capability model for system resources in Chapter 5 and develop an analysis to detect their use in library code. Third, we present a detection mechanism for dead paths in programs in Chapter 6. Developers can find complex bugs that may cause vulnerabilities with this analysis. The part closes with a thorough literature review for monitoring technologies for language-based security in Chapter 7

In Part III, we illustrate new ideas to install safeguards in programs to reduce their attack surface. We present the concept for a mechanism for slicing Java libraries to their actually-used parts to limit their permission footprint in an application in Chapter 8. We then go deeper into the rabbit hole of capability-based security and explore the effect of the Object-Capability Model on a more fine-grained level in Chapter 9. We close this part with a review of related work for isolation in the Java Virtual Machine and language-based isolation techniques in general (Chapter 10).

Part IV summarizes this thesis and presents an outlook to areas of further future work.

Part I.

Programming Languages and Security

Hardening Systems During Development

Numerous techniques exist to secure systems in production. Firewalls filter malicious network traffic, transport-layer encryption protects messages on a network from eavesdropping, and intrusion-detection systems alert administrators of suspicious system behavior. These defenses commonly treat the protected system as a black box. On the one hand, this makes these protection mechanisms portable between different systems, but on the other hand this approach dismisses the possibility to protect these systems by removing their vulnerabilities. Systematically removing vulnerabilities from a system during development is called *hardening* of the system.

Programming languages and runtime environments already provide a wide range of defensive techniques to harden systems. In this part of the thesis, we provide an overview of attacks and defenses that are based on programming languages or runtime principles.

In Chapter 2 we inspect the conceptual foundations of secure software engineering. We start with a discussion of the field of language-based security and present the techniques of certifying compilers, defensive execution, and the use of type systems for security. Furthermore, we present techniques directed at information-flow control. The chapter continues with a view on the topic of language-theoretic security. We present the idea of weird machines and the question why protocols are not different from programming languages. A constructive approach to the problem of effective authority assessment is the *Object-Capability Model*. As we assess the security of some of our approaches using this model, we present it in this chapter.

In Chapter 3, we then take a look at programming languages and specific methods of software security. Therefore, we assume the point of view of an attacker and inspect common attacks on programs compiled down to low-level languages. We also present techniques modern operating systems offer to defend against those attacks. When designing systems, security can be build in from the very beginning. We, therefore, present two important principles: *Least Privilege* and *Trusted Computing Base*. We use and extend these principles throughout this thesis. They have also been used in the design of the Java platform. As most of the approaches presented in this thesis are targeting the Java platform, we extend on Java's security model. An integral part of the security of the Java platform is its high-level access control feature. This stack-based access-control mechanism is responsible for guarding critical functionality, such as filesystem or network operations. Other platforms (e.g., .NET CLR) use similar approaches. However, despite the defenses protecting Java applications and the systems running them, the platform has been (and probably still is) the target of specialized attacks. We present some of these attacks that motivated our work. Specifically, we extend on a class of attacks named *Confused-Deputy Attacks*. We illustrate this class of attacks using a specific vulnerability that was present in the Java platform until 2012.

The terminology and techniques presented in this part are helpful to understand the contributions of this thesis. The insights from this study of previous attacks and state-of-the-art defenses motivate our work presented in following parts of the thesis.

2. Conceptual Foundations

In this chapter we present the conceptual foundations for our contributions and provide the relation to our own work. This aids in putting the contributions of this thesis into perspective. We start with techniques to secure low-level code, ranging from compilation techniques over monitoring approaches to type system-based methods. We follow this with a brief introduction to the idea of information flow and a more recent take on a language-theoretic view of software security. We close this chapter with a description of the Object-Capability Model.

2.1. Securing Low-Level Code

In the 1990s computer systems became more interconnected and accompanying this shift software became increasingly mobile, meaning that not only data but also programs are delivered through those new communication channels. An example for this is the applet mechanism of Java: a user downloads a program from a website and executes it immediately after. New questions about the trustworthiness of programs arose, which were addressed with methods from the field of *Language-Based Security* [Koz99, SMH01]. It uses program analysis and rewriting to ensure security policies on programs.

2.1.1. Certifying Compilers

The execution of programs in native code can have a significant impact on the security of a system (cf. Section 3.1), as they come with no guarantees on the safety of executing them. *Certifying Compilers* can help to alleviate this problem and to provide some guarantees on native code to be executed by an end user. In this model, a program in low-level code (e.g., assembly code or Java Bytecode) is accompanied with a certificate of some sort (cf. Figure 2.1). This can be a formal proof about some policy, type annotations, or other forms of certificates. The certificates are created during a compilation step when the program is rewritten from a high-level language (e.g., Java or C++) to a low-level language. These certificates can then be validated against the delivered program code and checked for desired properties. If the desired policies cannot be guaranteed the program execution can be prevented, thus, protecting the system.

The certificate produced by the compiler enables the user of a program to verify the information in the certificate against the program. Hence, the user receives a set of guarantees the program conforms to. Kozen [Koz99] argues that the minimum a program should guarantee are three fundamental safety properties: (i) Control-flow safety, (ii) memory safety, and (iii) stack safety. *Control-flow safety* means a program should only execute a jump or call within its own code segment and only to valid function entry

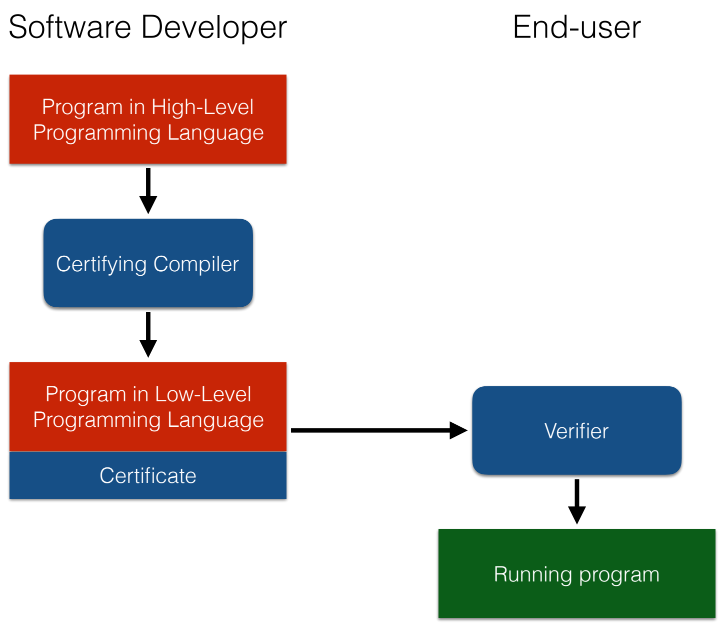

Software Developer
End-user
Program in High-Level Programming Language
Certifying Compiler
Program in Low-Level Programming Language
Certificate
Verifier
Running program

Figure 2.1.: General Workflow of a Certifying Compiler

points. Returns should only happen to the respective call site. *Memory safety* means a program should only read or write within the limits of its own static data segment, heap and valid stack frames. And finally, *stack safety* means a runtime stack should always be preserved across function calls. However, in Kozen's definition minor changes to a runtime stack may be valid as this is necessary for efficiency (e.g., tail-recursion elimination).

Kozen names Java as the first large-scale implementation of a certifying compiler. The Java platform uses an intermediate Bytecode format that includes information that a verifier can use to assert the type correctness, control-flow safety, stack safety, and memory safety of the program to be executed (cf.Section 3.3). It is Java's first line of defense for mobile code.

The concept has been picked up many times. The .NET Framework also compiles to an intermediate format (CIL) that is verified at load time. Several certifying compilers for various other languages and protocols have been proposed [CLN+00, BNR06, ABB+10, NL04, LZC+10].

Proof-Carrying Code The concept of proof-carrying code (PCC) [Nec97] takes this idea one step further. Here, code is delivered with a formal proof of general safety properties. A software provider is responsible for proof generation. A software consumer is then responsible for checking the proof for correctness given the program code. Safety policies are expressed in first-order logic that is augmented with symbols for the respective programming language or machine model.

The protocol (cf. Figure 2.2) consists of a two-phase interaction between the software provider and the software consumer. First, the provider supplies the annotated object code containing pre- and postconditions as well as loop invariants to the consumer. These annotations are produced by a certifying compiler. It uses information gathered during a compilation and program-analysis step to create loop invariants. The pre- and postconditions have to supplied by the programmer. The software consumer then produces a verification condition based on these annotations, which is a logical formula on the safety conditions the program needs to fulfill. The supplier then produces a proof for this formula which the consumer checks for validity with a proof checker.

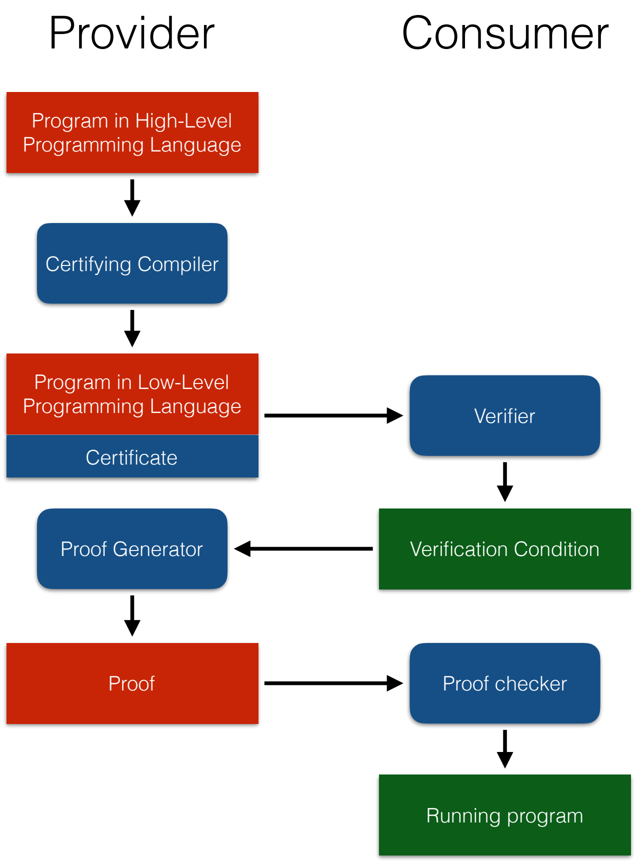


Figure 2.2.: Protocol of Proof-Carrying Code

As the PCC protocol requires a two-way interaction between the software developer and the end user, it is considered unsuitable for the delivery of end-user software. However, the concept is applicable to code that is traversing trust boundaries inside a system. For instance, a Linux kernel allows for code to be executed in kernel mode, which is able

```c
int fib(int n) {
  if (n == 0) return 0;
  else if (n == 1) return 1;
  else return (fib(n-1) + fib(n-2));
}
```

Listing 2.1: The Fibonacci Function Implemented in C

```llvm
define i32 @fib(i32 %n) #0 {
  switch i32 %n, label %2 [
    i32 0, label %8
    i32 1, label %1
  ]

; <label>:1 ; preds = %0
  br label %8

; <label>:2 ; preds = %0
  %3 = add nsw i32 %n, -1
  %4 = tail call i32 @fib(i32 %3)
  %5 = add nsw i32 %n, -2
  %6 = tail call i32 @fib(i32 %5)
  %7 = add nsw i32 %6, %4
  ret i32 %7

; <label>:8 ; preds = %0, %1
  %.0 = phi i32 [ 1, %1 ], [ 0, %0 ]
  ret i32 %.0
}
```

Listing 2.2: The Fibonacci Function Compiled to LLVM IR with Optimization Level 3

to do considerably more harm to a system than a process running in user mode. With proof-carrying code the kernel can verify that code provided by a user only performs the tasks it is allowed to do. Thus, proof-carrying code is an important concept for code that will be executed in an elevated privilege environment.

Typed Assembly Language In the compilation process of typed high-level programming languages type information is mostly removed in the produced low-level code. In typed assembly languages [MWCG99] this information is preserved, transformed, and included in the object code in a low-level language. Proofs for type safety of the program is provided with the program's low-level code. Therefore, code in a typed assembly language was certified by the compiler and can be safely run after a verification step.

The idea behind typed assembly languages is to provide automated facilities to check programs will not violate the rules of their language. For instance, in conventional stack-based instruction sets (e.g., x86) every operand is a fixed bit-length integer (e.g., 16-bit, 32-bit, 64-bit, depending on the register model). All instructions work on these operands, regardless of the underlying type of the operand. With TALx86 Morrisett et al. [MCG$^+$99] provide an implementation of a typed assembly language for the x86 instruction set, that augments instructions with type information in the form of pre- and postconditions. It supports register, stack, and heap type safety. Even higher-order constructs such as arrays, sums, or recursive types are supported.

Java Bytecode is a typed assembly language, but only provides some of these guarantees (cf. Section 3.3). In particular these are register, stack, and heap type safety, but higher-order constructs are missing. Interestingly, these guarantees are enough to provide a memory-safe environment for programs to execute. Bytecode instructions are typed and the Bytecode verifier checks whether all instruction work on correctly typed data.

Another widely-used implementation of the idea of typed assembly languages is LLVM IR [LA04]. Its underlying type system is designed to be language independent and allows primitive types of varying length as well as the derived types pointers, arrays, structures, and functions. Higher-level types such as classes are implemented using these types. We show an example of C code and the compiled bitcode in Listing 2.1 and Listing 2.2. Instructions in the LLVM IR instruction set are overloaded so that they can operate on different types, yet only in the correct combination. LLVM provides register, stack, and heap type safety to the degree that declared type information must not be trusted and has to be checked with an integrated pointer analysis, as weakly-typed high-level languages are supported by LLVM.

2.1.2. Defensive Execution

A different approach to the safe execution of possibly untrusted code is defensive execution. A program is run regardless of its origin and closely observed by a *reference monitor*. This reference monitor has the ability to pause or terminate program execution and to incept any operation of the program with an external component. This gives it the ability to enforce security policies and to secure the system.

References monitors can either be working outside of the address space of the program running in a seperate process or they can be inlined by program rewriting. In the latter case reference monitors become a language-based approach. During compile-time or load-time a program can be rewritten, such that an *inline reference monitor* (IRM) is installed into the program code.

The IRM is then executed as part of the possibly untrusted program. Typically, the IRM is injected at specific operations that form security events regarding the policy to be enforced. It tracks all of these intercepted operations in its own state. This allows for state-based policies, e.g., to disallow operations after an allowed threshold is surpassed. The obvious cost reference monitors entail is the runtime overhead stemming from the event checks that have to be executed even when operations are legitimate.

A popular approach applying runtime monitoring is Control-Flow Integrity (CFI) [ABEL05]. In a first step, CFI computes a static over-approximation of the control-flow graph of the program to be protected. This generates a policy for the runtime monitor, which, in the second step, is responsible to monitor jumps and calls in the program (security events) and intercepts them when they are not included in the pre-computed policy. We present this approach in more detail in Section 3.1.

2.1.3. Type Systems

Type systems have an intrinsic application to security. They ensure that operations are only applied to appropriate values. They effectively enforce memory and control safety as only appropriate memory locations can be used for operations and control can only be transferred to appropriate points in the program. Moreover, type systems of most programming languages allow the programmer to construct high-level abstractions that prevent unauthorized access by means of type-level access control (cf. Section 3.3).

Type information is a valuable asset to reason about the security properties of a program. In TAL, Java Bytecode, and other low-level languages it is preserved to be used in later verification steps. Using type systems for security is a shift from putting the burden of policy enforcement from the end user to the developer. The developer is forced to write a program conforming to the type system automatically, ensuring that it is safe to execute. It is also execution time efficient in the sense that most – if not all – type soundness checks can be made at compile-time or load-time and do not require costly runtime checks.

However, current mainstream type systems have limitations that lead to the need for dynamic value checking. For instance, the type system of C++ does not check for array-index constraints. Schneider et al. [SMH01] argue that this is just a matter of the expressiveness of the underlying logic. They name work on dependent types to eliminate array-bound checks [XP98] as an example for the dormant power in type systems.

2.2. Information-Flow Security

Defensive techniques for software security are distinguished by their protection goal, i.e., what policies they protect from what kind of attacks. The approaches presented so far protect the confidentiality of a part of the system. For instance, access control to sensitive data effectively protects the release of that data. Only authorized subjects may receive this data according to their privilege. However, access control does not protect the proliferation of this data. Once it is released, the source of this sensitive data cannot control any longer how this data is passed on to other, possibly unauthorized subjects. For example, the confidentiality of an encrypted piece of data is guaranteed (assuming the cryptography is unbreakable) only as long as the data is encrypted. Once the receiver decrypts the data, the sender cannot control what the receiver does with it.

Methods from the field of information-flow security [DD77, SM03] aim to protect confidentiality policies in an end-to-end design [SRC84], which means the confidentiality of information is enforced in the system as a whole. Using such an enforcement for the previous example means the piece of data is protected from its creation at the site of the sender to its destruction on the site of the receiver. It can only be used in accordance to its confidentiality policy. Information-flow security is a language-based technique, as it either uses the semantics of the language or the type system to reason about the confidentiality of a piece of information. An extensive inspection of stack-based access control and information-flow security is given by Banerjee and Naumann [BN05]. They also provide a static analysis for secure information flow that checks for the correct use of access control in a program.

Type-system-based approaches use type checking to statically enforce confidentiality policies. Implementations such as JIF [Mye99, MNZZ] augment regular types with annotations for specific security policies about the use of the such typed data. For instance, a variable may be declared as an integer belonging to a principal p that also grants access to principal q. These policies are then enforced during compilation by the type checker. Any use of a variable in a context that does not conform to the security policy does not type check and results in a compilation error. Just as type checking is compositional, checking security types is too. Abstraction mechanisms such as subtyping, polymorphism, dependent, or linear types can be used.

Semantic-based approaches have the ability to reason about non-interference or declassification policies on a program using a formal model of security in the form of program behavior rules. Non-interference means that an attacker cannot learn any information about confidential data from the system. This entails that every information in the system that depends on the sensitive data cannot be inspected by the attacker. As non-interference is usually too restrictive for real-world applications, declassification policies have been devised. They allow for the controlled declassification of information in the four dimensions: what, who, where, and when.

The main advantage of information-flow approaches is that they track implicit flows, meaning that not only explicit assignments are tracked in checking a confidentiality policy, but also any value that depends on sensitive values. However, all approaches need the programmer to augment their program with policy information or other formal specifications which programmers are rarely willing or able to provide. Thus, formal approaches are only applicable in cases where the security requirements justify the extra work.

2.3. Language-Theoretic Security

In previous sections, we reviewed defenses to allow the safe execution of untrusted code and attacks against either these defenses or attacks against the underlying system (e.g., buffer overflows). Yet, there are effective attacks that do not require the transfer of untrusted code to the attacked party. The systems attacked are vulnerable, simply because they receive a message of some kind from the attacker, which is essentially a request to perform a computation on the untrusted input value (e.g., parsing). What looks benign at first glance, turns out to be a severe security issue when inspected more closely.

Sassaman et al. [SPBL13] argue that the increased use of composition in current software development fosters the creation and use of communication protocols between those components as more boundaries have to be crossed. But, as Fred Schneider puts it: *"Since (by definition) interfaces provide the only means for influencing and sensing system execution, interfaces necessarily constitute the sole avenues for conducting attacks against a system."* [Sch11]. Thus, if we increase the number of boundaries (i.e., interfaces) we increase the attack surface of the system.

Intrusion detection systems have been designed with this particular class of attacks in mind. However, to be effective they have to mimic the behavior of the input recognizer code they protect. They may also introduce vulnerabilities that were not there before, because the code of the detection system can be faulty. Despite much effort, these systems do not fundamentally reduce the insecurity of the system they protect. Sassaman et al. found that the underlying problem is the computational power of the input language of the interface. If the input language itself is undecidable, there will be no feasible way to produce a recognizer for this language which will fulfill safety (nothing bad happens), liveness (something good will eventually happen) [Lam77, AS87], or termination (it will eventually terminate) properties.

Therefore, it is impossible to show the equivalence of the encoding and the decoding of the language between the components. As a simple example take the interface between the Java Class Library and its native counterpart for operating-system binding (cf. Section 3.3 and Section 4.3). These two components frequently exchange memory pointers. Since Java does not have an explicit type for pointers, the native code converts it to an integer value and passes it to the Java side. As it is not explicitly typed on the Java side, the integer value can be freely modified and then returned to the native code, which converts it back to a pointer. However, this pointer may now target a different possibly unrelated section of memory, which might lead to unexpected – if not harmful – results when the native code is executed using this pointer value. The two parties did not agree on a reliable protocol and, therefore, the Java component did not enforce the policies on the pointer value. To make pointer handling explicit and enforceable, other language, such as VB.NET or C#, feature specific pointer types (e.g., `IntPtr`).

Bratus et al. [BLP+11] use the term *weird machines* for the computational concept behind these kind of attacks, because they demonstrate hidden, unexpected computational power in the existing code base. This hidden power can be harnessed by attackers using crafted input values – the exploit code. In this exploit code, they argue, lies a constructive proof that computations that are not intended by the original developer can still be achieved. Thus, exploits provide evidence of incorrect assumptions on the powerfulness of the program. In terms of formal language theory, they demonstrate a computationally stronger automaton than the one intended.

Of course, languages of the inputs exchanged between components can be more complex than this easy example. Sassaman et al. argue that all programs processing input languages should be decidable. Input languages as they understand are file formats, wire formats, encodings, scripting languages, finite-state concurrent systems such as network protocols, and of course mobile code in low-level formats. A language in the class of recursively-enumerable languages is a security risk, as its parser is not guaranteed to terminate and could perform arbitrary computations on behalf of the attacker.

More complex examples are injection attacks such as SQL injections or buffer overflows. While in the former example, a parser fails to interpret an argument and provides a new command to the SQL database, in the latter example, an attacker uses an unchecked access to a bounded region of memory (e.g., an array or a buffer) to read or write beyond the bounds of that region. Sassaman et al. show that even complex protocols, such as PKCS#1, can be inspected in terms of language expressiveness and decidability.

2.4. The Object-Capability Model

Many programming languages and their underlying security models allow for authority to be transferred implicitly (e.g. Stack-Based Access Control, cf. Section 3.3). This enables attack schemes such as the Confused Deputy (cf. Section 3.4). A model that enforces explicit, and therefore traceable, authority transfer is the *Object-Capability Model* [MS03, Mil06]. It provides fine-grained privilege separation based on the insight that in an object-oriented system the only way to execute an operation on (or to retrieve data from) an object is to have a reference to that object in order to send a message to it. A capability, in this sense, embodies a transferable permission to perform operations on an object, which puts a strong emphasis on a reference and implies some requirements.

The first and most important requirement states that references must be *unforgeable*. That means that there is no way to obtain a reference to an object other than constructing the object the reference points to or receiving it through a method (or constructor) parameter. The Java language fulfills this requirement, but in C++ a programmer can construct references to arbitrary objects on the heap by constructing memory pointers to the objects position. When this requirement is not fulfilled, no assumptions on accessible objects can be made anymore, as any object can be reached at any point in the program.

As a second requirement, an object system needs to ensure private state encapsulation. Without this, a reference to an object always makes all references this object holds accessible, therefore, breaking isolation. Also, no shared mutable state may be used. Objects can collude and share references over such a channel and, hence, tracking references becomes significantly harder. As a last requirement, devices like the network socket or the filesystem must only be accessible through objects, otherwise they cannot be protected.

In order to obtain authority only these four mechanisms may be used:

Initial Condition In the initial condition of the program inspected, an object A already has a reference to object B. For example, the configuration of a program environment using dependency injection (e.g., Spring) already sets this reference and the environment initiates the program with that reference already in place.

Parenthood An object A creates a new object B and, thus, receives the only reference to the new object B.

Endowment An object A creates a new object B and provides it a subset of its references. Object A receives the only reference of the new object B. Object B now has the set of references A has provided.

Introduction An object A already has a reference to object B and C. Object A can now send a message to object B containing a reference to object C. After receiving this message object B now has a reference to object C.

When the requirements of the model are met and the four transfer rules are followed, the reference graph of the objects on the heap is exactly the access graph of the program. This allows to safely reason over the security of the system by means of this access graph.

The idea of capability-based security systems is indeed older. Dennis and van Horn introduced the idea of capabilities [DVH66] for the design of computer systems. Their notion of capability is a data structure that provides a guarded access to a computing resource. A capability has a concept of ownership and can be granted and ungranted by means of the programming language of the system. The idea has been adopted in system design, although not into current mainstream operating systems. For instance, the KeyKOS microkernel [Har85] implements the idea for IBM's System/370 and UNIX. Newer implementations are POSIX Capabilities [Bac02] and Capsicum [WALK10], both based on the idea to split monolithic UNIX processes in smaller parts with limited authority.

The first work transferring the idea from system design to language design was the E programming language[1] by Miller, Borstein, and others. It was designed for the robust composition of distributed systems and implemented object capabilities. In their J-Kernel system van Eicken et al. [vECC+99] apply the concept to Java and provide capability-based protection domains. Interestingly, their work was developed in parallel to the Version 1.2 redesign of Java's security model, but allows a more fine-grained separation.

Several language subsets have been proposed to derive capability-safe languages from common programming languages. The following list is not nearly complete but representative. Oz-E [SVR05] is a capability-safe variant of Oz [Smo95], a multi-paradigm research language. Joe-E [MWC10, MW10] is a subset of Java. In Joe-E not only the language was limited to conform to the Object-Capability Model, but also the class library was cut down to a capability-safe part. Miller et al. provide Caja [MSL+08], which is a capability-safe subset of JavaScript with the clear focus on composed applications in web browsers.

[1]`http://erights.org/`

3. Programming Languages from a Security Perspective

In this chapter, we present a security view on programming languages and runtime platforms with a particular focus on Java. We discuss attacks that use specific vulnerabilities of programs written in or compiled to low-level languages. We then take a step back and review two fundamental security principles: The *Principle of Least Privilege* and the *Trusted Computing Base*. These are the foundations that defensive mechanisms for security are build upon. Current programming languages and runtime environments already have various of these mechanisms for the safety and security of programs in place. We discuss them using the example of the Java platform.

3.1. Attacks on Programs: The Perils of Compiled Code

In memory, all things are created equal. Access to memory does not distinguish between data or program, between executable or argument. This enables many features we take for granted in modern computing, however, it also enables attackers to inject operations into programs. Therefore, most current operating systems implement some form of executable space protection, which marks parts of the memory as not writable or not executable. In the most common implementations the heap and stack memory areas of a program are protected by a not-executable (NX) flag. This hinders certain buffer-overflow attacks from succeeding to inject and execute new instructions.

A buffer overflow can be used for an attack known as *stack smashing* [Lev96]. The attacker uses a pointer to data on the stack to overwrite the return address of the function currently on the stack. Attackers can then change the control flow of the program and use the privileges of the program to gain access to a computer system. As an example, take the code in Listing 3.1. The `doStuff` function uses the `strcpy` function in line 9. This function, however, does not perform a bounds check on the arguments and allows to write beyond the allocated memory of the character array `c`. When inspecting the stack layout presented in Figure 3.1, we can see that crossing the allocation bounds of variable `c` can overwrite the return address. An attacker can jump to any point in the memory address space and execute instructions, if there is such an accessible return address.

Execution prevention techniques for memory effectively defend systems against conventional stack buffer overflow attacks and protect the control-flow integrity policy against jumps outside the program's memory. However, the control flow inside the program memory can still be hijacked by an attacker. In this category of attacks, called *Return-Oriented Programming* (ROP), chosen machine-instruction sequences called *gadgets* are chained together through return-address manipulation. Leveraging a bug in the

```c
#include <string.h>

int main (int argc, char **argv) {
  doStuff(argv[1]);
}

void doStuff(char *arg) {
  char c[10];
  strcpy(c, arg);
}
```

Listing 3.1: A Program Vulnerable to a Stack Buffer Overflow

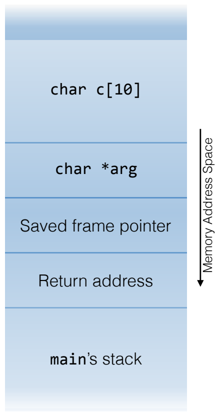


Figure 3.1.: Example Layout of a Stack for Function `doStuff` from Listing 3.1

program (typically an unchecked buffer), the return address is overwritten with the address of the next point in the control flow the attacker desires. If an attacker finds a sufficient set of these gadgets (typically a read, a write, and a loop gadget), she can execute arbitrary operations and, thus, take control of the program. Gadgets are usually found in shared library code (e.g., libc [Ner01]). As all such library code is located inside the memory area of the program and can be executed eventually as part of normal program flow, techniques for execution prevention do not limit the possibilities of ROP attacks.

An effective defense integrated into modern operating systems is *Address Space Layout Randomization* (ASLR). In this defensive mechanism, the address space positions of data areas, libraries, the heap and the stack are randomized for each process. Therefore, it is harder for an attacker to predict, where the gadgets necessary for a ROP attack are located.

A more recent mitigation attempt was described by Vasilis Pappas [Pap12]. His kBouncer technique was motivated by the fact that ASLR is often ineffective in practice, because of fixed parts of the address space, incompatible libraries, or computeable base addresses of libraries. It leverages the *Last Branch Record* feature included in Intel processors to check for problematic return instructions and intervenes, if necessary. However, Carlini and Wagner [CW14] as well as Schuster et al. [STP+14] have shown how ROP attacks can be masked in such a way that the most popular defenses kBouncer, ROPecker [CZY+14], and ROPGuard [Fra12] do not detect them.

Another strategy to prevent return-oriented programming is *Control-Flow Integrity (CFI)* [ABEL05]. The idea is to compute a static control-flow graph for a program and to have a runtime monitor prevent executions of any call or branch that is not included in this graph. It has long been considered as a strong defense mechanism against ROP and other control-flow attacks. However, recently Carlini et al. [CBP$^+$15] have shown that even this defense can be circumvented with reasonable effort. Their attack technique named *Control-Flow Bending* leverages valid jump targets allowed by the CFI implementation (e.g. using a modified function pointer) to achieve the attack goal. They show that this technique is effective even against fully-precise static CFI.

3.2. Guiding Security Principles for Secure Software Engineering

Besides adding defensive mechanisms to operating systems or runtime engines, it is also desirable to construct programs free of vulnerabilities. In the pursuit of developing secure software, principles can help to guide efforts towards this goal. We present two such principles, which are relevant for this thesis: The *Principle of Least Privilege* and the *Trusted Computing Base.*

The **Principle of Least Privilege** [SS75] states that every program and every user should operate using the least set of privileges necessary to complete the job. Also, if the system is connected to other systems, the amount and volume of interactions between these systems should also be limited to the necessary minimum. Following this principle limits the damage a program can do, when privilege is misused by an attacker that gained control. This has two consequences: (a) Privilege is less likely to be abused, because it is more scarce and (b) in case of a privilege abuse, the number of programs to review is reduced to those that need the privilege to function. Both consequences are strong arguments to follow the principle as it makes exploitation less rewarding and it guides resources for program review towards critical programs.

However, software development has changed considerably since the principle was defined in 1975. Programs are not monolithic blocks of code anymore. Library reuse has become a necessity in the efficient development of software [Boe99]. This entails that programs are delivered with a significant amount of third-party code. To ensure that this code conforms to the security requirements of the program, it is therefore necessary to review the code of all third-party libraries before including them into the program. This is required because this code is running in the same security context – the principal – as the rest of the program and thus possesses all the privileges of the program.

The Principle of Least Privilege permits library reuse, as this third-party code is a part of the delivered program and necessary for its functionality. Nevertheless, from a security standpoint this is not a satisfactory solution: We have modularity in code, such that different code providers can influence a final program, yet, we do not have modularity or separation of principality. This observations leads us to a reformulation of the Principle of Least Privilege.

> Every program, **every part of the program**, and every user of the system should operate using the least set of privileges necessary to complete the job.

This reformulation immediately leads to the need for mechanisms to determine and limit the set of privileges available to parts (e.g., libraries, components) of a software systems. We will present such mechanisms in this thesis.

The notion of **Trusted Computing Base** (TCB) [Rus81] describes a set of components of a system that are implicitly trusted. Usually, this is the hardware, firmware, (parts of) the operating system, and a compiler. As these components are implicitly trusted, a vulnerability in these components jeopardizes the security of the entire system. However, components outside the TCB can be effectively monitored and isolated by components in the TCB. As it is paramount for the overall security of the system, designers of TCB components attempt to keep its size as small as possible. This is also useful when formally verifying its security, as less code has to be checked, thus, making the proof process easier.

In a Linux system, the TCB consists of the hardware (including firmware), parts of the kernel, and the C compiler. The security of every program compiled with this compiler directly depends on the compiler's security. When we verify the kernel parts and the compiler and trust the hardware, we can, therefore, conclude that the system's foundation is secure and programs running on this setup thus can be secure as well. If we fail to show this, no program running on this setup can be considered secure, regardless of the programs individual properties.

3.3. The Java Security Model

The Java platform originated in 1991 and had its first public release in 1995. The design goals from the very beginning were to provide an easy programming language, operating-system independence, safety guarantees, and a sophisticated access-control model. The latter was motivated by the requirement to support so-called *mobile code*. This concept describes the ability of a program to be installed on-the-fly over a network and its ability to load components during runtime. From the beginning it was clear to the designers of Java, that this feature of the platform raised the need for an access-control mechanism to prevent abuse [Gon98, Gon11].

Java programs and libraries are not compiled to actual machine code immediately. The Java compiler emits code in an intermediate language – Java Bytecode (cf. Listing 3.2). Programs and libraries are delivered to the end user in this format. The Java Runtime Environment (JRE) then performs a Just-In-Time (JIT) compilation step to run it on the end user's operating system and hardware.

The first layer of defense for mobile code is Bytecode verification [Ler01, Yel95, Gos95]. When loading a class, the Java runtime first checks the Bytecode for various properties. The specific guarantees provided by these checks are:

- **Type correctness**, i.e., all arguments of an instruction are always of the types expected by the instruction. Java is a strongly-typed language and the benefits of this approach are preserved in Java Bytecode as well.

- **Balanced stack operations**, i.e., the stack can never be less than an empty stack and never be larger than the maximal stack size. Instructions will not be executed, if they violate this rule and, thus, cannot break into adjacent parts of the memory.

```
public class Test {
  public int multiply(int first, int second) {
    int val = first * second;
    return val;
  }
}

public class Test {
  public Test();
    Code:
       0: aload_0
       1: invokespecial #1 // Method java/lang/Object."<init>":()V
       4: return

  public int multiply(int, int);
    Code:
       0: iload_1
       1: iload_2
       2: imul
       3: istore_3
       4: iload_3
       5: ireturn
}
```

Listing 3.2: Example of Java Code and the Resulting Compiled Bytecode

- **Code containment**, i.e., program counters can only point to parts inside the same method and only the starting part of an instruction. Branches can, therefore, never jump to any other position than the start of the instruction they want to jump to.
- **Register and object initialization**, i.e., a `load` operation must always be preceded by at least one `store` operation. If an object is created, the initialization methods must always be invoked before the instance can be used.
- **Access control**, i.e., all method invocations, field accesses, and class references must honor the visibility modifiers of the respective construct.

Although all of these properties can be checked at runtime, these checks are performed at load time to prevent a slow-down of the program. Bytecode that has been verified once can be executed safely.

3.3.1. Type Correctness

The Bytecode verifier performs a type-level abstract interpretation [CC77] of the provided Bytecode. All instructions in Java Bytecode are typed already. Types processed by the abstract interpretation include the primitive types (`int`, `long`, `float`, and `double` – `boolean`, `byte`, `short`, and `char` are represented by `int` in Bytecode), object types (identified by their names), array types, and `null`. There is an additional element to describe uninitialized registers (local variables).

This is an important distinction of the Java Bytecode instruction set to other stack-based instruction sets (e.g., x86), where the instructions are free to operate on whatever arguments are on top of the stack. In Java, this verification step ensures that the operands on the stack and in local variables are of the correct type. This is also enforced for branches that meet on the same point.

During the abstract interpretation all Bytecode instructions of each method are executed, but operate over types and not concrete values. For instance, the interpreter executes an `iadd` operation only if two integer-typed values are on the top of the stack. Method invocations are treated as if they were well-typed, given that if the opposite was true, the verification of the called method would fail. Verification fails when the interpreter gets stuck, i.e., fails to find a valid transition in the operational semantics of the Bytecode language. This verification step guarantees the type correctness of the Bytecode blocks and prevents stack over- and underflows.

3.3.2. Low-Level Access Control

Classes, methods, and fields in Java can be declared with the visibility modifiers `public`, `protected`, and `private`. If this modifier is missing, a fourth visibility level (`default`) is assumed. Elements with a `private` modifier can only be used in the scope of the enclosing class. Elements without a modifier can only be used in the scope of the same package as the enclosing class. Elements with a `protected` modifier can only be used in the scope of the same package as the enclosing class or a subclass of this class. Elements with the `public` modifier can be used from everywhere. The Bytecode verifier checks whether these conditions are fulfilled and rejects Bytecode that does not conform to them. In a previous step, the Java compiler also checks for these conditions and does not compile Java programs if the check fails. However, as Bytecode might also be created without the use of the Java compiler, the verification step is necessary to enforce access control.

Java's access control implements the idea of information hiding [Par72], which fosters the limitation of critical data to the necessary scope and, therefore, implements the Principle of Least Privilege at the language level. For instance, an object that contains a piece of sensitive data effectively controls the access to this information by storing it in a private field. Each method of the object now has the ability and the obligation to check the permissions of a caller, before returning the data.

However, with reflection, access control can easily be circumvented. A principal with the permission to access the reflection capabilities of Java can view, manipulate, and delete the complete internal state of every object it has a reference to. This breaks any assumption on the protection of sensitive data or operations the object owns. Reflection can also be used in accordance to information hiding, so to distinguish the different uses we call reflection that breaks information hiding *Intrusive Reflection*.

3.3.3. Memory Safety

In contrast to languages like C or C++, Java does not allow direct manipulations of memory. The Java runtime is responsible for allocating and deallocating memory for the program. Also, it guarantees the proper initialization of value-type and reference-type variables. This eliminates the security problems with direct pointer access described earlier (cf. Section 3.1) and provides a safe environment to run code from sources of questionable trust.

Arrays and strings in Java are always bounds-checked. A write beyond the allocated memory is, therefore, impossible. As each variable is initialized by the runtime engine, a read from uninitialized memory can never happen. Also, as Java uses garbage collection, it is not possible to have a reference to an object that has already been deallocated from memory.

However, even though the language forbids direct memory manipulations, they are possible from Java code using the Java API, specifically with methods from `sun.misc.-Unsafe`. As Mastrangelo et al. [MPM$^+$15] have shown, the use of this API is rather prevalent in current Java code. Of course, the use of these methods invalidates any memory safety guarantees the platform can give. The particularly critical part here is that the use of `sun.misc.Unsafe` often happens unbeknownst to the application developer as part of libraries. Therefore, it is of paramount importance to make this visible to the application developer when including libraries to her application.

Direct memory operations from the Java side can have the same catastrophic consequences as they have from C/C++ code. The virtual machine can be compromised or crashed. The heap of a Java program cannot be trusted anymore, if such operations are in use.

Technically, this is not only true for uses of `sun.misc.Unsafe`, but every native function that can be reached from the Java side. Any vulnerability in the native part of the Java platform jeopardizes any guarantees given on higher layers. For instance, image processing functions in the native part of the JCL were prone to buffer overflow attacks triggered by prepared images passed from Java code (e.g., in CVE-2015-4760[1]). Therefore, any value passed to a native function in Java has to be treated as possibly harmful.

3.3.4. High-Level Access Control

A verified program in Java Bytecode can be safely run on a Java Virtual Machine w.r.t. the guarantees discussed before. However, these guarantees do not protect operating system resources such as the filesystem of the machine. For this purpose Java uses a sophisticated access control system. The platform is provided with a standard set of policies suited for the protection from harmful mobile code (e.g., applets). Developers can add policies and custom permissions to the access control system.

```
grant [SignedBy "signer_names"] [, CodeBase "URL"] {
  permission permission_class_name ["target_name"] [, "action"] [, SignedBy "signer_names"];
  permission ...
}
```

Listing 3.3: Syntax Example for a Java Policy File

Using a policy file (cf. Listing 3.3) specific parts of a Java program can be entrusted with a number of permissions. These parts are identified either by their origin

[1] `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-4760`

(`CodeBase`), or a digital signature (`SignedBy`), or both. Permissions can have arguments to limit the permission. For instance, `java.io.FilePermission` has two arguments: The first argument describes a file pattern for which the permission should be granted. The second argument describes whether the permission is only granted for reading or for reading and writing to files and directories matching this pattern.

Writing these policy files is a complicated task and they are hard to test. We performed a search on Java projects hosted on GitHub and found that less than 0.02% that were providing policy files at all. This implies the policy feature of Java is used very little in practice. Koved et al. [KPK02] contributed a technique to automatically infer a least privilege policy for Java components. They provide a data-flow analysis that tracks the use of `Permission` instances in order to determine the smallest set of permissions necessary to run a piece of code. However, as the components have to be maintained, so do these policy files, which puts an additional task on the list for software developers. This task cannot be fully automated as the analysis can only approximate a policy and will, e.g., miss dynamically created permission requests.

The enforcement of these policies is a runtime technique called *Stack-Based Access Control*. In this model, sensitive operations are protected with explicit calls to a check procedure that inspects the permission set of each frame on the current call stack according to some pre-defined policy. If at least one frame of the stack does not possess the requested permission, the operation fails. This access control scheme is not specific to Java. It is used in other platforms such as the .NET Framework (Code Access Security) as well. Yet, throughout this chapter, we use the Java security model to illustrate the concept.

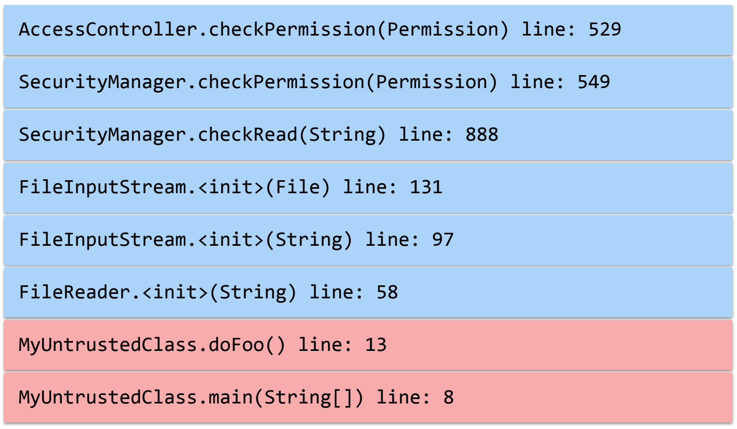
Permissions in Java are expressed as subclasses of `java.security.Permission`. Several permission representations are already provided in the standard class library and specific representations can be added by subclassing. Permissions can be specialized by providing values in their constructor arguments. For example, instances of `java.-security.FilePermission` can be specialized with a path or file pattern and an action (i.e. read, write, execute, delete, and readlink). Permission instances are used in policy files and in permission requests.

Critical resources or procedures have to be actively protected by calls to a check function. As Java's security model was changed significantly in version 1.2 of the platform, multiple of such functions exist. However, the only pattern that is backward compatible is presented in Listing 3.4. A call to `System.getSecurityManager()` retrieves the application's instance of the `SecurityManager` class, which is only set in a restricted security environment. This is the case when executing an applet in a browser or when setting the security manager instance directly upon starting the Java VM. In cases where there is no policy enforcement configured, the return value of `System.getSecurityManager()` is `null`.

When policy enforcement is enabled, a call to the `checkPermission` method starts the evaluation process. Consider the following example: An instance of class `MyUntrustedClass`, which as the name suggest is not trusted in this scenario, tries to read a file through an instance of the `FileReader` class. For clarity of presentation, we skip technical details in the description of this example, but present a full call stack in Figure 3.2.

```java
SecurityManager security = System.getSecurityManager();
if (security != null) {
    FilePermission perm = new FilePermission("somepathtofile", "read");
    security.checkPermission(perm);
}
```

Listing 3.4: Checking a Specific File Permission



```
AccessController.checkPermission(Permission) line: 529
SecurityManager.checkPermission(Permission) line: 549
SecurityManager.checkRead(String) line: 888
FileInputStream.<init>(File) line: 131
FileInputStream.<init>(String) line: 97
FileReader.<init>(String) line: 58
MyUntrustedClass.doFoo() line: 13
MyUntrustedClass.main(String[]) line: 8
```

Figure 3.2.: Layout of a Call Stack for an Example File Operation

`FileReader` delegates this to an instance of the `FileInputStream` class, which starts the permission evaluation process by calling `checkPermission` on the current instance of `SecurityManager`. This method retrieves the current call stack of the application and checks the policy of each stack frame against the permission requested, in this case the `FilePermission` for this particular file. If this permission was granted to each stack frame, the `checkPermission` method just returns to its caller and the operation is performed. If one of the stack frames does not possess this permission, `checkPermission` throws a `SecurityException`. This breaks the normal control flow and prevents the critical operation (i.e., the file access).

When a Java applet calls a method to write a file to the client's hard-drive, the check procedure walks down on the stack, finds a stack frame from the applet's code, and – according to the default policy – denies the operation. Yet, there are cases where suppliers of trusted code such as the JCL want to enable unprivileged clients to perform certain privileged operations. For instance, an applet might be allowed to invoke operations which need to read a specific configuration file from a client's hard drive. Using the model presented so far, the applet's methods would be on the stack at the time of the permission check and access to the file would be denied consequently.

The `AccessController` class offers a facility to perform such operations. Defined actions can be performed using the `doPrivileged` method. This action will run with all privileges of the caller (cf. Listing 3.5). Permission checks will end at the stack frame containing the `doPrivileged` call and, therefore, only check stack frames that have

```java
doSomething() throws FileNotFoundException {
   // code in normal privilege level

   try {
    FileInputStream specialFile = (FileInputStream) AccessController.doPrivileged(new
         PrivilegedAction() {
        public Object run() throws FileNotFoundException {
           // code running in elevated privilege level
           // example:
           return new FileInputStream("someFile");
        }
      });
    } catch (PrivilegedActionException e) {
      // unwrap exception and rethrow
      throw (FileNotFoundException) e.getException();
    }

   // code in normal privilege level
}
```

Listing 3.5: Elevating Privileges

been added with the privileged action. Callers of `doPrivileged` have the obligation to carefully check the resources and operations they provide to unprivileged code, as the sensitive function wrapped inside a `doPrivileged` call cannot check the full stack anymore. Therefore, particular care has to be taken not to insert new vulnerabilities.

Alternatives to stack-based access control have been discussed extensively, because of its limitation on information contained on the stack to express policies. Abadi et al. [AF03] suggest to base access control on execution history rather than on the current state of the stack. Their model does not only include method nesting as in stack-based access control, but also includes any method call that has been executed prior to the method that starts the check. Stack-based access control has no knowledge of these methods and their changes to the global state of the application. If this global state influences security-critical methods, stack-based access control will not prevent their execution. Martinelli et al. [MM07] integrated history-based access control into Java using IBM's Jikes VM. However, history-based access control comes with a significant overhead, as the method systematically needs to perform more costly checks.

An alternative called *security-passing style* was introduced by Wallach et al. [WAF00]. Here, security contexts are represented as pushdown automata, where calling a method is represented by a push operation and returning is represented by a pop operation. However, as these automata need to be weaved in with the program, their model requires program rewriting.

Pistoia et al. [PBN07] introduce information-based access control, which is based on stack-based and history-based access control. They argue that history-based access control may prevent authorized code from executing because of less authorized code executed previously, although it may not have influenced the security of the operation that is about to be executed. In information-based access control every access-control policy implies an information-flow policy. It augments stack inspections with the tracking

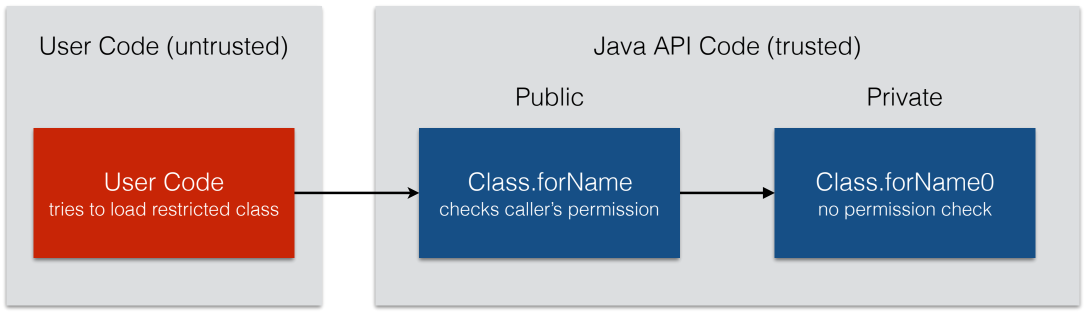


Figure 3.3.: Untrusted Caller Loading a Class in Java 6

of information flow to sensitive operations. An extensive review on the relation of access control and secure information flow is given by Banerjee and Naumann [BN05]. We discuss work on information flow security in Section 2.2.

3.4. The Confused Deputy

Stack-based access control puts a large obligation on the developer providing security-critical resources to correctly implement the necessary checks. For instance, if a filesystem access would not be properly guarded by a call to `SecurityManager.checkPermission` it would be open to every caller regardless of its actual permission. Moreover, using the `doPrivileged` facility, programmers vouch that the operation they perform is permissible for every caller.

Attackers can exploit missing or faulty permission checks or elevate their privileges with unprotected `doPrivileged` blocks. These accidental vulnerabilities of a system are called *Confused Deputies* [Har88], because the vulnerable but trusted code is used to execute operations on behalf of the unprivileged code. To the access-control facilities everything seems to be in order, but unprivileged code can freely perform sensitive actions, including deactivating access control altogether.

Multiple vulnerabilities[2] like these have been exploited in the Java platform, especially between late 2012 and early 2013. Most of these vulnerabilities exploit a flaw in the implementation of access control in Java in so-called *caller-sensitive methods* [CGK15]. These methods implement shortcuts in the access-control check and usually only check the permission of the immediate caller.[3] Since Java 8, these methods are marked with the `@CallerSensitive` annotation [RTC].

We illustrate this class of attacks with the exploit for CVE-2012-4681.[4] Java offers the ability to load classes at runtime and executing them using dynamic invocation with information from reflective facilities in the platform. This feature of the platform can

[2]e.g., CVE-2012-4681, CVE-2013-0422, CVE-2013-2460
[3]Some methods check specific stack frames, but also do not perform full stack walks.
[4]`https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-4681`

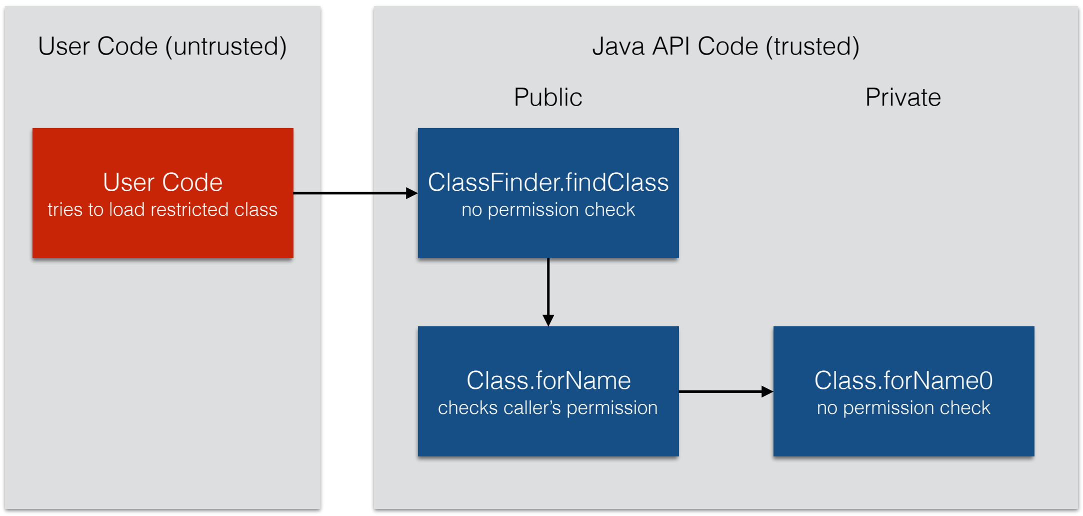


Figure 3.4.: Untrusted Caller Loading a Class in Java 7

be used to implement plug-in mechanisms or other forms of extensibility. Up to Java 6, untrusted code could call the `forName` method of `java.lang.Class` to load the desired class (cf. Figure 3.3). The `forName` method checks the authority of its caller to load the class and either returns the `Class` instance or denies the operation.

Java 7 introduced a new class `com.sun.beans.finder.ClassFinder` (cf. Listing 3.6) that is utilized to retrieve class instances. In order to retrieve the class by the name given in the parameter `className`, it calls `Class.forName` using the class loader of the current context. This performs the class loading operation in the security context of the caller. However, when this operation fails and throws a `SecurityException` in case the caller is not authorized to load this call, `findClass` calls `Class.forName` again using its own class loader by not providing one explicitly. As `ClassFinder` is inside the trust boundary of the JCL it is permitted to load any class. This implementation allowed every caller to load any class. It has been secured by an additional check since Java 7 update 10.

Using the open class loader, the exploit was able to get a reference to a class called `sun.awt.SunToolkit` that contained another vulnerability, which made it possible to deactivate visibility access control for fields. This, in turn, allowed the exploit to deactivate the `SecurityManager` altogether, allowing execution of arbitrary code in an applet context. The exploit was actively used to deliver other malware ranging from botnet nodes to remote control features.

However, it should have not been possible for the attacker to gain access to `sun.awt.-SunToolkit`, as it resides inside a so-called restricted package. Classes inside restricted packages cannot be used or loaded from a restricted security context such as an applet. The open class loader vulnerability inside `ClassFinder` enabled to load this class using `ClassFinder`'s security context and is, therefore, considered a confused deputy. The vulnerability of the unguarded call to `Class.forName` stems from the fact that

```java
public static Class<?> findClass(String className) {
  try {
      ClassLoader localClassLoader = Thread.currentThread().getContextClassLoader();
      if (localClassLoader == null)
        localClassLoader = ClassLoader.getSystemClassLoader();
      if (localClassLoader != null)
        return Class.forName(className, false, localClassLoader);
    }
    catch (ClassNotFoundException localClassNotFoundException) {}
    catch (SecurityException localSecurityException) {}
    return Class.forName(className);
}
```

Listing 3.6: Previous Implementation of `ClassFinder.findClass` (slightly shortened)

it is a caller-sensitive method and, as such, requires the caller to check the necessary permissions of the request, which `ClassFinder` failed to implement.

3.5. Open Issues Addressed in This Thesis

As seen in Section 3.1 attacks which alter the control flow of programs in favor of an attacker's goal are still common despite various operating-system-level defenses. While there are solutions to detect and remove some of these vulnerabilities from program code, developers still need to inspect the code manually for vulnerabilities these tools cannot detect. Especially in larger codebases such as the Java Class Library with over 640k lines of C/C++ code or the Linux Kernel with over 15M lines of code, developers need tools to guide them to critical areas where vulnerabilities are more likely to occur.

In case of Java, vulnerabilities in the native part can be triggered by code as part of an application or Applet that is considered to be safe to be executed from a Java point of view. This means that the guarantees of the Java platform such as access control and memory safety cannot be upheld anymore after an attacker triggered a vulnerability in a native function which can be reached from the Java side (cf. Section 3.3). In that sense the native part of the Java Class Library is a weird machine that can be triggered with crafted input values (cf. Section 2.3).

But, vulnerabilities might also be introduced upward the stack. Application developers widely reuse library code in order to efficiently produce applications. However, library code usually runs in the same security context as the application code. We consider this a violation of the Principle of Least Privilege (cf. Section 3.2) as it is rarely necessary to endow the library code with the complete permissions of the application code. Developers require tools to detect and remove excessive privilege usage in parts of the library that their application does not depend upon.

The Object-Capability Model (cf. Section 2.4) limits the means by which authority can be transferred between parts of a program. Thus, authority can be tracked and analyzed much easier than without these rules. The underlying assumption is that vulnerabilities related to authority transfer (e.g., the Confused Deputy, cf. Section 3.4) can be averted using the rules of the model. This assumption has not yet been shown to be true.

Part II.

Monitoring

A Slice of the Layer Cake

In the previous part, we identify many open challenges in software security. We present the danger of uncontrolled memory access in low-level languages, which is still a large problem in the wild despite various elaborate defense mechanisms. This does not only affect programs written in languages such as C or C++, but also Java programs as we have shown with the example of `sun.misc.Unsafe`. We describe the effect that reflective facilities can have on encapsulation guarantees.

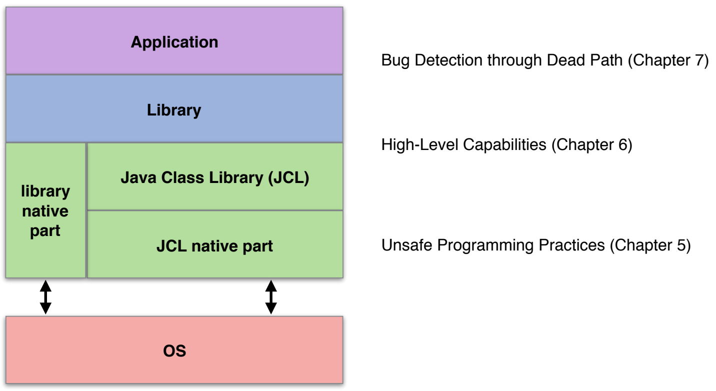
In this part of the thesis, we present three different program analysis approaches that monitor the static representation of a program and derive a set of warnings if problems are detected. All of them use the Java platform as their exemplifying case study. However, the concepts presented are portable to other platforms with little or no alteration. Also, all of the approaches have in common that they target the developer as their primary user focus. The analyses, thus, form a set of monitoring tools for developers to use in secure software development. They help to address the issues presented in the previous part.

In general, the purpose of monitoring is the detection of possible policy violations. The output of a monitoring system can then be used to guide ex-ante or ex-post isolation. Traditionally, monitoring is perceived as being a run-time technique. However, an analysis of the static representation of a program can also be considered as monitoring, as it also observes a state (the static representation) and infers a rating on policy violation. In a static-analysis-based monitoring system the policy is encoded in the detection criteria of the analysis.

In the construction of our monitoring defense mechanisms, we assume Baltopoulus and Gordon's *Principle of Source-Based Reasoning* [BG09]. It states that security properties of compiled code should follow from review of the source code and its source-level semantics. In our case, this applies to LLVM IR and Java Bytecode as the "source code" of our analyses. Therefore, covert channel attacks and attacks that void this assumption are out of scope for the guarantees we give.

When constructing monitoring schemes for the Java platform it is necessary to include every layer of the platform in the design of the analysis. In order to produce any observable effect, the Java Runtime Environment has to interact with the underlying operating system. Two parts of the platform are responsible for this binding to the operating system. There first is the Virtual Machine, the runtime engine for programs that have been compiled to Java Bytecode. The second is a large part of the Java Class Library that has been implemented in C or C++. This part can be (and has been) used to break the higher-level security measures of Java.

Therefore, we designed and implemented an analysis that targets unsafe programming practices in C and C++ code. We identified these unsafe programming practices by


Application
Library
library native part
Java Class Library (JCL)
JCL native part
OS
Bug Detection through Dead Path (Chapter 7)
High-Level Capabilities (Chapter 6)
Unsafe Programming Practices (Chapter 5)

inspecting known categories of previous bugs in the native section of the JCL. We show that we can effectively find possibly vulnerable parts in the native section of the JCL using this technique.

The major purpose of the native implementations is to provide access to system capabilities such as network sockets or the filesystem. In order to limit the access to those capabilities, the Java platform provides an access control system on the principle of stack-based access control. However, when composing an application from third-party libraries it is hard to control the permissions of these libraries in this model. We developed an analysis to detect the capability usage of a library, so that effective counter-measures can be taken in case a library exceeds its expected capability set.

The implementation of effective access control is a challenging task for developers (cf. Section 3.3). Small programming mistakes may lead to either non-functional or overly permissive code. These mistakes usually manifest themselves in infeasible code paths as do many other classes of errors. We found that a large number of these errors can be found with inspecting the causes of dead paths in the program. To find these paths and their causes, we implemented a data-flow sensitive analysis based on abstract interpretation.

To conclude this part of the thesis, we provide an overview over related work on monitoring the static state of software systems.

4. Assessing the Threat of Unsafe Languages

In contrast to languages like Java, which are considered *safe* languages, programming languages such as C or C++ allow operations directly on the systems memory. Parameters to these operations have to be carefully checked in order to prevent vulnerabilities in the program that can be exploited by attackers.

As Sergey Bratus et al. [BLP+11] present in their paper "Exploit Programming - From Buffer Overflows to 'Weird Machines' and Theory of Computation", this is notoriously hard for programmers and despite great care attackers still find parts of code to misuse for exploits. They state that creating an exploit is programming with crafted data in order to borrow code snippets that make up *weird instructions*, abstract instructions, which can be multi-step and spread over the program. These form *weird machines*, that can reach states in the state space of an automaton view of the program that were expected to be unreachable when just inspecting the desired behavior. A security vulnerability explodes the state space: the program can be forced to perform operation sequences, which the initial design did not intend. Therefore, detecting possible weird instructions in a codebase and removing them can actively prevent attackers from misusing it.

Literature in the security area knows these kind of attacks as *Return-Oriented Programming (ROP)*. The term was coined after the particularity that attackers seek those borrowed code snippets which can return to any point the attacker wishes. Elias Levy's (aka Aleph One) classic paper "Smashing The Stack For Fun And Profit" [Lev96] from 1996, which is the most referenced memory-corruption paper of all times [Mee10], explained these attacks first. He illustrates how to overflow small buffers using environment variables and gives hints on how to find buffer-overflow vulnerabilities in applications and operating systems.

Several approaches to detect and prevent these kinds of attacks exist [CPM+98, Pap12]. However, as soon as a new protection mechanism is invented, an attack to counter it is also devised. Carlini and Wagner [CW14] show that ROP attacks are still a problem.

In this chapter, we present an approach that can detect unsafe programming practices and highlights functions which could be exploited by memory corruption attacks. It can, thus, help enhance software security by lifting the task of finding vulnerable code parts from specialized problem categories to a general detection of unsafe programming practices. We apply this approach to the native part of the Java Class Library (JCL) and find interesting correlations between unsafe programming practices and patterns of exploitable bugs.

4.1. The Case of Java's Native Part

The Java platform is a widely used runtime environment installed on devices from mobile and embedded devices over desktop computers to cloud environments. It is equipped with a basic class library (JCL) to support efficient development of software applications running on the platform. Java uses an abstract memory model to prevent programs running on the platform from affecting system software and other applications. As Java programs cannot read from or write to arbitrary addresses in the memory, it is considered a *safe* language.

However, as Java programs need to interact with the underlying operating system to achieve observable effects, such as user interaction, filesystem access, and network operations, the JCL is shipped with a considerable amount of native code written in C/C++: Its size is about 620k lines of code in OpenJDK 6 and 640k lines of code in OpenJDK 7/8[1]. This native code base is a large liability to the Java platform.

This code base has full access to the complete memory area of the virtual machine, including the heap of Java programs. It can read or modify this heap freely without the restrictions imposed by information hiding or the access control model of Java. Thus, by exploiting vulnerabilities in the native part of the JCL through contrived input values, attackers can gain access to system resources or disable Java's security mechanisms. These vulnerabilities possess a significant magnitude of threat when attackers choose Java applets as their delivery mechanism. Applets may run undetected in a user's browser, allowing access to the client machine over networks. Using this technique, attackers may gain full access to the process the Java Virtual Machine (JVM) runs in and, therefore, the user's full security context. When breaking the security mechanisms of Java, attacking code is able to load new code from a distant host and execute it by spawning another process. Depending on the user's access permissions, the attacking code is then able to make itself resident in the target system.

The powerful capabilities of JCL's native part result in a significant potential attack surface. This threat is real: JCL's native part has been actively exploited and still is. For instance, the exploit filed in the NVD[2] under CVE-2015-4760[3] uses a vulnerability in the native implementation of 2D font rendering in the JCL to achieve buffer overflows. This allows for full access to the JVM instance from the applet used as the delivery mechanism.

The above observations clearly motivate the need for carefully analyzing Java's evolving native code base to discover potential vulnerabilities of any kind. One approach to do so, would be to apply existing techniques and tools that specifically target particular bug/vulnerability categories, e.g., buffer overflows. However, there are two problems with this approach. First, as reported by Tan and Croft [TC08], there are vulnerability patterns that are specific to JCL and, thus, not covered by tools for finding general bugs/vulnerabilities, such as buffer overflows. Second, tools that specifically target particular bug categories are ineffective in analyzing potential security threats in JCL's

[1]Measured using `wc -l` on `*.c`, `*.cpp`, and `*.h` files.
[2]National Vulnerability Database – `https://nvd.nist.gov/`
[3]`https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-4760`

native code base, because new bug patterns arise with new exploits being published constantly, continuously rendering the list of bug patterns to look for incomplete. Indeed, since the publication of the JCL's native-code-vulnerability categories by Tan and Croft [TC08], various new vulnerability categories have been identified in JCL's native code base, e.g., Invalid Array Index vulnerabilities.

In this chapter, we propose a different approach to identify vulnerabilities by searching for unsafe usages of certain programming-language features. We inspected the code of the bugs found in Tan and Croft's study [TC08] and identified four unsafe programming practices as major causes for vulnerabilities in the native code of the JCL. These unsafe programming practices are: Functional impurity, use of pointer arithmetic, use of pointer-type casts, and dynamic memory management. Neither of these practices can be omitted when writing code for JCL's native part, however, all of them bear the risk of being vulnerable if their use is not properly protected.

We developed four distinct static analyses using the LLVM framework [LA04] to scan JCL's native part for these four categories of unsafe practices in C/C++ code. We compile the complete native source code of the JCL using LLVM and apply our static analyses. We found that the majority of the functions in JCL's native part are impure; about 40% of the code uses pointer-type casts; pointer arithmetic or dynamic memory management was detected each in about 20% of the native functions. By triangulating the results of the individual analyses, vulnerable hot spots in the implementation are highlighted for further inspection: We found 94 of the 1,414 public native functions (approx. 7%) to be the most vulnerable hot spots.

We foresee several usage scenarios for our analysis infrastructure. It can help security experts to: (a) characterize the vulnerability level of their code, (b) guide code reviews towards more critical areas and efficiently skip over safe functionality, (c) point out where additional checks for safety, e.g., penetration tests, are necessary, and (d) localize areas that need special fortification, e.g., via confinement in sandboxes.

We evaluate our analysis infrastructure by performing studies to answer the following research questions: (a) are the results of the analyses accurate, (b) are our four analyses independent with regard to pinpointing problematic parts of code, (c) is there a (strong) correlation between accumulations of unsafe practices and known vulnerabilities? We argue that our analyses are precise by construction. We show that all four analyses are notably independent from each other, i.e., they will find different issues and, hence, not overlap in what they detect. A case study on vulnerability categories from Tan and Croft's study and current exploits reveals that our approach effectively points out three out of the five vulnerability categories considered and provides at least helpful results for the other two categories.

To summarize, we make the following contributions:

- The design and implementation[4] of an analysis infrastructure that detects unsafe programming practices based on the LLVM compiler infrastructure (presented in

[4]The source code and raw data is available here: `http://www.st.informatik.tu-darmstadt.de/artifacts/peaks-native/`

Section 4.3).

- A detailed evaluation of the approach, including (a) an analysis of its precision, (b) an analysis of the independence of its four parts as a means of justifying the mix of analyses supported by our infrastructure, and (c) a case study on known vulnerabilities of the native part of the JCL (Section 4.5)
- A detailed study of the occurrences of problematic programming practices in the native part of the JCL in OpenJDK 8 (Section 4.4).

We provide an introduction to Java Security w.r.t. the Java Native Interface (JNI) and a precise threat model in Section 4.2. Section 4.6 concludes this chapter with a summary of the benefits and limitations of the approach.

4.2. Problem Description

4.2.1. Java Security and the JNI

The Java platform has several protection mechanisms that prevent untrusted code from executing sensitive operations [GE03]. The most significant feature is Java's memory safety, i.e., references to objects can only be created and passed through Java primitives; direct access to the memory is prohibited. Thus, features like information hiding can effectively be enforced. In order to restrict the access to sensitive operations like file access, Java makes use of stack-based access control enforced mainly via the `SecurityManager` class.

The JNI is the only way for programs written in Java to interface with the operating system and other native code written in C/C++. Native methods can be invoked like any other Java method. The implementation of the methods can use any feature available in C or C++. In addition, the JNI provides an interface to interact with the Java side. Therefore, native implementations are also able to create and modify Java objects as well as to handle and throw exceptions. However, it is possible to manipulate the memory of the Java heap directly, ignoring information hiding. Besides, native code is not guarded by Java's security mechanisms.

4.2.2. Our Attacker Model

The Java platform allows for the execution of code from diverse – even untrusted – locations. Besides the standard execution model of a Java application located on the client machine, Java features two other models: (a) execution as an Applet in the client's browser, and (b) execution via Java Web Start, where applications from remote locations are executed on the client machine, outside the browser. In both of these models the default policy for Java forbids the loading of non-local native code for security reasons.

We assume that our attacker has no possibility of tampering with hardware and is, therefore, strictly limited to the software side. Furthermore, we assume that the active security policy disallows non-local native code, otherwise it would be easy for an attacker to introduce exploit code. Also, we do not consider the user's browser or the Java part of the JCL as an attack surface in this chapter, although both systems have been attacked successfully in the past (e.g., CVE-2013-0809[5], CVE-2013-1491[6], CVE-2013-1493[7]).

In this chapter, we target attacks that try to spot a vulnerability in the native part of the JCL and then exploit these vulnerabilities by executing the affected methods with malicious parameters. Outside of Java's memory safety, attackers will thus mainly try to cause accesses to parts of the memory they are not supposed to have access to. Accordingly, our analyses focus on detection mechanisms that enable such violations.

4.3. Analysis Design

The analyses we created to inspect the JCL's native methods are written in C++ and based on the LLVM framework [LA04]. LLVM is a compiler infrastructure that aims at providing analyses and optimizations at compile-, link-, and runtime for programs of any programming language. It defines its own intermediate representation (IR) on which the analyses are run, the so called LLVM IR. This IR is an assembly-like single-static-assignment language that allows a uniform representation of code for all programming languages. All our analyses have to run on a whole compilation unit, thus we implemented them as `ModulePasses` in LLVM that allow interprocedural analyses by looking at the whole IR module.

In pursuit of analyzing them, we compile the native part of the JCL for the Linux version of OpenJDK 8[8] using LLVM and a build process modified by us. We, thus, analyze the code in the `share/native` and `solaris/native` subfolders. However, the procedure can be ported to Windows, which will allow to analyze the Windows version as well. We then use `opt` to perform our analyses on the compiled bitcode in LLVM IR. Every analysis implementation iterates over the IR module's functions and looks for native function definitions that are accessible from the Java side as a starting point for the analysis. These function's names start with `Java_` by convention. Every analysis can trace native function calls to internal native functions, but those visible to the Java side serve as the entry points.

We designed and implemented four distinct analyses that we combine to achieve a threat rating of native implementations in the JCL for every function. These analyses look for functional purity, use of pointer arithmetic, pointer type casts, and dynamic memory management. In the following, we explain all four analyses, give a motivation for their respective usefulness, and describe their design and implementation.

4.3.1. Functional Impurity

In their paper *Verifiable Functional Purity in Java* [FMSW08], Finifter et al. define pure functions as being deterministic, meaning that the function only depends on its

[5]`https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0809`
[6]`https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-1491`
[7]`https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-1493`
[8]Build 132 - March 3rd, 2014

Algorithm 1 Functional Purity Analysis

```
function isPureFunction(Function $f$)
    if $\neg f$.onlyReadsMemory then return false
    for all Instructions $i$ in $f$ do
        if $i$.mayThrowException $\vee$ $\neg i$.mayReturn then return false
        for all Operands $o$ in $i$ do
            if $o$.isGlobalVariable then return false
        if $i$.isJavaCall then return false
        if $i$.isCall $\wedge$ isPureFunction($i$.calledFunction) = true then return false
    return true
```

parameters and yields equal results for equal arguments. Also it has to be side-effects free, i.e., the function has no visible effect other than generating a result. Finifter et al. also outline a set of benefits of such functions w.r.t. to software security and safety. One of these benefits is reproducibility, i.e., the fact that a pure function will deterministically produce the same result no matter where and when it is called. Another benefit is that the function's inputs are clear and explicit in the arguments.

Furthermore, if untrusted code is functionally pure it allows us to hold a set of security assumptions. First of all, due to the lack of side-effects, a pure function naturally executes in a sandbox-like environment and cannot tamper with the rest of the program. Secondly, if a pure function processes untrusted and potentially malicious data it is sufficient to mistrust the function's output, the execution itself cannot be harmful (excluding denial of service-like attacks that aim at depleting resources or preventing termination). It follows that pure functions have only limited capability to leak through overt channels as the return value is the only one. Still the function can leak through covert channels to impure functions via e.g. timing and resource usage.

There is no single universally-accepted formal definition of functional purity, but we follow the definition by Finifter et al. as it is most appropriate to give us the security assertions we are looking for.

Our analysis to detect impure functions (cf. Algorithm 1) first checks for functions that have been annotated with the `readonly` attribute by LLVM. The LLVM Language Reference Manual [Pro14] defines this flag as being attributed to functions that do not write through any pointer arguments (including `byval` arguments) or otherwise modify any state (e.g., memory or control registers) visible to caller functions. The function, however, may dereference pointer arguments and read state that may be set in the caller. A readonly function always returns the same value (or throws an identical exception) when called with the same set of arguments and global state. It cannot throw an exception by calling the C++ exception-throwing methods. We consider all functions not having this attribute as impure, as they will cause write-related side effects.

We then process each operand in the function's instruction sequence and check if a global variable is used by either reading from it or writing to it. This is necessary, as the `readonly` attribute tolerates global variable reads. We also check every instruction for additional side effects, specifically throwing an exception or lack of a defined return. If either is the case, we consider the function as impure.

Finally, we need to check for calls to impure functions, which would naturally render the caller impure as well. Therefore, we transitively apply our analysis to any called function and if any of them is detected as impure, the calling function is also impure. As calls might be recursive, we added a simple detection mechanism based on a set of visited and currently-inspected functions.

4.3.2. Pointer Arithmetic

Pointer arithmetic is the process of manipulating a pointer with an arithmetic operation. For instance, array access in C is implemented as pointer arithmetic, so that (`arr[i]` is the same as `*(arr + i)` (Note that both operations implicitly take the size of `arr`'s base type into account). Pointers and the ability to manipulate them are fundamental to the C programming language's performance (and insecurity as we will see).

A significant amount of research has been conducted on the security risk pointer arithmetic can induce, however, for the sake of brevity of presentation, we will discussion only two papers. Younan et al. [YPC+10] refer to the fact that buffer overflows enable diverse attacks and – while languages like Java provide inherent protection – C and C++ lack such a mechanism (at the benefit of performance and brevity). These protection mechanisms prevent attackers from abusing buffer overflow vulnerabilities by introducing run-time bound checks whenever pointer arithmetic happens and justify the enduring necessity of pointer-arithmetic related checks with a huge number of buffer overflow vulnerabilities still existing.

Avots et al. [ADL+05] describe a number of assumptions, called "common usage". Everything that falls outside of these assumptions can be seen as potentially dangerous and thus illustrates why pointer arithmetic can be critical. Among these assumptions, they mention that pointers to an object should only be derivable from a pointer to the same object, that pointers are only retrievable with the correct type of the underlying structure specified and that arithmetic is applied only to pointers pointing to an element of an array to compute another element in the same array. As we have seen, pointer arithmetic can be dangerous if used in an incorrect manner or if insufficient checks take place.

LLVM uses an instruction called `getelementptr` (GEP) to get the address of a subelement of an aggregate data structure, which can be an array or struct. This instruction is the only one in LLVM IR that is used to perform pointer arithmetic. It will be automatically inserted for pointer arithmetic by LLVM's `clang` front-end and can, thus, reliably indicate pointer arithmetic in the original C/C++ code. The instruction's first argument is always a pointer or a vector of pointers that is used as a base for the address calculation. The additional arguments index into the aggregate data, one argument for each level of nesting.

So, in order to detect pointer arithmetic in a given function, the analysis (cf. Algorithm 2) looks for GEP instructions. If all indices in the GEP instruction are zero, it indicates that the instruction only retrieves a pointer and does not perform pointer

Algorithm 2 Pointer Arithmetic Analysis

```
function usesPointerArithmetic(Function $f$)
    for all Instructions $i$ in $f$ do
        for all Operands $o$ in $i$ do
            if $o$.isGEP $\wedge \neg o$.allIndicesZero $\wedge \neg o$.isStruct then return true
        if $i$.isCall $\wedge \neg i$.isJavaCall $\wedge$
            usesPointerArithmetic($i$.calledFunction) $=$ true then return true
    return false
```

```java
/**
 * Returns the ADLER-32 value of the uncompressed data.
 * @return the ADLER-32 value of the uncompressed data
 */
public int getAdler() {
    synchronized (zsRef) {
        ensureOpen();
        return getAdler(zsRef.address());
    }
}
```

Listing 4.1: The Java Implementation of the `getAdler` Method of the `java.util.zip.Deflater` Class

arithmetic, which is considered benign. The GEP instruction is used for both array and structure access. We filter out the structure accesses, because we consider them as harmless for the following reason: An illegal index to an array access leads the pointer out of bounds, yet structure elements are always accessed by name in C and, thus, cannot be abused in the same manner. As there may be a cascade of pointers finally pointing to a structure we recursively determine if a pointer eventually points to a structure.

If the GEP instruction is found to neither have all zero indices and to not operate on a structure, the analysis flags the function as using pointer arithmetic. We also consider a function as using pointer arithmetic when one of the functions called in this function is flagged in that way.

4.3.3. Pointer Type Casts

Type casts between integers and pointers appear frequently in native functions that use the JNI, because Java does not have an explicit type for pointers. If a C pointer needs to be passed through the Java side, it is converted to an integer and back to a pointer when it comes back to native code. An example of this can be found in the `getAdler()` method of the `Deflater` class (cf. Listing 4.1) and its native implementation in Deflater.c (cf. Listing 4.2).

Whenever a Java program instantiates an object from the `Deflater` class, its constructor instantiates a private field with a new object of type `ZStreamRef` that encapsulates the address of a C structure in a private long integer field.

If the Java program then calls the `Deflater` object's `getAdler()` method, the method

```
JNIEXPORT jint JNICALL
Java_java_util_zip_Deflater_getAdler(JNIEnv *env, jclass cls, jlong addr)
{
    return ((z_stream *)jlong_to_ptr(addr))->adler;
}
```

Listing 4.2: The C Implementation of the `getAdler` Method of the `java.util.zip.Deflater` Class

invokes its native counterpart. The native method receives the pointer to the C structure as a long integer in an argument. It takes the parameter and casts it to a pointer, before accessing and returning the desired field of the structure. As the C pointer is treated as an integer in the Java code, we call this code pattern *pointer-as-integer*. The safety of pointer-as-integers is ensured by storing the integer values in private fields and declaring all native methods private, to avoid their direct invocation. If this access control is intact and enforced, it is a sufficient protection. An attacker can then only indirectly invoke the native methods and never tampers with the critical pointer-as-integers.

However, if a Java program has access to the reflection API, it can change private fields at runtime, including those containing pointer-as-integers, as Tan and Croft [TC08] already pointed out. A native method, like `getAdler()`, can then be used to read arbitrary memory locations by passing it the target address minus the offset of the field it is supposed to return. Writing arbitrary memory works similarly with methods that update data in a struct given through a pointer-as-integer. Even though the default policy for untrusted Java code normally disallows reflection, enabling it can have more severe consequences than a user might assume. While it is obvious that reflection allows for a violation of access control, it does not allow arbitrary read and write operations in pure Java programs, thus memory safety remains intact. Only when the aforementioned problematic methods can be passed pointers-as-integers through the JNI, reflection will enable arbitrary read and write operations.

As Java's native code is steadily evolving, we argue that it is worth highlighting the use of pointer-as-integers, even if they only appear dangerous once reflection is allowed. An error in the consistent declaration of all native methods as private to a Java class would render them exploitable. Furthermore, casting between pointers and integers could be used to circumvent detection of pointer arithmetic, if first a pointer is cast to an integer, then arithmetic operations are performed on it and it is finally cast back. As no arithmetic operations on pointers took place, this case will not be flagged by our pointer-arithmetic analysis, while a pointer has been effectively manipulated. This seems to be uncommon behavior in non-malicious code like the JCL's native code, nonetheless it would be highlighted by our analysis, if it had been accidentally introduced.

Our analysis (cf. Algorithm 3) detects any cast between an integer and a pointer type. It checks for every type cast in the instruction sequence of a given function, whether it casts an integer type to a pointer type or vice versa. Specifically, the LLVM instructions detected are the `inttoptr <type> <value> to <type2>` and `ptrtoint <type> <value> to <type2>` instructions. If such a type cast is detected, the function is marked

Algorithm 3 Pointer Type Cast Analysis

```
function usesPointerTypeCasts(Function $f$)
    for all Instructions $i$ in $f$ do
        if $i$.isCast $\wedge$ ($i$.isIntToPtrCast $\vee$ $i$.isPtrToIntCast) then return true
        if $i$.isCall $\wedge$ $\neg i$.isJavaCall $\wedge$
            usesPointerTypeCasts($i$.calledFunction) = true then return true
    return false
```

```c
#define jlong_to_ptr(a) ((void*)(a))

#ifdef _LP64 /* 64-bit Solaris */
typedef long jlong;
#else
typedef long long jlong;
#endif
```

Listing 4.3: Definitions Taken from src/solaris/native/common/jlong_md.h and src/solaris/javavm/export/jni_md.h

as containing pointer type casts.

Thereby, we detect a complete class of casts that looks much less uniform in C code than in IR code. For example the `getAdler()` method in the `Deflater` class (discussed earlier) casts a `jlong` to a pointer via the `jlong_to_ptr()` function. Its definition and the definition of a `jlong` are shown in Listing 4.3. Example code and compiled LLVM IR bitcode is presented in Listing 4.4 and Listing 4.5.

The `jlong_to_ptr()` function performs a cast to a `void` pointer, which is what we find represented in the LLVM IR by a `inttoptr` instruction.

We perform this analysis transitively for any called function. In detecting any kind of pointer type casts in a function, we assume that all of them are critical w.r.t. security, regardless of whether they are applied to input values or local variables.

4.3.4. Dynamic Memory Management

C and C++ both allow a program to dynamically manage an application's memory. The functions `malloc`, `calloc`, `realloc`, and `free` of the C standard library manage memory dynamically at runtime. C++ adds object orientation and, thus, `new`, `new[]`, `delete`, and `delete[]` reserve and release memory for new objects / object arrays. Memory, once

```c
#include "jni_md.h"

void* f(jlong x) {
  return jlong_to_ptr(x);
}
```

Listing 4.4: Example Usage Code for `jlong_to_ptr`

```
define i8* @f(i64 %x) #0 {
  %1 = inttoptr i64 %x to i8*
  ret i8* %1
}
```

Listing 4.5: Compiled LLVM IR for `jlong_to_ptr`

allocated, has to be freed by the program eventually to avoid running out of memory. This process of manual memory management is error prone.

Akritidis [Akr10] motivates why dangling pointers are potentially insecure. Dangling pointer are pointers that point to objects that have been `free`d already. The author outlines that attackers may use appropriately crafted inputs to manipulate programs containing use-after-free vulnerabilities into accessing memory through dangling pointers. When accessing memory through a dangling pointer, the compromised program assumes it operates on an object of the type formerly occupying the memory, but will actually operate on whatever data happens to be occupying the memory at that time. The author states that dangling pointers would likely abound in programs using manual memory management as its consistent implementation across large programs is notoriously error prone.

Attackers can exploit use-after-free vulnerabilities, if they are able to place attack code or a reference to it in the free memory where the dangling pointer is still pointing to. This is rather difficult, but techniques like *heap spraying* [DWW$^+$10] enable an attacker to assume the necessary control over the heap layout to reliably exploit such vulnerabilities.

As we analyze the JCL's native code, it is important to notice that there is a number of JNI function calls that resemble C's dynamic memory management and can lead to memory leaks and dangling pointers. For instance, the JNI allows native methods to manage Java heap memory by communicating with the Java garbage collector. Access to a Java integer array requires that a native method first invokes `GetIntArrayElements()` to get a pointer to the integer array. When the method finishes working on the array, it is supposed to invoke `ReleaseIntArrayElements()` to release the pointer. If it fails to do so, a memory leak occurs, as the Java garbage collector still considers the array as "in use" and does not free it. If the C pointer is used after it is released, this is similar to a dangling pointer. The array it points to may have already been moved. This can be as difficult to detect as lacking free calls to the C memory manager or dangling pointers caused by a use-after-free.

While the list of functions used by C and C++ to manage memory dynamically is clear, we have to manually scrutinize every function in the JNI API to check if it falls into the category of dealing with Java's dynamic memory management. Table 4.1 shows a table of the functions we consider critical, grouped as related pairs. We compiled this list based on the JNI's documentation [Ora14b]. The most critical are the functions of the families `Get<PrimitiveType>ArrayElements()` and `Release<PrimitiveType>ArrayElements()`, where `<PrimitiveType>` stands for any of the eight primitive types. They allow access to a Java array by pinning it down in memory, thus, preventing the garbage collector

Function name(s)
`Get<PrimitiveType>ArrayElements()`, `Release<PrimitiveType>ArrayElements()`
`GetPrimitiveArrayCritical()`, `ReleasePrimitiveArrayCritical()`
`GetStringChars()`, `ReleaseStringChars()`
`GetStringUTFChars()`, `ReleaseStringUTFChars()`
`GetStringCritical()`, `ReleaseStringCritical()`
`NewGlobalRef()`, `DeleteGlobalRef()`
`NewLocalRef()`, `DeleteLocalRef()`
`NewWeakGlobalRef()`, `DeleteWeakGlobalRef()`
`PushLocalFrame()`, `PopLocalFrame()`

Table 4.1.: Memory-Managing JNI Functions

Algorithm 4 Dynamic Memory Management Analysis

```
function usesDynamicMemoryManagement(Function $f$)
    for all Instructions $i$ in $f$ do
        for all Operands $o$ in $i$ do
            if $o$.isGEP $\wedge$ $o$.isJNIDynMemFunction then return true
        if $i$.isCall $\wedge$ $i$.isDynamicMemoryCall then return true
        if $i$.isCall $\wedge$ $\neg i$.isJavaCall $\wedge$
          usesDynamicMemoryManagement($i$.calledFunction) = true then
          return true
    return false
```

from removing it. All the other functions in the table have less impact, but all of them can cause memory leaks and/or dangling pointers to some extent.

To find calls to these functions, our analysis (cf. Algorithm 4) inspects all `getelementptr` instructions in the function's instruction sequence and determines if they return a pointer to one of the JNI functions listed in Table 4.1. It also detects calls to `malloc`, `calloc`, `realloc` and `free` and C++'s `new`, `new[]`, `delete` and `delete[]`. If either is found, the function is marked as containing operations for dynamic memory management.

The analysis is invoked transitively on every called function and assumes that called Java functions are uncritical since they cannot contain manual dynamic memory management.

4.4. Unsafe Coding Practices in the JCL

The results of the analyses are displayed in Table 4.2. The first two columns indicate the packages functions belong to, the third column shows the number of functions in that package; the remaining columns show for how many of them the respective analysis detected unsafe behavior. The fraction of unsafe functions to total functions is presented

Package		#Func.	Impurity		Ptr. Arith.		Type Casts		Dyn. Mem.	
com	sun	105	103	(98%)	47	(45%)	57	(54%)	33	(31%)
java	awt	66	27	(41%)	2	(3%)	7	(11%)	3	(5%)
	io	49	48	(98%)	9	(18%)	0	(0%)	5	(10%)
	lang	102	98	(96%)	8	(8%)	5	(5%)	11	(11%)
	net	58	58	(100%)	15	(26%)	0	(0%)	20	(34%)
	nio	9	9	(100%)	7	(78%)	9	(100%)	7	(78%)
	security	6	6	(100%)	0	(0%)	0	(0%)	0	(0%)
	util	46	36	(78%)	21	(46%)	34	(74%)	13	(28%)
sun	awt	281	269	(96%)	43	(15%)	179	(64%)	40	(14%)
	font	45	42	(93%)	19	(42%)	32	(71%)	26	(58%)
	instrument	10	10	(100%)	0	(0%)	10	(100%)	0	(0%)
	java2d	174	135	(78%)	52	(30%)	89	(51%)	60	(34%)
	management	92	92	(100%)	5	(5%)	3	(3%)	4	(4%)
	misc	15	15	(100%)	3	(20%)	2	(13%)	1	(7%)
	net	8	7	(88%)	2	(25%)	0	(0%)	1	(13%)
	nio	192	180	(94%)	20	(10%)	64	(33%)	11	(6%)
	print	12	12	(100%)	2	(17%)	0	(0%)	6	(50%)
	reflect	19	19	(100%)	0	(0%)	0	(0%)	0	(0%)
	security	109	108	(99%)	18	(17%)	40	(37%)	66	(61%)
	tools	10	10	(100%)	2	(20%)	0	(0%)	3	(30%)
	tracing	5	5	(100%)	1	(20%)	0	(0%)	2	(40%)
	util	1	1	(100%)	1	(100%)	0	(0%)	1	(100%)
Total		1,414	1,290	(91%)	277	(20%)	531	(38%)	313	(22%)

Table 4.2.: Results in OpenJDK 8 b132 grouped by the first two packages. Fraction of total functions is given in parenthesis.

in parenthesis after the values.

While nearly all packages consist mainly of impure functions, our analysis does not classify half of `java.awt`'s functions as such. All other kinds of unsafe behavior are detected very rarely as well in this package. Looking at the source code, we noticed that it contains a lot of `*_initIDs()` methods with an empty function body. Accordingly, none of our analyses detected dangerous behavior. Obviously, empty functions can be safely ignored.

For `java.nio`, our analyses show the highest detection rates of unsafe behavior among all of the packages. Besides its purpose of providing I/O functionality, the package provides methods for copying between arrays. Therefore, it makes excessive use of JNI functions (rendering methods impure), array manipulations (involving pointer arithmetic), casts from integers to pointers to access external data, and the JNI's array pinning methods.

The `java.security` package stands out in that all six methods are impure, but no other analysis detects unsafe behavior. They all reside in the same source file and simply return the result of a call to the Java Virtual Machine. What happens in the JVM is out of scope and, thus, functional purity cannot be asserted. Nevertheless, we can

# Analyses	# Func.
4	94
3	172
2	479
1	561
0	108
Total	1414

(a) with functional impurity

# Analyses	# Func.
3	94
2	172
1	495
0	653
Total	1414

(b) without functional impurity

Table 4.3.: Number of Functions in OpenJDK8 b132 Ranked by the Number of Analyses Reporting Them

assume that no dangerous pointer arithmetic, pointer type casts, or insecure dynamic memory management happens on the Java side. We made a similar observation for the `sun.reflect` package, which contains 19 functions, all of them impure, because they just return the result of a call to the JVM.

A look at the total detection numbers reveals that about twice as many functions use casts between pointers and integers than pointer arithmetic. Considering that the latter is involved in every array access and the former is normally unusual behavior for most other code, a clear particularity of the JDK's native code shows.

Table 4.3 shows how many functions are detected by how many analyses at once. The left table (a) shows the full distribution with functional impurity. As the functional impurity analysis is flagging more than 90% of the functions in total, we also show the distribution without this analysis in the right table (b). Regardless of this, the top-rated 94 functions in both tables are the same. Reviewers can now concentrate on these top-rated functions and work their way towards the lower-rated functions, guiding their efforts efficiently toward the most critical parts. Moreover, 108 functions (more than 7%) are classified as functionally pure and not containing any of the other unsafe practices and, thus, can be considered safe and will not need to be reviewed further.

4.5. Evaluation

In our evaluation we inspect three research questions in detail:

RQ1 Are the analyses precise in finding unsafe practices?

RQ2 Are the analyses notably independent from each other, in that sense that they detect different vulnerabilities?

RQ3 Are the analyses able to point out the causes of known vulnerabilities?

4.5.1. Setup

Each analysis in this section is run on the *Openjdk-8-src-b132-03_mar_2014* version of OpenJDK 8, unless otherwise noted. In order to run analyses on the native libraries in the OpenJDK, we modified the JDK's makefiles, such that when the native C and C++ files are compiled during the build process, LLVM IR for each source file is emitted in parallel. The resulting bitcode files are then linked library-wise using LLVM's *llvm-link*, such that interprocedural analyses are possible.

The analysis implementations are integrated into the LLVM framework's source code and compiled with it. This enables us to use them via a parameter with the *opt* tool (LLVM's optimizer) that performs analyses and transformations on the Bitcode. Our combined analysis outputs its results as a true/false value for each function and each analysis into a file in the CSV format.

We detected 1,414 functions that are accessible from the Java side. Running all four analyses on them yields 5,656 boolean values. Running the complete analysis on the codebase takes around 15 seconds of user time, measured on an Ubuntu 14.04 LTS 64-Bit Linux operating system with an Intel Core i5-4200U CPU and 4 GB of RAM.

4.5.2. RQ1: Precision of the Analyses

Static analyses are usually evaluated in terms of soundness and precision. As we implemented our analyses specifically to capture unsafe programming practices that are the direct result of using specific language features, our analyses are by design sound and precise as we will argue in the following. However, as we apply all of the analyses interprocedurally to determine if a called functions is flagged by the respective analysis, we are directly depending on the precision of the used call-graph algorithm. This means that any technique that makes call-graph algorithms more precise can be used to receive more accurate results from our analysis. For the remainder of this section, we will, thus, discuss the precision of the intraprocedural analyses.

Our functional-purity analysis is conservative. We use the `readonly` attribute, which is applied automatically to functions without writes to pointer arguments and global state. We add an extended check for global state read access, a behavior that the `readonly` attribute does not exclude. Thus, we classify functions as impure, if they either use writes to pointer arguments or access/change global state. As this is our definition of impurity, we have no false positives and no false negatives.

We build our analysis on LLVM IR bitcode and, therefore, all kinds of pointer operations are already represented by the `getelementptr` instruction. We safely detect and filter out instructions working on structures and simple pointer reads or writes. Thus, we are left only with those instructions using pointer arithmentic and have no false negatives. This includes array accesses as it can be seen in CVE-2015-4760[9] that unchecked array access operations can also lead to serious vulnerabilities. However, as we cannot statically determine the complete memory layout, we cannot actually tell if pointer arithmetic or array-access operations are benign or hostile. But, as we want to report all occurrences of unsafe practices in code, we do not consider benign operations as false positives.

For pointer type casts, analysing compiled code pays off as well. A source code analysis

[9]`https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-4760`

Agreement	Impurity	Ptr. Arith.	Type Casts	Dyn. Mem.
Impurity	1			
Ptr. Arith.	0.2836	1		
Type Casts	0.4406	0.6252	1	
Dyn. Mem.	0.3091	0.8190	0.6124	1

Table 4.4.: The Agreement Among the Analyses

has to consider all macros and `typedef`s, as they may contain new integer types and conversions (cf. Listing 4.3). In LLVM IR, all casts between integer types and pointers are represented by exactly two instructions (`inttoptr` and `ptrtoint`). As we want to find all type casts of this kind and all of them are represented by these two instructions, there cannot be any false positives or false negatives.

When detecting memory management it is important to have a complete list of all functions allowing programmers to access this feature. In our analysis these are the functions from the standard library and those provided by the JNI. To our knowledge both lists are complete and thus the analysis cannot produce false negatives. The only reasons where our analysis would produce false positives, is when functions in this list would not be offering access to dynamic memory management.

Therefore, we conclude that our analyses are indeed precise w.r.t. their detection purpose.

4.5.3. RQ2: Independence of Analyses

In order to answer the second research question, we measure the agreement between the four analyses, i.e., in how many cases two separate analyses report the same function. Table 4.4 shows the amount of the pairwise agreement for the four analyses.

Impurity shows the lowest agreement rates with the other analyses. We see a high agreement between the analyses for pointer arithmetic and for dynamic memory management. Yet, a high agreement does not imply a high correlation. Any two analyses that both mark many functions as either critical or non-critical will have a high agreement, simply because the latter is likely to occur, even if the cause is just random coincidence.

A measure to take that into account is the *Pearson product-moment correlation coefficient* that is defined as the covariance between two samples divided by the product of their standard deviations. The coefficient ranges on the interval $[-1, 1]$, with -1 being total negative correlation, 0 being no correlation and 1 being total positive correlation. The coefficients for our analysis are presented in Table 4.5.

As we can see, there is a slightly positive correlation between all the analyses, likely because all of them naturally agree on negative detection for empty function bodies. It stands out, though, that pointer arithmetic and dynamic memory management correlate more than expected. Looking at the functions that use both pointer arithmetic and dynamic memory management, we notice a certain pattern: A function uses one of the array-pinning methods, works with the array, and releases it. An example is easily

Correlation	Impurity	Ptr. Arith.	Type Casts	Dyn. Mem.
Impurity	1			
Ptr. Arith.	0.1530	1		
Type Casts	0.1578	0.1287	1	
Dyn. Mem.	0.1653	**0.4536**	0.1071	1

Table 4.5.: The Pearson Product-moment Correlation Coefficient for Each Pair of Analyses

```c
JNIEXPORT jint JNICALL
Java_java_util_zip_Adler32_updateBytes(JNIEnv *env, jclass cls, jint adler, jarray b, jint
      off, jint len)
{
    Bytef *buf = (*env)->GetPrimitiveArrayCritical(env, b, 0);
    if (buf) {
        adler = adler32(adler, buf + off, len);
        (*env)->ReleasePrimitiveArrayCritical(env, b, buf, 0);
    }
    return adler;
}
```

Listing 4.6: A Native Method from src/share/native/java/util/zip/Adler32.c

found in the JNI's zlib glue code and is displayed in Listing 4.6. The function pins down the Java array `b` (influencing Java's dynamic memory management) and assigns it to `buf`, a pointer to the array. After ensuring that the operation was successful, the zlib method is called with, among others, the array pointer plus an offset (performing pointer arithmetic). Then, the array is released (again influencing Java's dynamic memory management). This pattern is found frequently and explains the correlation between the two analyses.

The overall low correlation between the pairs of analyses indicates that they detect different unsafe coding practices, implying an overall good separation. This leads us to conclude that our analyses are indeed independent, so that they point out different vulnerabilities.

4.5.4. RQ3: Effectiveness on Known Vulnerabilities

In order to answer our third research question, we applied our analyses to five different vulnerability categories. We choose four out of the six bug patterns from Tan and Croft's study. We omitted the *C pointers as Java integers* pattern, as we build a specialized analysis for this that completely covers this issue, and the *Insufficient Error Checking* pattern, because it is the only one not based on unsafe programming practices and, therefore, not in our focus. We added the *Invalid Array Index* pattern, since it is regularly found in actual exploits of the native part of the JCL.

As our approach only points out unsafe practices, we do not spot the actual cause of the vulnerability, but only highlight accumulations of such unsafe practices to guide

Vulnerability Category	
Invalid Array Index	Detected
Exception Mishandling	Detected
Race conditions	Not detected
Buffer Overflows	Not detected
Memory Leak	Detected

Table 4.6.: Effectiveness on Known Vulnerabilities

code reviews. Our evaluation metric for this research question is, therefore, as follows: We deem our analysis to be effective in this case study, if it points towards the native functions involved in the vulnerability.

For every set of native functions, we outline if and by which analyses their *current* versions are detected. While these vulnerabilities do not exist anymore in these current versions, the potentially dangerous basic patterns are likely to persist throughout increasing versions. For example, a function using dynamic memory management without sufficient checks, thus vulnerable to a buffer overflow, will likely be fixed in a higher version by introducing the necessary checks. Nonetheless, it still uses dynamic memory management and the pattern persists, as it should, because even the improved checks might contain errors.

Invalid Array Index CVE-2013-2463[10] and CVE-2013-2465[11] both exploit an invalid array indexing vulnerability in the `storeImageArray()` function. This internal native function is used by the three Java accessible native functions `convolveBI()`, `transformBI()`, and `lookupByteBI()` in the class `sun.awt.image.ImagingLib`.

These three methods are used by the Java method `filter()` in the same class. This method works on objects of class `BufferedImage` and returns an object of that very class. An exploit, like the one described in [Jax13], can trigger the vulnerability through a Java applet which creates and transforms images that were initialized with contrived parameter values, causing a call to the `filter()` method somewhere down the applet's call chain. The special parameter values cause a buffer overflow in the vulnerable native method and this allows an attacker to gain additional privileges, deactivate the security manager and subsequently spawn a shell, thus, gaining full access to the system.

The three vulnerable methods are all detected by most of our analyses: `convolveBI()` and `transformBI()` are classified as impure, using pointer arithmetic, and performing dynamic memory management, `lookupByteBI()` is detected by all of our analyses. Comparing this to the overall results of the `sun.awt` package, we observe that detection by three or even by all four analyses is unusually high. Our analyses only points out exactly these three methods as particularly unsafe.

[10]`https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2463`
[11]`https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2465`

```c
static void* xmalloc(JNIEnv *env, size_t size) {
    void *p = malloc(size);
    if (p == NULL)
        JNU_ThrowOutOfMemoryError(env, NULL);
    return p;
}

#define NEW(type, n) ((type *) xmalloc(env, (n) * sizeof(type)))

static const char * const * splitPath(JNIEnv *env, const char *path){
    [...]
    pathv = NEW(char*, count+1);
    pathv[count] = NULL; [...]
}
```

Listing 4.7: An Excerpt of src/solaris/native/java/lang/zip/UNIXProcess_md.c (Java version 6 update 2)

Exception Mishandling One of the vulnerabilities found by Tan and Croft [TC08] is a dynamic memory allocation (presented in Listing 4.7), which is succeeded by a non-null check. In case the allocation fails, an error is thrown. In contrast to Java code, exceptions thrown in the native code do not disrupt the control flow.

When `xmalloc()` throws an error, the control flow is returned to the callee and not to the Java Virtual Machine. This means that if the `malloc()` fails, an exception will be thrown, but the `pathv[count] = NULL;` instruction will be executed anyway, dereferencing unallocated memory and crashing the JVM. The current versions of the JCL transfered this critical code part to the `effectivePathv()` function. This function now explicitly returns `NULL` to the JVM upon allocation failure.

Our analysis classifies this function as impure, performing both pointer arithmetic and dynamic memory management. Again, being detected by three of our four analyses, the function stands out in the package `java.lang`.

Race Conditions Tan and Croft [TC08] describe another class of vulnerabilities. The native function `Java_java_io_UnixFileSystem_setPermission()` performs calls to the external `stat` and `chmod` functions to get information about a file and modify permissions. If an attacker manages to manipulate the file reference in between the calls to `stat` and `chmod`, a file may get permissions only another file was allowed to get.

`Java_java_io_UnixFileSystem_setPermission()` is only detected as impure, none of the other analyses match. This is very common for the `java.io` package, and low overall. In this case, our analyses do not point out the vulnerable function. As none of the analyses seek race conditions, this is expected.

Buffer Overflows Another vulnerability discussed by Tan and Croft [TC08] is one that can lead to a buffer overflow. It lies in the internal native method `initLocalIfs()`. Older versions of the native code would call `fscanf()` with a format specifier that did not specify the input's length. Listing 4.8 shows the problematic part. The last format specifier `%s` will let the `ifname` array overflow, if the last entry on any line of

```
char ifname [32]; [...]
if ((f = fopen("/proc/net/if_inet6", "r")) == NULL) { return ; }
while (fscanf (f, "%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x "
                "%d %x %x %x %s",&u0,&u1,&u2,&u3,&u4,&u5,&u6,&u7,
                &u8,&u9,&ua,&ub,&uc,&ud,&ue,&uf,
                &index, &x1, &x2, &x3, ifname) == 21) { [...] }
```

Listing 4.8: An Excerpt of src/solaris/native/java/net/net_util_md.c (Java version 6 update 2)

```
JNIEXPORT jobject JNICALL Java_java_net_NetworkInterface_getByName0
    (JNIEnv *env, jclass cls, jstring name) {[...]
    const char* name_utf = (*env)->GetStringUTFChars(env, name, &isCopy);
    [...]
    ifs = enumInterfaces(env);
    if (ifs == NULL) {
        return NULL;
    } [...]
    (*env)->ReleaseStringUTFChars(env, name, name_utf);
[...] }
```

Listing 4.9: An Excerpt from src/solaris/native/java/net/NetworkInterface.c (Java version 6 update 2)

the `/proc/net/if_inet6` file is greater than 32 bytes. This is a network interface name, which can indeed be chosen to exceed 32 bytes. Newer versions of the code increase `ifname`'s size to 33 bytes and modify the format specifier to be `%32s`, which solves the issue. `initLocalIfs()` is used by 13 Java-accessible functions.

All these functions are impure according to our analysis, `Java_sun_nio_ch_sctp_SctpChannelImpl_send0()` performs pointer arithmetic and pointer-type casts, `Java_sun_nio_ch_DatagramChannelImpl_send0()` only does the latter, `Java_java_net_PlainDatagramSocketImpl_send()` and `Java_sun_nio_ch_sctp_SctpNet_bindx()` use dynamic memory management. These values are not particularly high, neither for the respective packages nor overall. None of our analyses specifically detects the format-specifier buffer-overflow pattern. An additional analysis that classifies format specifiers without size annotation in combination with a scanning function as unsafe could have caught this unsafe practice. Since this pattern rarely occurred in the inspected version of the JDK and the issue has an easy fix, we refrained from seeking such niche cases in favor of more concise overall results.

Memory Leak Tan and Croft [TC08] found a memory leak, where function `Java_java_net_NetworkInterface_getByName0()` uses `GetStringUTFChars()` to pin down a string but does not release it in certain control flows. Listing 4.9 shows the code. As we can see, if `enumInterfaces()` returns `NULL`, the function will return without calling `ReleaseStringUTFChars()`, thus causing a memory leak. Current Java versions simply put the `GetStringUTFChars()` call behind the conditional block and avoid potentially returning before releasing.

Our analysis marks the function as impure and as using dynamic memory management, just as expected, pushing it into the more critical half of its package's functions.

4.5.5. Limitations

We have seen that our analyses only find occurrences of unsafe programming practices and do not check if these practices actually lead to vulnerabilities of the program. Therefore, a valid next step would be to combine our analyses with a data-flow analysis that determines valid and given ranges for the operands of the unsafe operations. This would include a detection for guard clause and sanitizer functions. In combination with an additional analysis like this our analyses would then only report issues that are vulnerable. If we further combine this with a guided dynamic analysis and exploit generation, we could detect only those vulnerabilities that are actually exploitable. The four analyses presented here are the foundation for such an analysis stack and necessary for the efficient guidance of the downstream analyses.

Our four analyses presented in this chapter have some technical limitations. First, we can only inspect the body of functions that are defined in source code that has been compiled to Bitcode, which excludes functions from the standard library. Whenever such a function is used, it is declared on-the-fly in the LLVM bitcode and gets appropriate annotations with function flags based on standard implementations. For instance, if the standard implementation of such a library function is read-only, LLVM will declare it with a `readonly` flag. As these annotations are well tested and in ubiquitous use, we do not consider this as a problem. Also, because we analyze Bitcode emitted by the clang compiler, the Bitcode is naturally platform-dependent. As we use the Linux operating system to conduct our experiments, only the shared native code and the native code for Linux systems will be compiled, thus, excluding native code for other operating systems from the study. However, the native code for different operating systems is very similar in its functionality and mainly adapts to specific data formats and system calls.

4.6. Conclusion

In this chapter we presented four static analyses that find unsafe coding practices in C/C++ code. These practices are functional impurity, use of pointer arithmetic, pointer-type casts and dynamic memory management. Our four analyses accurately discover diverse potentially insecure behaviors. The analyses complement each other in that what one analysis misses another one may detect, and they can indeed point to bugs for most of the inspected bug categories. However, our analyses will also detect all kind of pointer arithmetic, which will include harmless array modifications. These benign operations have to be filtered out in a downstream analysis for guards and sanitizers.

We used these analyses to inspect the native part of the JCL in order to find accumulations of those practices that point toward possible vulnerabilities. In our study, we pointed out that a large number of the functions available in the JCL in fact use these unsafe practices. By inspecting five common vulnerability categories, we showed that the analyses are indeed helpful to detect vulnerabilities and in the majority of cases give strong hints to their causes.

This case study shows the strengths and weaknesses of our analyses with regard to pinpointing exploitable vulnerabilities (cf. Table 4.6). We saw that our analysis reliably hints toward critical functions, when it comes to bugs related to dynamic memory management, indexing, and memory leakage. Pointer-type casts have not appeared as explicitly exploitable yet. Race conditions and overflows caused by scanning functions like `fscanf` are not directly hinted to by the analysis except for marking the respective functions impure. Unfortunately, the available data set used for the experiment is too small to make more decisive statements. We found it hard to evaluate against vulnerabilities that reside in Java's native code in a large scale since none of the available online databases mark native vulnerabilities explicitly. Nevertheless, the results of our analyses are useful to characterize the level of vulnerability of native code, to guide code reviews effectively, to provide useful subsets for more expensive analyses (e.g. dynamic analysis), and, finally, to identify areas where intensive code review or strong isolation is needed.

5. Tracing High-Level Capabilities[1]

The efficiency of software development largely depends on an ecosystem of reuse [Boe99, Gri93, SE13, HDG+11]. Numerous software libraries are available that solve various problems ranging from numerical computations to user interface creation. The safe use of these libraries is an exigence for the development of software that meets critical time-to-market constraints.

However, when including software libraries into their products software developers entrust the code in these libraries with the same security context as the application itself regardless of the need for this excessive endorsement. For instance, a system that makes use of a library of numerical functions also enables the library to use the filesystem or make network connections although the library does not need these capabilities. If the library contains malicious code it could make use of them.

In commonly used languages like Java no effective mechanism to limit or isolate software libraries from the application code exists. So developers face a dilemma: Either trust the component and finish the project in time or be secure, review the library's source code and possibly miss deadlines.

We propose to consider this excessive assignment of authority as a violation of the *Principle of Least Privilege [SS75]*. The principle states that every program should operate under the least set of privileges necessary to complete its job. In order to alleviate the described dilemma, we introduce an effective mechanism to detect the actual permission need of software libraries written in Java.

Drawing inspiration from Android, we construct a capability model for Java. It includes basic, coarse-grained capabilities such as the authority to access the filesystem or to open a network socket. As Java programs by themselves cannot communicate with the operating system directly, any interaction with those capabilities has to happen through the use of the Java Native Interface (JNI). By tracking the calls backwards through the call graph, we produce a capability set for every method of the Java Class Library (JCL) and by the same mechanism towards methods of a library. We can thus effectively infer the necessary capabilities of a library using our approach. We can also infer the subset of these capabilities used by an application, as it may not use every functionality supplied by the library.

As the precision of our approach is directly depending on the precision of the algorithm used to compute the call graph of the library, we took several measures to compute a reasonably precise call graph while not compromising the scalability of the algorithm too severely.

[1] This chapter is based on and contains verbatim content of work previously published at Foundations of Software Engineering 2015 [HREM15].

We evaluated our approach by comparing our results against expectations derived from API documentation. We found that for 70 projects from the Qualitas Corpus [TAD+10], that we evaluated against, actual results exceeded expectations and produce a far more accurate footprint of the projects capability usage. Thereby, our approach helps developers to quickly make informed decisions on library reuse without the need for manual inspection of source code or documentation.

In our pursuit to mitigate the software developer's dilemma w.r.t. library reuse, we thus contribute the following:

- an algorithm to propagate capability labels backwards through a call graph (presented in Section 5.2.3),
- a labeling of native methods with their capabilities necessary to bootstrap the process (in Section 5.2.2),
- a collection of efficient analysis steps to aid the precision of common call-graph algorithms (explained in Section 5.2.3),
- an evaluation of the approach (Section 5.3) against extracted capability expectations from documentation.

We furthermore motivate our work in more detail in Section 5.1. Section 5.4 provides concluding remarks and discusses interesting challenges.

5.1. Motivation

One of the major drivers of efficient software development is the ability to reuse parts of software systems in the development of other systems in the form of software libraries. Software developers can thus concentrate on the key requirements of their programs and reuse functionality that is common to many software systems. Developing software this way leaves more time for the creation of new functionality, mitigates rewriting already existing functionality and also makes errors less likely as functionality in form of libraries gets used more often than their individual counterparts. It has been observed that programming with library reuse can speed up software development by over 50% [Boe99]. Therefore, library reuse is an important part of an efficient software engineering process.

The concept of library reuse has been adopted by many different programming environments from C and C++ over Java which ships with the Java Class Library to academic languages like Racket or even embedded languages. Library reuse is part of software development processes from the smallest wearable devices to the largest cloud-based solutions. For instance, the Java Runtime Environment is shipped with multiple libraries that are not part of the runtime but can be used by Java programs. Their functionality includes graphical user interfaces, cryptography and many more. By providing the functionality in form of libraries the Java platform gives developers the ability to use the functionality without implementing it first or leaving them out of the program if not needed. The C standard library is probably the most used software library worldwide.

It is loaded in basically any system, although in different versions, from mobile devices, desktop or laptop computers to cloud computing environments.

However, in order to be efficient developers rely on the described and observed behavior of the libraries. The source code of the libraries hardly ever gets reviewed. Even in open source projects the rigor of code reviews is quite lax [Bla14]. Examples of this can be seen in the quite prominent Heartbleed incident [DKA$^+$14], where the server-side implementation of OpenSSL had a major programming error that allowed clients to read the server's complete memory in 64kb blocks exposing inner state that can be and was used against the server. Although being a de-facto standard with a large user base the error was not discovered through a code review but through an exploit. What makes this exploit particularly interesting is that the error was in code not necessary for the pure functionality of SSL connections but for a convenience heartbeat functionality. Developers using server-side OpenSSL never opted in to such functionality.

As library code runs in the same security context as application code a developer automatically entrusts the library code with the same privileges the end user has provided to the application. She implicitly transfers the users trust towards her to the developers of the library. Thus, any capability the complete application may use is available to library code as well.

Moreover, consider a scenario where a central point for library delivery like Maven Central gets compromised. Many applications may be shipped with malicious versions of libraries and be delivered to end users without noticing.

Considering all the aforementioned problems it is rather surprising that developers still blindly trust libraries. Besides code reviews that may not be possible for a closed source component, very little tool support for developers in need to make the right choice of library is available. Developers can use tools like FindBugs [HP04] to check for well-known patterns of bad behavior in a library, but will only find instances of known problems not a full footprint of the library's critical resource usage.

As library reuse is an essential and widely adopted practice for software development it is crucial that developers have access to trustworthy software libraries. Trust in those libraries can be gained by inspecting them manually – which is often tiresome – or with bug detection tools – which is often not helpful. Hence library code runs within the same security context as the application code and it is seldom under the scrutiny of a code review, it may pose a significant risk when rolling out libraries as part of an application. Thus, new methods to determine the actual usage of system capabilities of software libraries are needed.

Therefore, in this chapter, we present an approach to capability inference for software libraries written in Java. In the following sections we present our approach and the bootstrapping necessary for it in detail.

5.2. Capability Inference

We use the OPAL framework [EH14], an extendable framework for static analysis of Java Bytecode based on abstract interpretation, to perform all static analyses relevant for our inference algorithm. After giving an overview of our approach, we will discuss it in detail.

5.2.1. The Approach in a Nutshell

The purpose of the analysis presented in this chapter is to uncover the capability footprints of software libraries written in Java. Making them aware of these footprints helps developers to make informed decisions as whether to integrate a library into their application based on how it uses system resources. For instance, consider a library for the decoding and display of images. An application developer considering to use this library expects the library to use functionality for reading from the filesystem and for displaying images on the graphical user interface. However, it is unlikely that this library needs to perform operations to play sounds or to open network sockets. If the library does use these capabilities, a developer might want to invest more time into the inspection of the library.

In order to access system functionalities a library written in Java will have to make use of the Java Native Interface (JNI) either directly or through the Java Class Library (JCL) (cf. Figure 5.1). Our analysis uncovers the use of system capabilities through the JCL. It consists of three main steps.

We first manually inspected all native functions of the core part of the JCL. According to their name, implementation and documentation we assigned them capability markers manually.

These sets of capability markers are then propagated backwards through a call graph of the JCL and the library under test. Through refinement and filtering steps on the call graph we ensure a precise but practical result. In the example presented above the native functions using the capabilities for the filesystem and the graphical user interface are traced backward to library functionality (cf. Figure 5.2).

In a last step, the results of the analysis are consolidated to receive either a complete footprint of the library or a partial footprint depending on the use of library functionality, because only parts of the library may have been used in the application. In our example from Figure 5.2, the union set of all used capabilities in the library consists of the capabilities for the filesystem and the graphical user interface as expected, but it does not include capabilities for playing sound and using network sockets. Developers can thus save the time for an inspection of the library's source code.

5.2.2. Bootstrapping

In Java, the capabilities considered in our case, are created outside of the Java Virtual Machine and are reached only via a call through the JNI (cf. Figure 5.1). As any system resource like the filesystem, network sockets or the graphical user interface is a matter of the underlying operating system – when viewed from the Java side – this assumption naturally holds. As libraries seldom bring their own native code, they rely on the native functions and their wrapping functions inside the JCL. Therefore, to compute meaningful

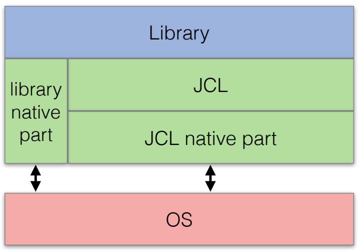


Figure 5.1.: Architectural Layers for Java Libraries

results for arbitrary Java libraries it is necessary to first infer capability usage for the JCL.

We identified 14 distinct capabilities presented in Table 5.1, whose usage we want to track. They represent different critical resources that are accessible through the native part of the JCL. The `CLASSLOADING`, `DEBUG`, `REFLECTION`, and `SECURITY` capabilities refer to internal resources of the JVM used to achieve dynamic code loading, instrumentation, introspection, or securing code, respectively. Even though these capabilities are not system resources, we decided to include them as they represent authority to circumvent other mechanisms like information hiding, memory safety, or the security model. For instance, using reflection it may be possible for a library to call into filesystem functionality although our analysis does not recognize it, because the call graph we use to extract the information does not contain a respective call edge. We also decided to include a marker capability (`NATIVE`) for all native calls that do not access system resources (e.g. `java.lang.StrictMath.cos()`). Although these functions do not provide access to a capability, they still are native functions with the possibility to read and write arbitrary memory locations in the JVM heap. All other capability names have been chosen w.r.t. the system resource they represent.

To retrieve a list of all native method stubs of the JCL, we use the OPAL framework. The core part of the JCL is included in the rt.jar file shipped with the Java Runtime Environment. Therefore, we limited our bootstrapping analysis to this file. All libraries that ship with the JCL are based on the rt.jar and those that also have native code are mainly concerned with graphical user interface display (e.g. JavaFX). We implemented a simple analysis that collects all methods with the `ACC_NATIVE` marker in the `method_info` structure [LYBB14]. This corresponds to `native` methods in Java source code. On the 64-bit version for Windows of the OpenJDK version 7 update 60 this analysis returned 1, 795 methods.

We manually assigned each of these methods the set of its capabilities. To assign the correct capabilities, we reviewed the source code of the native implementation of the method. We also took the naming and the documentation of the function into account, as they provide insight into its purpose. Column $\#m$ in Table 5.1 presents the number of methods for each capability in the result set of the bootstrapping process. As some methods are annotated with more than one capability, the sum of these figures is higher than the number of methods in the result set.

capability	# of methods	description
CLASSLOADING	24	Definition and loading of classes
CLIPBOARD	9	Access to the system clipboard
DEBUG	5	Debugging instrumentation
FS	377	Access to the filesystem
GUI	449	Graphical user interface
INPUT	10	Retrieve values from input devices
NATIVE	419	No specific facility, but calls into native code
NET	274	Network socket access
PRINT	54	Print on physical printers
REFLECTION	78	Introspective access to objects
SECURITY	14	Influence the security mechanisms of Java
SOUND	36	Play or record sound
SYSTEM	126	Operating system facilities
UNSAFE	85	Direct memory manipulation

Table 5.1.: Capabilities Identified in the JCL

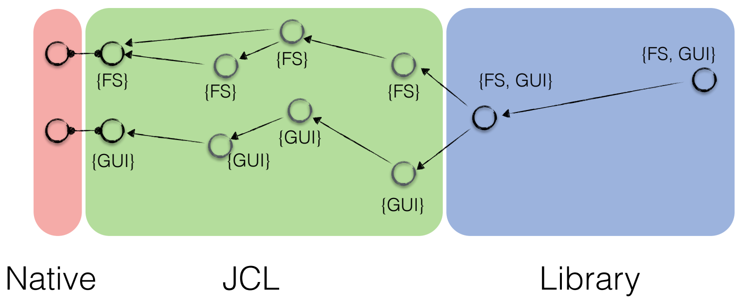


Figure 5.2.: Call Graph with Annotated Capabilities

5.2.3. Building Capability Footprints

Our analysis propagates capabilities backwards from JCL methods to the library's API (cf. Figure 5.2). The propagation consists of the following three major steps which are detailed afterwards:

1. The call graph that includes the JCL and the analyzed library is build.
2. The capabilities identified in the bootstrapping phase are propagated backwards through the graph. As part of the propagation, call edges in the call graph are filtered that will result in excessively overapproximated capability sets.
3. In the last step, either the complete footprint of the library (e.g. `{FS,GUI}` in Figure 5.2) or a footprint based on the usage context of the library is build.

Call-Graph Construction Step We build the call graph using a variant of the VTA algorithm that we implemented in OPAL. Compared to the original VTA algorithm [SHR$^+$00], the algorithm only propagates precise type information intra-procedurally and does not first calculate a whole-program *type propagation graph*. However, intra-procedurally the analysis is more precise as it uses the underlying abstract interpretation framework and is therefore flow- and path-sensitive. This way we are able to construct an equally precise call graph, but be more efficient at the same time. To further increase precision, we added two steps. First, we added a shallow object-insensitive analysis of all `final` or `private` fields to OPAL to determine a more precise type of the values stored in the respective fields than the field's declared type. Second, we added a basic data-flow analysis of all methods that return a value to determine a more precise return type than the declared one if available. Both information is used during the call graph construction. Overall, the number of call edges of the resulting call graphs are comparable, but our VTA algorithm is more efficient than the original one.

For the call graph's entry points we use all non-private methods and constructors, all static initializers and those private methods that are related to serialization. The latter are added to capture the implicit calls performed by the JVM when serializing or deserializing objects.

Propagation Step In the bootstrapping phase, we manually assigned each native method of the JCL the set of its capabilities. To determine a library method's capabilities, we propagate the capabilities through the call graph by traversing the call graph backwards, starting with the native methods of the JCL. While traversing the graph, we propagate the initial sets of capabilities to compute the set of capabilities attached to each method of the JCL and the library. At the end, each method of the library is annotated with a set of its capabilities based on the transitive native calls it performs.

Unfortunately, this naïve propagation of the capabilities results in a hugely overapproximated capability distribution as basically every method of the JCL would be annotated with all capabilities. This is due to the fact that often no precise type information for the receiver of a method call is available, so all possible receivers have to be taken into account by the analysis although these receivers are never called.

For instance, if the `toString()` method is called by some method and the runtime type of the receiver is unknown and, hence, the upper type bound `java.lang.Object` needs to be assumed, the call graph will contain $1,304$ edges[2] for this call.

These edges will propagate capabilities to methods that will in practice never be reached. Hence, we filter these problematic edges during the capability propagation step. For instance in the above example, it is rather unlikely that each of these $1,304$ alternative implementations will be called in practice from this call site. However, as all of these implementations are annotated with their respective capability sets the method including the call site will be annotated with the union set of all these sets. We decided for filtering although it gives up soundness of the analysis by replacing the overapproximation with an underapproximation. We ignore calls to unlikely implementations and thus get a much stricter capability propagation that is closer to runtime reality. Alternatively, a points-to analysis could have been used to determine more exact receiver types, but as our approach needs to scale up to the JCL such a costly analysis is not permissible. We implemented two filters in our approach.

The first filter removes edges for method calls whose receiver type is `java.lang.Object`. Listing 5.1 shows an example where this filter effectively removes problematic call edges. The method `currentContent()` calls `toString()` on the field `o` whose receiver type is `Object`. Although the field is initialized with the concrete type `MyInt`, the call cannot be resolved to the implementation in this type, because the field is public and may be overwritten with an object of any other type. This results in call-graph edges to all implementations of `toString()` of subclasses of `java.lang.Object`. Our filter removes these edges from the capability propagation.

```java
public class A {
    public Object o;

    public A() { o = new MyInt(42); }

    public String currentContent(){ return o.toString(); }
}
```

Listing 5.1: Example for Inconclusive Type Information

[2]The `toString` method is implemented by $1,304$ JCL classes.

filter type	call edges	reachable methods
Without filter	2.068.946	102.896
Object filter	1.221.293	102.411
Abstract filter	1.974.261	101.656
Interface filter	1.322.949	98.728
Subtype filter	1.194.172	91.813
all filters	368.231	91.135

Table 5.2.: Effect of Filter Application in OpenJDK

The second filter handles a similar problem that occurs with the use of interface types and abstract classes. Like receivers, whose type cannot be refined more precisely than `java.lang.Object`, receivers of interface and abstract types also have a large number of subtype alternatives within the JCL. Again, the call-graph algorithm does not have any other choice than to include edges to all these alternatives, resulting in a over-propagation of capabilities.

However, in this subtype filtering process we perform a prefix match on the package name of callee and caller. We only propagate capabilities for alternative edges that point to a callee located in the same package as the caller, as we assume that the most relevant implementations often reside in the same package. For example, consider a call to the `compare` method of the `java.util.Comparator<T>` interface in some method of a class `C`. Further assume that the implementation of `compare` to which the call dispatches is inside an anonymous class located inside `C` and is thus implicitly in the same package as `C`. When detecting a call to the `compare` method our filter only accepts implementations in the same package, which in this case is only the one in the anonymous class.

Without filtering, the call graph of the OpenJDK version 7 update 60 (Windows, 64bit) includes roughly $2M$ edges and over $100k$ methods that transitively call into native methods. The exact figures are presented in Table 5.2. Applying the first filter regarding receivers of type `java.lang.Object` will drop approximately $847k$ methods ($\sim 40\%$). When using abstract receiver types for the second filter, the considered edges for capability propagation drop only to $1.9M$, but when we apply the filter to interface type receivers, the number drops to $1.3M$ edges. If we combine interface and abstract receiver types (named *Subtype filter* in Table 5.2), we have only about $1.2M$ edges. Finally, when combining both kinds of filters, we only consider about $370k$ edges. This reduces the processed edges by 82% and the number of methods in the inverse transitive hull by 13%.

As discussed before, overapproximation in the analysis produces overly populated results in our approach. Due to that we choose to filter, which has the subsequent effect that we find a capability set of a given library that is closer to the runtime situation. For instance, a simple call of the `read` method and `InputStream` as receiver type would imply the capabilities `CLASSLOADING`, `SOUND`, `NET`, `UNSAFE`, `FS`, `SYSTEM`, `SECURITY` and `GUI` which are eight out of the 13 capabilities in our model. In case of the more concrete type `FileInputStream`, the call of `read` would only yield the `FS` capability. Hence, we are underapproximating to that end that instead of assuming that all call edges are possible, we only consider edges where caller and callee are in the same package. Furthermore, we decided to apply our filter mechanism only for callees within the JCL as we did not find large sets of alternative call edges inside libraries.

Result usage and reporting The last step in the analysis is the construction of a report of the results. Depending on the developer's needs we can construct two different reports. First, by building the union set over all methods of a library we can construct the complete capability footprint of a library. Second, when taking the usage context of a library into account, we can report on the actual used capabilities as developers might not always use a library to its full extent and thus may not use capabilities included in the complete footprint.

Beyond our motivating usage scenario, where this information is displayed to the application developer to decide on libraries to use, it may also be interesting in other situations. Considering the example in our motivation regarding the Heartbleed incident, we could also display the difference between the complete footprint of a library and the used part. This can hint to features of a library that are unused by the application, just as the heartbeat feature was not used in the OpenSSL scenario. Also this information could be used to slice the library down to the used part in order to prevent unintended or malicious use. We explore this idea in Chapter 8.

The information we infer can also be interesting for library developers. We already compute a list of methods and the assigned capabilities which can be used to support code inspections for libraries. Because through this list developers have assistance to identify and find critical methods with ease, our approach can help to guide code inspections and make them more efficient.

By analyzing and collecting the information from open source libraries in a database, we could also build a recommendation system that, based on metrics like the delta from expectation or the number of used capabilities, could assist developers to find a suitable library.

5.3. Evaluation

In this section, we present the measures taken to evaluate the capability inference in our approach. We are guided by these research questions:

RQ1 Does the capability inference algorithm effectively find capability sets equal to developer's expectations of library capability usage?

RQ2 Does the capability inference algorithm exceed these expected sets by more true positive values?

Setup Developers make educated guesses on the capabilities used in a library when integrating it into their application. A math library, for example, should not use the

	CL	CB	DB	FS	GU	IN	NT	PR	RF	SC	SD	SY	UN
antlr	+		−	=	=		+	+	=	+		+	=
aspectj	+			+	+		+		=	+		+	+
axion	+		−	=			=		=	+		+	+
cglib	=			+					=	=			+
checkstyle	=		−	=	+	−	+		=	+		+	+
collections			−	=	−		−		=	+		+	+
commons-beanutils	=		−	=	−		−		=	=		+	+
commons-cli	+		−	−	−				=	+			+
commons-codec	+			=			=		=	+		+	=
commons-configuration	=		−	=	−		=		=	+		=	=
commons-fileupload				=			−		=	=		+	+
commons-io	=		−	=			=		=	=		=	+
commons-lang	=		−	=			−	−	=	=		=	+
commons-logging	=		−	=			+		=	=		+	+
derby	+		−	=			=		=	=		+	+
displaytag	=			=	−		−		=	+	−		+
easymock	+		−	+			+		=	+		+	+
geotools	+	+		=	+	+	=	+	=	=		+	+
gson	+								=	=		+	=
guava	=		−	=	−		=		=	=	−	+	=
guava-gwt									=				
guice	=			=	−		=		=	+		+	+
gwt	=		−	=	−	−	=	+	=	=	−	=	=
jFin_DateMath	+			=					=	+		+	+
jackson-annotations									−	+			
jackson-core	+		−	=	−		+	−	=	+		+	=
jackson-databind	+		−	=			=	−	=	=		+	=
jasml				+			+		=	+		+	+
java-hamcrest				+					=	+			
javassist	=		−	=			=		=	=		=	+
javax-inject													
javax-mail	=		−	=	−		=		=	=		+	+
jedit	−	−	−	=	=	−	=		=	=		=	=
jgroups	=		−	=	=	−	=	+	=	+		=	+
joda-time	+		−	=			+	−	=	=		+	+
jrat	+	+		=	=		+	+	=	+		+	+
jsoup			−	=			=		−	+			+
jspwiki	+		−	=	−		=	−	=	=	−	+	+
junit	+		−	=					=	+		+	+
log4j	=		−	=	=		=		=	=		=	=
lucene-analyzers-common	=		−	=	−		−		=	+		+	=
lucene-analyzers-icu	+			+			=		=	+		+	+
lucene-analyzers-kuromoji				−					=	+			+
lucene-analyzers-morfologik	+			+			+		=	+		+	+
lucene-analyzers-smartcn				+					=	+			+
lucene-analyzers-stempel			−	=					−	+			
lucene-classification									−				
lucene-codecs				−			−		=	+			=
lucene-core	=		−	=	−		=	−	=	=		=	=
lucene-demo				=					−	+		+	
lucene-facet				=					=	=		+	+
lucene-grouping									−	+			+
lucene-highlighter			−	+	−		+		=	+		+	+
lucene-join									=	+			
lucene-memory			−						−	−			
lucene-misc				=			+		=	+		+	+
lucene-queries							−		=				
lucene-queryparser	+		−	=					=	+		+	+
lucene-spatial	+			+					=	+			+
lucene-suggest				=			−		=				+
maven-core	+		−	=			−		=	=		+	+
maven-plugin	+		−	+			=		=	+		+	+
mockito	+		−	+			=		=	+		+	+
pooka	+	=	−	=	=	−	=	=	=	=		=	+
sandmark	+		−	=	=		=		=	+		+	=
slf4j			−	=			+		=	+		+	+
sunflow	+			=	=		=		=	+		+	+
testng	+		−	=			=		=	=		+	+
weka	+	=	−	=	=	−	=	=	=	+		=	=
xstream	=		−	=			=		=	=		+	+
total =	18	2	0	46	9	0	27	2	62	24	0	11	15
total -	1	1	40	3	14	6	10	6	7	1	4	0	0
total +	28	2	0	12	3	1	13	5	0	40	0	40	45

Figure 5.3.: Capability-Usage Observed in Libraries

network or the filesystem. To check such an assumption developers can leverage the documentation of the library. However, to save time developers might perform only a key phrase scan to search for the presence (or absence) of certain terms. In our evaluation, we mimic this process and use techniques from Information Retrieval to scan documentation for key phrases.

We use a subset of the Qualitas Corpus [TAD+10] incorporating 70 Java libraries. They were selected based on two criteria. First, the documentation for the library must be accessible. Second, we choose libraries with a low number of dependencies on other libraries, so that capabilities used by dependents will not influence the outcome of the experiments. That is because we assume that capabilities used by dependencies will only be documented in the original documentation of that library and not in the documentation of the library depending on it.

The selected libraries have different sizes and capture various domains. Ranging from ANTLR, which is a parser generator, to Weka, a collection of machine-learning algorithms, or Pooka, a mail client with user interaction, multiple domains are captured in the selection. Therefore, the capability expectations for these libraries vary as different domains require different capabilities. For instance, a developer will expect pooka to use the `GUI` capability, because it should display information for the user, while she would not expect it from ANTLR.

As all libraries need the Java Class Library to work, we set the baseline for all our experiments to the 64-bit build of the OpenJDK version 7 update 60 running under Windows.

We constructed a list of key phrases for each capability (cf. Table 5.3). For this, we used terms in the English language that can be assigned to the capabilities without ambiguity between capabilities. We derived these key phrases from the documentation of the JCL functions we annotated manually in the bootstrapping phase. As the `NATIVE` capability is just used as a marker, we do not assign any keyword to this capability.

In order to search in API documentation for key phrases, we first obtain the documentation either by downloading the complete set of documentation or by requesting it directly from the projects webserver. We then traverse the complete documentation starting at `allclasses-noframe.html`, which contains a complete list of all available classes, abstract classes and interfaces in the project. For each of the listed classes we extract the documentation given for each method of that class. This text as well as the method signature for reference is passed to Apache Lucene.

Lucene creates an inverted index using all tokens found inside the provided text. We then use this index to search for the key phrases in the key phrase list. Depending on whether we search for a word, e.g., ”classloader”, or a phrase e.g., ”load a class”, we use either a `TermQuery` or a `PhraseQuery`. While term queries have support for stemming, phrase queries do not. Stemming is a process where a term is reduced to its root, the stem. For example, the term ”paint” would match ”painting” as well as ”painted”. The use of stemming results in a more complete set of hits for this term.

For each project we record the capabilities whenever the respective key phrases are found in the documentation. The result of this process is a union set over the complete project and hence is expected to be the complete footprint of the library.

Results The results of the evaluation are presented in Figure 5.3. For each project a line in the diagram shows the expected and found capabilities. It uses the abbreviations already introduced in Table 5.3. A green square with an equality symbol (`=`) denotes that the capability was expected from the documentation and also found with the capability inference algorithm. A red square with a minus symbol (`-`) means that the capability was expected from the documentation, but not found. A blue square with a plus symbol (`+`) represents a capability that was found by the inference algorithm, but was not expected from the documentation. Empty squares represent capabilities that where neither found in documentation nor inferred.

For example, for the ANTLR project, the `FS`, `GUI`, `REFLECTION`, and `UNSAFE` capabilities were expected from key phrases in the documentation and also found through capability inference. The documentation also suggested the `DEBUG` capability, but no evidence for this was found in the call graph. Moreover, the `CLASSLOADING`, `NET`, `PRINT`, `SECURITY`, and `SYSTEM` capabilities were found, but not expected from documentation.

We measured the agreement of our capability inference with the results obtained from the key phrase search. For the agreement we consider every capability the analysis and the key phrase search agree upon – both positively and negatively. For example, for the ANTLR project the agreement is 87.50%. The mean agreement[3] in the inspected project set was 86.81% while the algorithm missed only 3.90% of the capabilities detected from documentation.

When looking closer to individual capabilities, we see a similar result (cf. Table 5.4). The only outlier here is the `DEBUG` capability. Moreover, the capability inference algorithm was able to find 14.1% more capabilities than documented in mean over all projects. It found evidence for undocumented use for all capabilities except `DEBUG`, `REFLECTION`, and `SOUND`. The `UNSAFE` capability was found in 1.8 times more projects than it was documented.

Figure 5.4 shows the capability distribution of each capability over all projects. For the `SYSTEM` and `UNSAFE` capability it can be seen that every documented capability use was successfully detected and for most other capabilities the figures are very close. However, in contrast to Table 5.4 we present the true positives and not the agreement here, so true negatives are not represented in the figure.

Discussion First and foremost, we are interested in meeting the developers expectations (RQ1) of the capability usage of libraries. As our approach has a mean agreement of 86.81% over the inspected projects, we clearly reach this goal. The capability inference only misses a mean of 3.90% in our experiments.

What is even more interesting is that the capability inference systematically discovers usage of capabilities that are not to be expected by the developers when they use documentation as a source of information (RQ2). While a mean of 14.1% more capabilities over all inspected projects were found only shows this as a tendency, when looking at technical capabilities like `CLASSLOADING` (66.67%), `SECURITY` (133.33%), `SYSTEM`

[3]As the values per project are already normalized, we use the geometric mean. Zero values are set to 0.001.

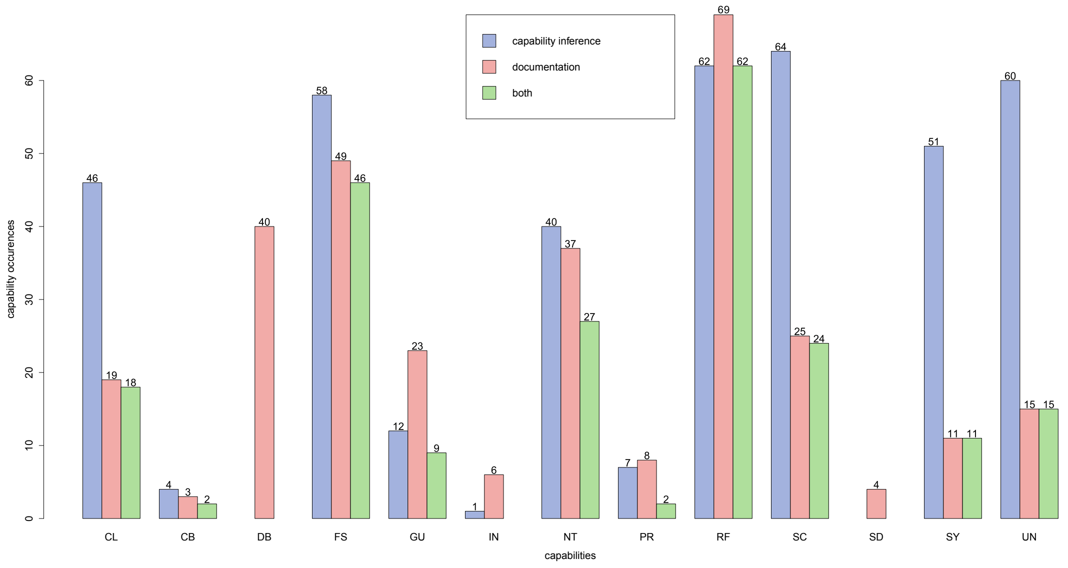


Figure 5.4.: Capability Distribution over Projects

(133.33%), and `UNSAFE` (180.00%), it is apparent that the use of these capabilities by a library is often not documented.

These technical capabilities may give hints towards whether a library might be vulnerable to exploits. For instance, as it may not be surprising that JUnit is using the `CLASSLOADING` capability, developers might not expect this from jFin_DateMath, which is a library for financial date arithmetic. However, in our evaluation results both libraries use the `CLASSLOADING` capability although not suggested by the project documentation. A quick inspection of jFin_DateMath revealed that two methods both named `newInstance`[4] use the `forName` method of `java.lang.Class`. According to Oracle's Secure Coding Guideline (Guideline 9-9)[ora14a], however, this method performs its task using the immediate caller's class loader. As both implementations of the `newInstance` method do not perform further checks on the provided class name and simply pass it to the `forName` method, attackers in an untrusted security context could use this so-called *Confused Deputy* [Har88] in order to execute code in the library's context given that it is more privileged than the context of the caller.

Results show that our capability inference algorithm works well for most of the capabilities. The capability with the least convincing results regarding its agreement is the `DEBUG` capability. Our approach misses every occurrence found through key phrase scanning. While we choose a rather narrow set of methods for this capability during the bootstrapping phase, the key phrases assigned to the capability occur in comparatively many method documentations. For instance, the phrase "Method that is most useful for debugging or testing; ..." occurred in the documentation of Jackson-core. While

[4]In `org.jfin.date.daycount.DaycountCalculatorFactory` and `org.jfin.date.holiday.-HolidayCalendarFactory`.

this phrase clearly does not imply any kind of JVM debugging on the methods part, it still is included in the results for the term query for the "debug" key phrase included in the key phrase list for the `DEBUG` capability. However, when we apply our algorithm to the jvm monitoring tool jconsole, we are indeed seeing the `DEBUG` capability included in the results. In general, this observation also applies to the `SOUND` and to the `INPUT` capability. This is what leads us to the conclusion that documentation might not be precise enough in these cases.

However, we receive rather good results for `CLIPBOARD`, `FS`, `NET`, `PRINT` and `REFLECTION` where we find most of the capabilities and even undocumented ones. This indicates that our capability inference algorithm is a helpful approximation. Moreover, the excellent results for the capabilities `CLASSLOADING`, `SECURITY`, `SYSTEM` and `UNSAFE`, where the inference significantly exceeds the expectations, are a very interesting outcome. It may indicate that API documentation is not a reliable source of information to build capability expectations in a security aware context.

Most projects in our evaluation set make good use of capabilities; one project is an exception. The Javax-inject project shows no capability usage w.r.t. our inference, but also no expectations can be extracted from the documentation. This is because the project consists entirely out of interfaces, which by definition cannot have concrete methods up to Java 7. We deliberately included the project in the evaluation set to inspect whether the documentation of this project may raise expectations. As it did not raise any and also by its contents could not use any capabilities, it is a good example for true negatives.

As the evaluation answers both research questions positively, we conclude that our approach is effective in helping to mitigate the developer's dilemma when making library choices for applications.

Threats to validity We only used API documentation for the extraction of developer expectations. However, there are alternatives, which we did not consider. In an open source setting, for instance, one could inspect the source code. Another alternative would be to statically analyze the code for the presence or absence of specific, interesting calls, as e.g., static bug checkers do. Manuals or other documents could also be a source of information to developers. However, we think that capability information should be denoted clearly in the public API documentation of a project.

We constructed the key phrase list manually from the observations we made while inspecting the source code and the documentation of the annotated native methods in the bootstrapping phase. It is possible that we missed key phrases that denote capabilities in API documentation. Moreover, it is possible that we included key phrases that are not suitable to detect capabilities and introduce many false expectations. Finally, the API documentation of the projects could be outdated [FL02], incomplete, or even wrong. In order to mitigate this threat, we reviewed the key phrase list and the results of search queries to remove questionable key phrases.

There are also threats to validity with regard to our inference approach. First, we are aware that we might introduce false negatives through our filtering phase. However, we use these filters to reduce false positives by removing call graph edges introduced by incomplete receiver type information. If we get rid of the filters while keeping our coarse-grained capability model, we would receive a result, where almost every time the complete set of possible capabilities would be found in every library. In order to mitigate the risk of introducing false negatives, we limit the filter to be only applied for receiver types inside the JDK. Second, the analysis we use is unaware of everything that is beyond the scope of the used call graph algorithm, so that we do not detect calls made through reflection and direct manipulation of the heap through native code.

The largest manual step in our approach is the bootstrapping of native methods of the JCL. As we inspected the implementation and documentation of these methods manually, we may have documented too less, too much, or incorrect capabilities for methods, even though we reviewed the dataset multiple times. In future work, we would like to exchange this process with an automated analysis of the syscalls made in the implementation. Beyond the process being more efficient, it is also repeatable for new versions of the JCL and applicable to native code shipped with libraries. It might also be more reliable in terms of precision.

5.4. Conclusion and Future Work

In this chapter, we presented a novel approach towards a capability inference for Java. It is based on a manually created dataset of native methods and their capabilities in the JCL. An automated process propagates these capabilities along a call graph through the JCL and a library in order to find the capability footprint of the library. To produce helpful results, we added a refinement step to the used call-graph algorithm and also apply filtering for alternative edges during capability propagation. Although filtering may impede the soundness of the approach, the produced results are closer to expectation.

Being able to produce a capability footprint for a library helps developers in their task of selecting libraries for their projects. With our analysis, a developer can get answers towards capability usage in a few seconds instead of inspecting the source code or the documentation manually. We show that the approach is able to find capabilities expected from documentation with an agreement of 86.82%. Moreover, we show that the approach exceeds these expectations by finding 14.1% more capabilities than expected from the documentation and produces a more accurate footprint of a library's actual capability usage.

We plan to study the accuracy of the approach regarding developer expectation more closely with the help of a user study. Developer experts will be asked to document their expectations regarding multiple software libraries. By this, we will be able to further determine the utility of the approach for secure software development.

Furthermore, we plan to infer the capabilities for native methods by an automated process rather than the manual process presented in this chapter. A static analysis on the native implementations of the methods can extract and trace system calls in order to achieve a reproducible result for new versions of the JCL. Furthermore, it is useful to include native library code into the process.

capability	key phrases
CLASSLOADING (CL)	classloader, load a class, classloading, jar
CLIPBOARD (CB)	clipboard, paste
DEBUG (DB)	jvm, debug, instrumentation, debugging
FS (FS)	filesystem, file, folder, directory, path
GUI (GU)	textfield, menubar, user interface, canvas, paint, editor, color, rendering, render
INPUT (IN)	press key, mouse, keyboard, trackball, pen
NATIVE (N)	
NET (NT)	network, socket, port, tcp, udp, ip, host, hostname, protocol, connection, http, imap, pop3, smtp, ssl, mail, transport, mime
PRINT (PR)	printer, paper
REFLECTION (RF)	reflection, class, field, method, attribute, annotation
SECURITY (SC)	security, security provider, policy, privilege, authority
SOUND (SD)	sound, midi, wave, aiff, mp3, media
SYSTEM (SY)	system, command line, execute, process, thread, environment, runtime, hardware
UNSAFE (UN)	pointer, memory address, memory access, unsafe

Table 5.3.: Expected Key Phrases for Capabilities

capability	agreement	miss	excess
`CLASSLOADING`	97.62%	2.38%	66.67%
`CLIPBOARD`	98.53%	1.47%	2.94%
`DEBUG`	42.86%	57.14%	0.00%
`FS`	94.83%	5.17%	20.69%
`GUI`	79.10%	20.90%	4.48%
`INPUT`	91.30%	8.70%	1.45%
`NET`	82.46%	17.54%	22.81%
`PRINT`	90.77%	9.23%	7.69%
`REFLECTION`	90.00%	10.00%	0.00%
`SECURITY`	96.67%	3.33%	133.33%
`SOUND`	94.29%	5.71%	0.00%
`SYSTEM`	100.00%	0.00%	133.33%
`UNSAFE`	100.00%	0.00%	180.00%

Table 5.4.: Results by Capability

6. Hidden Truths in Dead Software Paths[1]

Programming errors can have severe consequences on the secure of the program. When a programmer makes a mistake in the implementation of a security checks the complete security model might be affected, because it can be deactivated when exploiting the mistake. Thus, the development of secure software also entails the use of static bug finding tools.

Since the 1970s many approaches have been developed that use static analyses to identify a multitude of different types of issues in source code [CHH+06, AB01, GYF06, CCF+09]. These approaches use very different techniques that range from pattern matching [CHH+06] to using formal methods [CCF+09] and have very different properties w.r.t. their precision and scalability. But, they all have in common that they are each targeting a very specific kind of issues. Those tools (e.g., FindBugs [CHH+06]) which can identify issues across a wide range of issues are typically suites of relatively independent analyses. This limits the number of issues that can be found to those that are identified as relevant by the respective researchers and developers.

In this chapter, we present a generic approach that can detect a whole set of control- and data-flow dependent issues in Java Bytecode without actually targeting any specific kind of issue per se. The approach applies abstract interpretation to analyze the code as precisely as possible and while doing so records the paths that are taken. Afterwards, the analysis compares the recorded paths with the set of all paths that could be taken according to a naïve control-flow analysis that does not consider any data-flows. The paths computed by the latter analysis but not taken by the former analysis are then reported along with a justification why they were not taken.

For example, when implementing a security check as a condition to perform a critical operation, a mistake in the implementation may manifest as an infeasible path either containing the critical operation or the case where the operation is denied. In the first case, the security critical operation is never performed. While this does not pose a security risk, it is still undesirable as the functionality is also not available for authorized clients. In the second case, the operation is performed regardless of client authorization, which makes the code vulnerable.

The rationale underlying this approach is as follows. Many issues such as null dereferences, array index out of bounds accesses or failing class casts result in exceptions that leave infeasible paths behind. Hence, the hypothesis underlying the approach is three-fold. First, in well-written code every path between an instruction and all it's direct successors is eventually taken, and, second, a path that will never be taken indicates an

[1] This chapter is based on and contains verbatim content of work previously published at Foundations of Software Engineering 2015 [EHMG15].

issue. Third, a large class of relevant issues manifests itself sooner or later in infeasible paths.

To validate the hypotheses we conducted a case study analyzing the Java Development Kit (JDK 1.8.0_25). Additionally, we did a brief evaluation of the approach using the Qualitas Corpus [TAD+10] to validate the conclusion drawn from the case study. The issues that we found range from seemingly benign issues to serious bugs that will lead to exceptions at runtime or to dead features. However, even at first sight benign issues, such as unnecessary checks that test what is already guaranteed, can have a significant impact. Manual code reviews are a common practice and the biggest issue in code reviews is comprehending the code [BB13]. A condition that always evaluates to the same value typically violates a reviewer's expectation and hence impedes comprehension causing real costs.

The analysis was explicitly designed to avoid false positives. And, yet it identified a large number of dead paths ($\approx 50\%$ of all identified dead paths), which do not relate to issues in the source code. This is because the analysis operates on Bytecode and is as such sensitive to compilation techniques and a few other intricacies of the language. To make the tool usable [JSMHB13, BBC+10], we implemented and evaluated three heuristics to filter irrelevant issues.

Though we opted for analyzing the code as precisely as possible, we deliberately limited the scope of the analysis by starting with each method of a project then performing a context-sensitive analysis with a very small maximum call chain size. This makes the analysis unsound – i.e. we may miss certain issues – but it enables us to use it for large industrial sized libraries and applications such as the JDK. As the evaluation shows, the approach is effective and can identify a wide range of different issues while suppressing over 99% of all irrelevant findings.[2]

To summarize, we make the following contributions:

- We put forward the idea that infeasible paths in software are a good indication for issues in code and show that a large class of relevant issues does manifest itself sooner or later in infeasible paths.
- We present a new static analysis technique that exploits abstract interpretation to detect infeasible paths. The analysis is parametrized over abstract domains and the depth of call chains to follow inter-procedurally. These two features enable us to make informed reasonable trade-offs between scalability and soundness. Furthermore, the analysis has an extremely low rate of false positives.
- We validate the claims about the proposal in a case study of industrial size software; the issues revealed during the case study constitute themselves a valuable contribution of the paper.

This chapter is structured as follows. In Section 6.1 we discuss a selection of issues that we were able to find. After that, we discuss the approach in Section 6.2 along with

[2]The tool and the data set are available for download at `www.opal-project.de/tools/bugpicker`.

its implementation. The evaluation is then presented in Section 6.3. The chapter ends with a conclusion in Section 6.4.

6.1. Classification of Issues

In this section, we discuss the kind of issues that we found by applying our analysis to the JDK 1.8.0_25[3]. Given that the approach is not targeted towards finding any specific category of issues, it is very likely that further categories emerge from investigating the results of applying the analysis to other projects. Yet, the following list already demonstrates the breadth of the applicability of the proposed approach.

Obviously Useless Code In some cases, we were surprised to find code that is obviously useless in such a mature library as JDK. For illustration consider the code in Listing 6.1. In the method `isInitValue`, the developer checks whether an `int` variable contains a value that is smaller/larger than the smallest/largest possible value, which is obviously `false` and does not need to be checked for. Such useless code has two problems. First, it wastes resources. More importantly, it negatively impacts reasoning from the perspective of code that uses `isInitValue`. A developer has to understand that a call to `isInitValue` always returns `true`. This is most likely not obvious in the context of the calling method, as `isInitValue` suggest something different.

```java
boolean isInitValueValid(int v) {
  if ((v < Integer.MIN_VALUE) || (v > Integer.MAX_VALUE)) {
    return false;
  }
  return true;
}
```

Listing 6.1: Obviously Useless Code in `com.sun.jmx.snmp.SnmpInt.isInitValueValid`

Confused Conjunctions The binary conjunction operators in Java (`||` and `&&`) are a steady source of logic confusion not only for novices but also for experienced developers. Mixing them up will either lead to an expression that is overly permissive or one that is overly restrictive compared to the original intent.

```java
if (maxBits > 4 || maxBits < 8) {
  maxBits = 8;
}
if (maxBits > 8)
  maxBits = 16;
}
```

Listing 6.2: Confused Conjunction in `com.sun.imageio.plugins.png.PNGMetadata.mergeStandardTree`

[3]Some of the code samples shown in the following are abbreviated to better present the underlying issue.

In the example in Listing 6.2, a variable `maxBits` is checked if it is greater than 4 or (operator `||`) less than 8. This is always true as the partial expressions overlap. As a result, the `maxBits` variable is always set to 8 rendering the following check (Line 1845) useless. In this example, the developer probably wanted to use the `&&` operator instead.

```java
if (ix<0 && ix>=glyphs.length/*length of an array >= 0*/) {
  throw new IndexOutOfBoundsException("" + ix);
}
```

Listing 6.3: Confused Conjunction in `sun.font.StandardGlyphVector.getGlyphCharIndex`

In Listing 6.3 we have the dual situation. Here, the developer probably wanted to use —— instead of &&. Currently, the condition – as a whole – will always evaluate to false; an integer variable (`ix`) can never contain a value that is at the same time less than zero and also larger or equal to zero. Currently the precondition check is useless and `ix` could exceed the array's length or even be negative. In general, such useless precondition/post-condition checks may lead to deferred bugs at runtime which are hard to track down. For example, imagine that `ix` is just stored in a field for later usage and at runtime some other method uses it and fails with an exception. At that point-in-time the method that caused `ix` to contain an invalid value may no longer be on the stack and is therefore typically hard to identify.

In general confused conjunctions can render pre- or post-condition checks ineffective or mask paths that are necessary for correct behavior.

Confused Language Semantics The semantics of Java's `instanceof` check is in borderline cases sometimes not well understood. For example, the `instanceof` operator will return `false`, if the value is `null`.

```java
public boolean containsValue(Attribute attribute) {
  return
    attribute != null &&
    attribute instanceof Attribute &&
    attribute.equals(...(Attribute)attribute...));
}
```

Listing 6.4: `Instanceof` and `null` Confusion in `javax.print.attribute.HashAttributeSet.containsValue`

In the example in Listing 6.4, the developer first checks the parameter `attribute` for being not `null`. After that she checks if it is an instance of `Attribute`. However, the `instanceof` operator also checks if the object is not `null`, so the first check is redundant. But, given that the declared type of the parameter is `Attribute`, the respective value will always be either an instance of `Attribute` or `null`, hence the not-null check would be sufficient.

A second issue in this category is shown in Listing 6.5. In this case the developer checks if a newly created object is indeed created successfully. This is, however, guaranteed by Java's language semantics. In this case the developer most likely had a background in programming using the C family of languages where it is necessary to check that the allocation of memory was successful.

```
doc = new CachedDocument(uri);
if (doc==null) return null; // better error handling needed
```

Listing 6.5: New Objects are Never `null` in `com.sun.org.apache.xalan.internal.xsltc.dom.DocumentCache`

Instances of this category usually lead to redundant code that is detrimental to comprehensibility of the program.

Dead Extensibility Programmers may deliberately introduce *temporary* infeasible paths into code when programming for features that are expected to be used in the future. However, in some cases the implementation of a feature is aborted but no clean-up is done; leaving behind dead code. For example, a method that was defined by an interface with a single implementation and where the implementation of that method returns a constant value is an instance of dead extensibility.

We classify issues as belonging to dead extensibility only if (a) the package is closed for extension (e.g, everything in the `com.sun.*` or `sun.*` packages), (b) the development of the respective code is known to have ended (e.g., `javax.swing.*`) or (c) we find an explicit hint in the code.

```
// For now we set owner to null. In the future, it may be
// passed as an argument.
Window owner = null;

if (owner instanceof Frame) { ... }
```

Listing 6.6: Infeasible Paths from Unused Extensibility (taken from `javax.print.ServiceUI.printDialog`)

For example, in Listing 6.6 the variable `owner` is set to `null` but later on checked for being of type `Frame`. This will always fail as discussed previously. Here, the comment in the code however identifies this issue as a case of programming for the (now defunct) future.

Another instance of this issue can be found in `java.util.concurrent.ConcurrentHashMap.tryPresize`. In that case our analysis identified that a variable (called `sc`) always has the value 0 which leads to a decent amount of complex dead code. Given the complexity of the code we directly contacted one of the developers and he responded:

> “Right. This code block deserves a comment: It was not always unreachable, and may someday be enabled ...”.

In general dead extensibility primarily hinders comprehension; sometimes to a highly significant level as in the last case. Additionally, it may lead to wasted efforts if the code is maintained even though it is dead [EJJ$^+$12].

Forgotten Constant In case the declaration of a local variable and its first use are very distant, developers might have already forgotten about its purpose and value and assume that its value can change even though it is (unexpectedly) constant.

For example, in `javax.swing.SwingUtilities` in method `layoutCompoundLabelImpl` a variable called `rub` is set to zero and never touched over more than 140 lines of code. Then the variable is used in an expression where it is checked for being larger than zero and, if so, is further accessed. But, that code is never executed.

Issues in this category are often a hint to methods that need to be refactored because they are very long and hard to comprehend. Overall these issues are also causing maintainability problems, because they are hindering comprehension and the resulting dead code still needs to be maintained.

Null Confusion The most prevailing category of issues that our analysis discovered in the JDK is related to checking reference values for being (non)-`null`. The issues in this category can further be classified into two sub-categories: (a) unnecessary checks, which do not cause any immediate harm, and (b) checks that reveal significant issues in the code. We don't consider this distinction in the following sections because both categories are very relevant to developers [AP10].

An example of an issue of the first sub-category is a non-`null` check of a value stored in a (`private` and/or `final`) field that is known to only contain non-`null` values. Another instance is shown in Listing 6.7 - the variable `atts` is checked for being non-`null` and an exception is thrown if the check fails. Later in the code (Line 227 - not shown here) the variable is, however, checked again for being non-`null`, resulting in an infeasible `else` path.

```
if (atts != null)
  types.add(att);
else
  throw new IOException(paramOutsideWarning);
```

Listing 6.7: Ensuring Non-`null`ness in `javax.management.loading.MLetParser.-parse`)

Issues of the second sub-category are those where a non-`null` check is done after the reference was already (tried to be) dereferenced. An example of the latter category is shown in Listing 6.8. In this case, either the check is redundant or the code may throw `NullPointerException`s at runtime.

```
int bands = bandOffsets.length;
... // [bandOffsets is not used by the next 6 lines of code]
if (bandOffsets == null)
   throw new ArrayIndexOutOfBoundsException("...");
```

Listing 6.8: Late Null Check (taken from `java.awt.image.Raster.createBandedRaster`)

Range Double Checks We classify issues as *range double checks* if the range of a value is checked multiple times in a row such that subsequent checks are either just useless or will always fail. As in case of *Null Confusion* issues, such issues are benign if they are related to code that is just checking what is already known. However in other cases (more) significant issues may be revealed.

```
byte jcVersion = jc.javaSerializationVersion();
if (jcVersion >= Message.JAVA_ENC_VERSION) {
  return Message.JAVA_ENC_VERSION;
} else if (jcVersion > Message.CDR_ENC_VERSION) {
  return jc.javaSerializationVersion();
} ...
```

Listing 6.10: Contradicting Range Checks in `com.sun.corba.se.impl.orbutil.-ORBUtility`

An issue of the first category is shown in Listing 6.9. Here, the variable `extendableSpaces` is first checked for being zero or less (Line 1095) and if so the method is aborted. Later on the dual check (Line 1097) is unnecessarily repeated.

```
1095  if (extendableSpaces <= 0) return;
1096  int adjustment = (targetSpan - currentSpan);
1097  int spaceAddon = (extendableSpaces > 0) ?
1098                   adjustment / extendableSpaces : 0;
```

Listing 6.9: Useless Range Check in `javax.swing.text.ParagraphView$Row.-layoutMajorAxis`

An issue of the second category is shown in Listing 6.10. Here, the value of the variable `jcVersion` is first checked for being equal or larger than the constant `JAVA_ENC_VERSION`, which has the value 1. After that, the variable is checked again for being larger than the constant `CDR_ENC_VERSION`, which is 0. Hence, both checks are equivalent, but the code that is executed in both case clearly differs which makes it likely that it is incorrect.

Type Confusion Names of methods often explicitly or implicitly suggest that the handled or returned values have a specific type and that specific type casts are therefore safe. This, however, can be misleading and lead to runtime exceptions.

For example, the method `createTransformedShape` shown in Listing 6.11 suggests that a transformed version of the `Shape` object is returned that has been passed to it. However, the method always returns an instance of `Path2D.Double`. Now the method `getBlackBoxBounds` which is defined in `java.awt.font.TextLayout` calls `createTranformedShape` passing an instance of `GeneralPath` and casts the result to the very same type. This will always result in an exception as `GeneralPath` is not a subtype of `Path2D.Double`.

```
public Shape createTransformedShape(Shape pSrc) {
   if (pSrc == null) { return null; }
   return new Path2D.Double(pSrc, this);
}
```

Listing 6.11: Confusing Implicit Contract Leading to a Type Confusion in `java.-awt.geom.AffineTransform.createTransformedShape`

Unsupported Operation Usage If a method is called that always just throws an exception and that exception causes the calling method to also behave abnormally then we considered the related issues as *Unsupported Operation Usage*. Consider for example the code shown in Listing 6.12. Here, the `extract` method calls the `read` method, which always throws an exception. This results in two issues: First, both methods cannot be called without triggering an exception that disrupts the control flow. Second, the `extract` method calls the `create_input_stream` method returning an `InputStream`, which is afterwards not further used and – in particular – not closed.

```
static CodecFactory extract (org.omg.CORBA.Any a) {
  return read (a.create_input_stream ());
}
static CodecFactory read (InputStream istream){
  throw new org.omg.CORBA.MARSHAL ();
}
```

Listing 6.12: A Method Always Throwing an Exception in `org.omg.IOP.-CodecFactoryHelper`

In such cases the code often seems to be in a state of flux where it is unclear what has happened or will happen.

Unexpected Return Value We found some cases where the name of a method suggests a specific range of return values – in particular if the name starts with `is`. In this case, Java developers generally expect that the set of return values is {`true`, `false`} and write corresponding code. This will, however, directly lead to dead code if the called method always returns the same value. A concrete example is shown in Listing 6.13. Here, the developer calls a function `isUnnecessaryTransform` and, if the result evaluates to `true` sets the used transformer to `null`. However, this code is dead because the called method, which is shown in Listing 6.14, always returns `false`.

```
if (isUnnecessaryTransform(...)) {
  conn.getDestination().setTransform(null);
}
```

Listing 6.13: Dead Code Due to a Bug in the Called Method; found in `com.sun.-media.sound.SoftPerformer`

```
static boolean isUnnecessaryTransform(... transform){
  if(transform == null) return false;
  ... // [the next four tests also just return false]
  return false;
}
```

Listing 6.14: The result is always `false` in `com.sun.media.sound.SoftPerformer`.

As demonstrated by the code in Listing 6.13, these issues are typically related to significant problems in the respective code. The example also indicates that an inter-procedural analysis is required to discover such issues.

6.2. The Approach

As described in the introduction, we are able to identify the issues presented in the previous section using a generic approach that relies on the detection of dead paths. The approach's three main building blocks: the generic detection of dead paths, the underlying analysis and the post-processing of issues to filter irrelevant ones. We present these building blocks in the following sections.

6.2.1. Identifying Infeasible Paths

To identify infeasible paths in code, we first construct a control-flow graph (CFG) for each method. The CFG contains all possible control flow edges that may occur during the execution *if we do not consider any data flows*. In a second step, we perform an abstract interpretation (AI) of the code, during which a second flow graph is implicitly generated and which consists of the edges taken during the AI. We denote this flow graph AIFG in the following. The AIFG is potentially more precise than the CFG because the underlying analysis is context- and data-flow sensitive. Hence, if the AIFG contains fewer edges than the CFG then some paths in the CFG are not possible. Consider for illustration a conditional branch expression (e.g., `if`, `switch`). The CFG of such an expression contains an edge to the first instruction of each branch. On the contrary, the AIFG will not contain edges from the condition instruction to those branches, whose guarding condition cannot be the result of evaluating the condition, according to the AI.

Given a method's AIFG and CFG, we remove the edges of the former from the latter. The source instruction of the first edge of every remaining path in the intersection graph is reported to the developer, as it is directly responsible for an infeasible path. This instruction is the last executed one and is called the *guard instruction*. More precisely, the algorithm to detect infeasible paths is shown in Listing 6.15. Given a `method` (Line 1) and the `result` of its AI (Line 2), the algorithm iterates over each instruction, `instr`, of `method` (Line 4) testing whether it was evaluated (Line 5). If `instr` was executed, we iterate over all possible successors and check whether each expected path was taken (Line 7); if not, `instr` is a guard instruction and a report is generated which describes which path is never taken and why. If `instr` was not executed, it cannot be a guard instruction and the iteration continues with the next instruction.

```
1 val method : Method { val instructions : Array[Instruction] }
2 val result : AIResult { val evaluated : Set[Instruction] }
3 for {
4    instr <- method.instructions
5    if result.wasEvaluated(instr)
6    staticSuccInstr <- instr.successors // static CFG
7    if !result.successorsOf(instr).contains(staticSuccInstr)
8 } yield { /* rank and create error report w.r.t. instr */ }
```

Listing 6.15: Detecting Infeasible Edges

Consider for further illustration the graph shown in Figure 6.1. Each node in the graph represents an instruction in some piece of code, its edges represent the set of control-flows in the CFG. Now, let us assume that – by performing an abstract interpretation of the respective code – we can determine that the instructions 4, 5 and 6 are never executed and, hence, that the control-flow paths [2→**4**→6], [2→**4**→5→6], [3→**4**→6] and [3→**4**→5→6] are infeasible. In this case, the analysis will create one report for the guard instruction 2 and one for guard instruction 3.

To illustrate how the technique reveals issues presented in the previous section, we

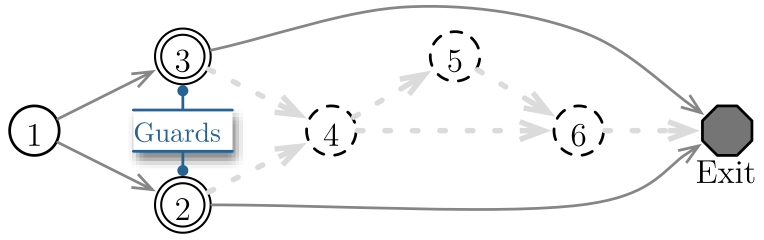
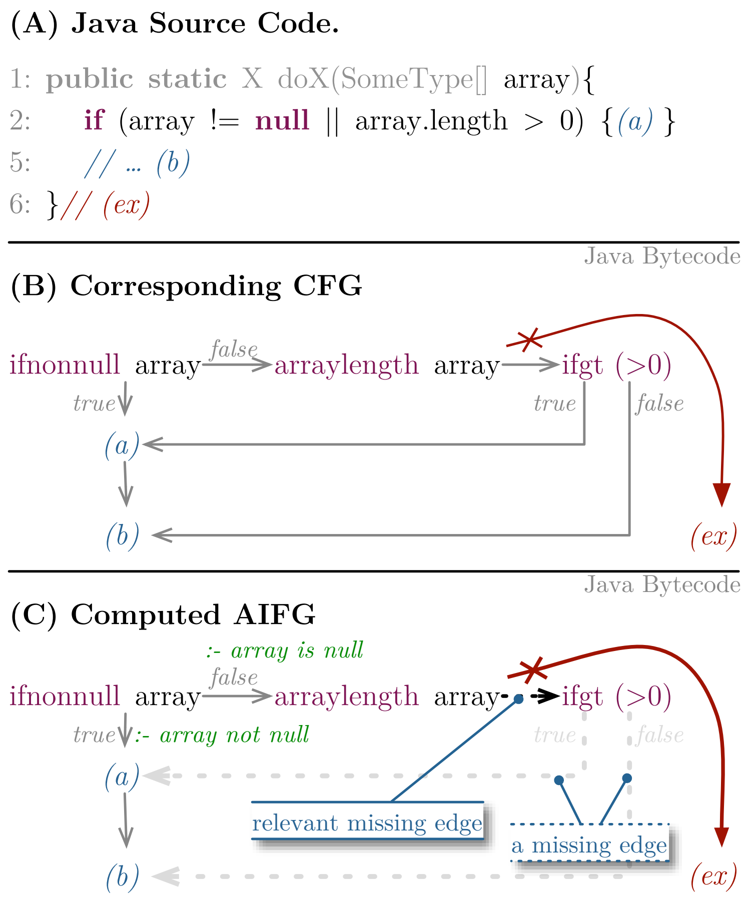


Figure 6.1.: A CFG vs. an AIFG.



```
1: public static X doX(SomeType[] array){
2:     if (array != null || array.length > 0) {(a) }
5:     // ... (b)
6: }// (ex)
```

Figure 6.2.: The Approach Exemplified

apply it to a concrete example. The issue under consideration that we found multiple times in the JDK is the use of the wrong logical operator (e.g., `||`) in combination with a `null` check. The code in Figure 6.2(A) (Line 2) exemplifies this issue. The method `doX` will always fail with a `NullPointerException`, if the given `array` is `null`, because `length` is not defined on `null` values. Given the check of `array` against `null` that precedes the field access `array.length`, we assume that the developer actually intended to use the *logical AND operator* (`&&`) instead of `||` to avoid the runtime exception. This issue is identified by our analysis because the AI determines the nullness property of reference values. When the analysis of the `null` check is performed, a constraint is created (cf. Figure 6.2(C)) that states that `array` is `null` if the test `ifnonnull` fails and vice versa. Now, the AI will – when it evaluates the `arraylength` instruction – always only follow the exception path and will never evaluate the `ifgt` instruction and the instructions dominated by it. The generated report will state that the predecessor instruction of `ifgt`, the `arraylength` instruction, is the root cause of the issue, as it always throws a `NullPointerException`.

6.2.2. The Abstract Interpretation Analysis

The analysis that performs the abstract interpretation is built on top of OPAL [EH14] - an abstract interpretation framework for Java Bytecode implemented in Scala. Central to our idea is that the abstract interpretation does not pursue any specific goal – it simply tries to collect as much information as possible; the identification of infeasible paths is a side effect of doing so. Furthermore, unlike most abstract interpretation based analyses approaches, our analysis is not a whole program analysis. Instead, the analysis treats each method as a potential entry point and makes no special assumptions about the overall state or input parameters. For example, when analyzing a method `m(Object o,int i)`, no assumptions are made about the parameters `o` and `i` (`o` could, e.g., be `null`, an alias for `this`, or reference an independent object). For each method, we perform an inter-procedural, path-, flow-, object- and context-sensitive analysis, however, only up to a pre-configured – typically very small – call chain length.

The rationale for the above design decisions is twofold: (a) to make the approach usable for analyzing libraries, and (b) to make it scalable to industrial sized applications. Libraries do not have a designated `main` method - any public method can be an entry point to the library. Hence, our decision to handle each method in isolation. Our understanding of a very large codebase is, e.g., the entire JDK 8, which consists of more than 190,000 concrete methods with more than 9,000,000 Bytecode instructions. An approach that does not restrict the length of call chains to be followed may find more issues, but does not scale for such codebases as the effort roughly scales exponentially. So, we deliberately trade off the number of issues that we find for scalability. This is a common design decision taken by static analysis tools [CHH^{+}06, BBC^{+}10].

The analysis is designed as a product line, currently customizable with respect to: (a) abstract domains used to represent and perform computations on values, and (b) the maximum call chain length that the analysis follows.

Customizing the Abstract Domains Concerning (a), the current implementation is configured with the following abstract domains.

(i) For computations on integer values we use a standard interval domain based on the description found in [NNH99]. Our domain implements all arithmetical operations supported by the JVM and facilitates path-sensitive analyses. It is further parameterized over the maximum size of the intervals for which we try to compute the result before the respective variable is assumed to take any value (the top value in the abstract domain's lattice). The parameterization over abstract domains enables us to adapt the precision of the analysis and to make it scalable to large code bases. For example, for the method in Listing 6.16 the analysis would compute that the return value is in the range $[200, 1000]$ if the configured maximum cardinality is larger than 800. If the maximum is, e.g., 32 the result would be *AnyInt* (the top value).

```java
int m1(boolean test){
  int i = 100;
  if (test) i *= 10;
  else i *= 2;
  return i; // i is a value in the range [200,1000] or "Any Int"
}
```

Listing 6.16: Test Code for Integer Ranges

We could as well use a domain that represents integer values as sets. In the example above, such a domain could precisely capture the information that the return value is either 200 or 1000. However, in a pre-study, we found that a domain based on ranges is superior in terms of its efficiency and hence better suited for our setting.

(ii) For the other primitive data types supported by the Java Virtual Machine (`float`, `long`, `double`), the current implementation performs all computations at the type level; hence, currently we cannot identify bugs related to these values. This limitation, however, is only a matter of putting more engineering effort, which is currently ongoing.

(iii) The domain for reference values is object-sensitive and distinguishes objects based on their allocation site. Hence, for objects created within an analyzed method, we can derive precise type information. The domain also supports alias and path-sensitive analyses. Currently, this domain cannot be configured any further.

Customizing the Maximum Call Chain Length Let us now consider the second customization dimension (b); adapting the maximum call chain length. At each call site, the analysis uses the configured maximal call chain length to determine whether to make the call. If the length of the current call chain is smaller than the configured maximum, the analysis invokes the called method using the current context - however, only if the information about the receiver type is precise, i.e., the call site is monomorphic. If we only have an upper type bound for the receiver object it could happen that a developer overwrites the respective method later on and would render the analysis unsound.

To illustrate the effect of configuring the call chain length, consider the code in Listing 6.17 and assume that the maximum call chain length is 2 and the analysis tries to find issues in method `m1`. In this case, the method `m2` will also be analyzed with the current context (`i <= 0`) to find out that `m2` will always throw an exception. However, since

```java
1 void m1(int i){
2     if(i <= 0 && m2(i) > 1) System.out.println("success");
3     else System.out.println("failure");
4 }
5 int m2(int i){
6     if(i > 0) sqrt(i);
7     else throw new IllegalArgumentException();
8 }
```

Listing 6.17: Example for the Effects of Call Chain Length

the max call chain length is 2 the constructor call in `m2` (Line 7) will not be analyzed. Hence, the analysis draws no conclusion about the precise exception that is thrown[4].

We complement the abstract interpretation analysis described so far with two simple pre-analyses. Their goal is to determine precise types for values stored in private variables respectively of values returned by private methods. In Java (and other object-oriented) libraries it is common to use interface types for fields and method return values, even if these values are actually instances of specific classes. The pre-analyses that we perform are able to determine the concrete types of private fields and of return values of private methods in 40% of the inspected cases[5].

We limit the application of these analyzes to private methods and fields only to make them usable for libraries. Extending the scope of the analyses beyond private methods/fields may lead to wrong results. A developer, using the library, may create a new subtype which stores other values of different kinds in the field or overrides the method returning a different kind of object. This cannot happen with private fields/methods, both are not accessible in subclasses and, hence, cannot be changed.

Both pre-analyses are very efficient as they only analyze one class at a time and perform abstract interpretations of a class methods using domains that perform all computations at the type level. However, they still contribute significantly to the overall power of the analysis w.r.t. identifying issues in particular if the maximum call chain length is small. If the call chain length is 1 then 10% of the issues are found based on the pre-analysis.

6.2.3. Post-processing Analysis Results

Obviously, it is desirable to minimize the number of false reports produced by the analysis. However, completely avoiding them is due to the complexities associated with statics analyses generally not possible. In our setting, we distinguish between two kinds of false reports. First, there are infeasible paths that are correctly identified as such by the analysis, but which do not hint at issues in the software. They are rather due to (i) code generated by the compiler, (ii) intricacies of Java and (iii) common best practices. We call reports related to such issues *irrelevant reports* to indicate that they are not real false reports in the sense of being wrong. Nevertheless, they would be perceived as such by developers as they would not help in fixing any source code level issues. Second,

[4]The constructor of `llegalArgumentException` may throw some other exception.
[5]When analyzing the JDK 1.8.0 Update 25.

there are paths that are wrongly identified as infeasible. Such paths are due to the imprecision of the analysis w.r.t. code that uses reflection or reflection-like mechanisms. In the following, we discuss each source of false reports and the heuristics used to identify and suppress them.

Compiler Generated Dead Code The primary case that we identified, where Java compilers implicitly generate dead code, is due to the compilation strategy for `finally` blocks. The latter are typically included twice in the Bytecode, for both cases when an exception occurred respectively did not occur. This strategy often results in code that is provably dead, but where there is nothing to fix at the source code level.

```
1  void conditionInFinally(java.io.File f) {
2    boolean completed = false;
3    try {
4      f.canExecute();
5      completed = true;
6    } finally {
7        if (completed) doSomething();
8    }
9  }
```

Listing 6.18: Implicit Dead Code in Finally

For illustration, consider the code in Listing 6.18. The `if` statement (Line 7) gets included twice in the compiled code. If an exception is thrown by the `canExecute` call (Line 4), `completed` will always be `false`. Therefore, the call to `doSomething` would be dead code. However, if no exception is thrown, `completed` will be `true` and, hence, the branch, where completed is `false` is never taken. Now, to identify that there are no issues at the source code level, it is necessary to correlate both Bytecode segments to determine that both branches are taken. In other words, we have to recreate a view that resembles the original source code to determine that there is nothing to fix at the source code level.

To suppress such false warnings, we use the following simple heuristics. We search in the Bytecode for a second `if` instruction that accesses the same local variable and which is in no predecessor/successor relation with the currently considered guard instruction, i.e., both instructions are on independent paths. After that, we check that one of the two `if` instructions strictly belongs to a finally block, i.e., we check that the `if` instruction is dominated by the first instruction of a finally handler, which the other one is not.

The Intricacies of Java In Java, every method must end each of its paths with either a return instruction or by throwing an exception. Now, if a method is called, whose sole purpose is to create and throw an exception (e.g., `doFail` in Listing 6.19), thus intentionally aborts the execution of the calling method (`compute` in Listing 6.19), the calling method still needs to have a `return` or `throw` statement. These statements will obviously never be executed and would be reported without special treatment.

```java
Throwable doFail() { throw new Exception(/* message*/); }

Object compute(Object o) {
   if(o == null) [ return doFail(); OR throw doFail(); ]
   else return o;
}
```

Listing 6.19: `doFail()` Always Throws an Exception

Such infeasible paths, however, are not related to a fixable issue in code and should hence be suppressed. We do not generate a report if (a) the first instruction of an infeasible path is a `return` or `throw` instruction and (b) the guard instruction is a call to a method that always throws an exception.

Established Idioms A source of several irrelevant reports in our study of the JDK is the common best practice in Java programs to throw an exception if an unexpected value is encountered in case of a `switch` statement. The implementations in these cases always contain a default branch that throws an error or exception stating that the respective value is unexpected. A prototypical example is shown in Listing 6.20.

```java
switch (i) {
  case 1 : break;
  // complete enumeration of all cases that should ever occur
  default : throw new UnknownError();// should not happen
}
```

Listing 6.20: Infeasible Default Branch

In the JDK we found multiple instances of such code, which vary significantly. In particular, the exceptions that are thrown vary widely ranging from `UnknownError` over `Exception` and `RuntimeException`, to custom exceptions. Furthermore, in several cases the code is even more complex. In these cases a more meaningful message, which captures the object's state, is first created by calling a helper method that creates it. In some cases that helper method even immediately throws the exception. To handle all cases in a uniform way, we perform a local data- and control-flow analysis that starts with the first instruction of the default branch to determine whether the default branch will always end by throwing the same exception.

Assertions Another source of irrelevant reports related to correctly identified infeasible edges is code related to assertions. We found several cases, where we were able to prove that an `assert` statement will always hold or where an `AssertionError` was explicitly thrown on an infeasible path. In the latter case, the Java code was following the pattern `if( /* condition was proven to be false */ ) throw new AssertionError( /*optional message*/ )`) and is directly comparable to the code that is generated by a Java compiler for `assert` statements. As in the previous case, reports related to such edges are perceived as irrelevant by developers. We suppress related reports by checking whether an infeasible path immediately starts with the creation of an `AssertionError`.

Reflection and Reflection-like Mechanisms Though we tried to reduce the number of false positives to zero, we found some instances of false positives for which the effort of programmatically identifying them would by far outweigh the benefits. In these cases, the respective false positives are due to the usage of Java Reflection and/or the usage of `sun.misc.Unsafe`. Using both approaches it is possible to indirectly set a field's value such that the analysis is not aware of it. For example `java.lang.Thread` uses `Unsafe` to indirectly set the value of the field `runner` using a memory address stored in a long variable. Our analysis in this case cannot identify that the respective field is actually set to a value that is different from `null` and hence, creates an issue related to every test of the respective variable against `null`. As the evaluation will show, the absolute and relative numbers of false positives are so low that the heuristics can still be considered effective.

6.3. Evaluation

We evaluated our analysis by using the approach on the JDK to (i) get a good understanding of the issue categories that our analysis can identify and the effectiveness of the techniques for suppressing false warnings, and (ii) to derive a good understanding of how the maximum call chain length and the maximum cardinality of integer ranges effects the identification of issues as well as the runtime. After that, we applied the approach to the Qualitas Corpus [TAD+10] to test its applicability to a diverse set of applications and to validate our findings.

6.3.1. JDK 1.8.0 Update 25

Issue Categories

The issue categories that we identified were discussed in Section 6.1. The distribution of the issues across different categories is shown in Table 6.1. We manually evaluated all reported issues in the packages that constitute the JDK's public API: `java*`, `org.omg*`, `org.w3c*` and `org.xml*`. The analysis was executed using a maximum call chain length of 3 and setting the maximum cardinality of integer ranges to 32; no upper bound was specified for the analysis time per method.

Overall 556 reports were related to the public API; including the reports that were automatically identified as irrelevant because they are expected to belong to compiler generated dead code, to assertions or to common programming idioms. All 556 reports were manually inspected to check for false positives, to classify the reports and to assess the filtering mechanism. In the results of the analysis we found 19 reports that were related to code for which we did not find the source code and, thus, dropped them from the study.

Category	Percentage
Obviously Useless Code	1%
Confused Conjunctions	2%
Confused Language Semantics	3%
Dead Extensibility	9%
Forgotten Constant	4%
Null Confusion	54%
Range Double Checks	11%
Type Confusion	3%
Unexpected Return Value	5%
Unsupported Operation Usage	7%
False Positives	1%

Table 6.1.: Issues per Category

From the 537 remaining reports 279 ($\approx$ 52%) were automatically classified as irrelevant. A further analysis revealed that 81% of the 279 irrelevant reports are related to compiler generated/forced dead code - a finding that clearly demonstrates the need to suppress warnings for compiler generated dead code. Another 12% are due to assertions or common programming idioms. The remaining 7% were false negatives, i.e. the filtering was too aggressive and suppressed warnings for true issues. All these cases were related to a method `read`[6] that just throws an exception and which was called at the end of the calling methods. Hence, they were automatically classified in the *Intricacies of Java* category. Given that the filter otherwise proved very helpful, we decided to accept these false negatives.

The vast majority of the reported issues – i.e., the issues that were deemed relevant by the analysis – is related to `null` values. A closer investigation revealed that developers are often literally flooding the code with `null` checks that only check what is already guaranteed. However, as already discussed in the introduction we also found numerous cases where field and method accesses were done on `null` values. The second top most category of issues is related to checking that an integer value is in a specific range. As in the previous case this is sometimes useless, but in many cases it identifies situations where it is obvious that the code does not behave as intended.

For each of the other categories we found so few issues that the development of a specially targeted analysis would probably not be worthwhile for other approaches. However, taken together, these categories make up 34% of all issues and given that we identify them for free, these findings are significant.

Finally, we found two false positives ($<$ 1% of all issues) as discussed in the previous section.

[6]Presented in Listing 6.12.

Varying the Analysis Parameters

To determine the sensitivity of the analysis on changed analysis parameters, we ran several instances of the analysis with maximum call chain lengths of 1, 2, 3, 4 and 5 and with a maximum cardinality settings for integer ranges of 2, 4, 8, 16 and 32. If the analysis time of a single method exceeded 10 seconds, the respective analysis was aborted. This time limit was chosen because it enables the vast majority of methods to complete within the timeframe, but it still avoids that the entire analysis is completely dominated by a few methods with extraordinary complexity[7]. The analysis was executed on a 8-Core Mac Pro with 32GB of main memory. Overall, we ran the analysis 25 times.

As expected, increasing the max call chain length or the maximum cardinality also increases the number of identified issues. However, the effectiveness of the analysis in terms of *number of issues* per *unit of time* decreases sharply. As shown in Table 6.2, the number of additional relevant issues that are found if the call chain length increases from 1 to 2 is remarkable ($\approx 50\%$). However, the number of additional issues that are found if the maximum call chain length is increased from 4 to 5 is much less impressive (depending on the maximum cardinality of integer ranges between 1 and 2 additional issues).

Hence, increasing the maximum cardinality of ranges of integer values is initially much less effective than increasing the maximum call chain length. If the maximum call chain length is 1 then increasing the cardinality from 2 to 32 increases the necessary effort more significantly than increasing the call chain length by one, though the latter will lead to the detection of more issues. Nevertheless, certain issues can only be found if we increase the maximum cardinality and at some point increasing the cardinality is the only feasible way to detect further issues. For example, increasing the call chain length from 4 to 5 does not reveal significantly new issues. However, the analysis still revealed new issues when we increased the range's cardinality. Furthermore, if we specify an upper bound of 10 seconds for the analysis of a single method – which includes the time needed to analyze called methods – the number of aborted methods rises significantly and we therefore even miss some issues.

In summary, from an efficiency point-of-view, a maximum call chain length of 2 or 3 and a maximum cardinality of 4 or 8 seems to be the sweet spot for the analysis of the JDK.

We also examined some of the issues (using random sampling) that only show up if we increase the call chain length from one to two to get an initial understanding of such issues. This preliminary analysis reveals that most additional issues were related to a defensive coding style, which can also be seen in the *Null confusion*, *Range Double Checks* or *Confused Language Semantics* categories.

A prototypical example is shown in Listing 6.21. The check of `tmpFile` against `null` (Line 2) is useless as the method `createTempFile` will never return `null`.

```
1 tmpFile = File.createTempFile("tmp","jmx");
```

[7]A particularly complex method can be found in `jdk.internal.org.objectweb.asm.ClassReader`. The method `readCode` is more than 500 lines long and contains several loops that each call multiple methods.

Max Call Chain Length	Max Integer Ranges Cardinality	Issues Relevant	Issues Filtered	Issues Total	Time [s]	Aborted Methods
1	2	690	1157	1847	10	0
1	4	699	1172	1871	11	0
1	8	701	1180	1881	13	0
1	16	702	1182	1884	21	0
1	32	702	1182	1884	27	0
2	2	1078	1248	2326	25	0
2	4	1090	1269	2359	31	0
2	8	1093	1277	2370	43	0
2	16	1094	1279	2373	73	0
2	32	1094	1279	2373	149	1
3	2	1189	1252	2441	60	0
3	4	1201	1273	2474	78	0
3	8	1204	1281	2485	139	0
3	16	1205	1283	2488	289	1
3	32	1205	1283	2488	894	10
4	2	1224	1252	2476	156	0
4	4	1236	1273	2509	205	1
4	8	1237	1281	2518	438	7
4	16	1237	1283	2520	1259	27
4	32	1233	1283	2516	6117	63
5	2	1225	1252	2477	457	4
5	4	1235	1273	2508	990	7
5	8	1239	1281	2520	1566	20
5	16	1234	1283	2517	5482	80
5	32	1233	1283	2516	39274	143

Table 6.2.: Evaluation of the Analysis Parameters Sensitivity

```
if (tmpFile == null) return null; (* return null is dead *)

File createTempFile(String pre, String suf) throws ... {
   return createTempFile(pre, suf, null); }
File createTempFile(String pre, String suf, File dir) throws ... {
   ... File f; ...
   if (!fs.createFileExclusively(f.getPath())) throw ...
   return f; (* The only return statement. *) }
```

Listing 6.21: Defensive Coding in `javax.management.loading.MLet.getTmpDir`

6.3.2. Qualitas Corpus

We ran the analysis twice on all 109 projects of the Qualitas Corpus [TAD$^+$10][8]. The corpus is well suited to evaluate general purpose tools such as the proposed one as it is a curated collection of projects across all major usage areas of Java (Desktop-/Server-side Applications and Libraries). The evaluation is done once using a maximum call chain length of 1 and once using a maximum length of 2. Overall, this study supports our previous findings. As in case of the JDK study, increasing the maximum call chain length from 1 to 2 led to the identification of a significantly higher number of issues. Sometimes even more than twice as many issues are found (e.g., Eclipse, JRuby or Findbugs), which further stresses the importance of context-sensitivity for bug finding tools. Overall, we found more than $11,000$ issues across all projects. Interestingly, we did not find any issues in the projects jUnit, jFin, jGraphT, Trove, sablecc and fit. A close investigation of these projects revealed that they are small, maintained by a small number of developers, and possess good test suites.

6.4. Conclusion

In this chapter, we have presented a generic approach for the detection of a range of different issues that relies on abstract interpretation. The approach makes it possible to adapt the executed analysis to a library's or application's needs and can easily be extended in a generic fashion. As the evaluation has shown, the presented approach is able to find a large number of issues across many categories in mature large Java applications. We have furthermore motivated and discussed the filtering of false positive that are technical artifacts as well as those related to common programming idioms. The presented techniques reduce the number of (perceived) irrelevant reports to nearly none.

[8]Version 20130901.

7. Related Work for Monitoring

In this chapter we present related work that targets monitoring the static or dynamic state of a program. We start with looking at approaches for bug finding in C or C++ code. Going up the stack, we present techniques to find excessive privilege assignment or usage in Java code. We follow this with a presentation of work in the field of taint-flow analysis, which allows to find complex data-flow dependent issues. We conclude this chapter with a review of techniques to find dead paths in programs.

7.1. Bug Finding in Native Code

Techniques for bug finding emerged right after the first programs were written. We will concentrate on methods for bug finding that can be used for C or C++ code

In an empirical study [TC08], Tan and Croft discovered 59 previously unknown bugs in the JDK's native code. For this purpose, they leveraged existing tools and constructed new ones. To scan for common C bug patterns they use Splint [EL02], ITS4 [VBKM00] and Flawfinder (static analysis tools that highlight probable bug patterns). For JNI specific bugs they provide grep-based scripts and scanners implemented in CIL [NMRW02]. The results were manually inspected as all of the analyses provided high false positive rates (97.5-99.8% for the off-the-shelf tools).

In their paper, the authors identified six vulnerability categories from their found bugs. We designed our analyses (cf. Chapter 4) to target unsafe programming practices rather than particular vulnerability patterns, as they can be exhaustively covered, whereas new vulnerability categories are detected frequently. Indeed, since publication of the study by Tan and Croft, various new vulnerability categories have been identified, e.g., *Invalid Array Index*.

Their analyses produce a high number of false positives of over 97%. As a result, a considerable amount of manual effort is needed to detect the actual bugs. In comparison, we present a ranked list of functions by their detected use of unsafe practices. Also, our analysis produces a considerably smaller amount of warnings in the process, even though Tan and Croft only process a subset of the native part of the JCL[1], as their tools did not scale for the complete codebase. While Tan and Croft produce 0.124 warnings per inspected line of code in this subset, we show only 0.002 warnings per inspected line of code for the complete code base. Therefore, fewer cases have to be considered for manual inspection. As we present them in a ranked list the most vulnerable cases can be reviewed first, making the process more efficient.

[1]They only considered approximately 6% of JCL's native code.

Furr and Foster [FF06] directed research on Foreign Function Interfaces (FFI) towards the JNI and introduce a type-inference system for checking the type safety of programs that use the JNI. As class names, field names and method signatures are represented by strings in the JNI, they track the flow of string constants through the program. This allows them to assign type signatures to the JNI functions and to infer them for most user-defined functions. They discovered 155 errors coming from incorrect type usage in a set of programs using the JNI. Kondoh and Onodera [KO08] leverage static typestate analysis to find four distinct error patterns: missing statements for explicit JNI exception handling, mistakes in resource management due to erroneous manual allocation, local references that are not converted to global references when living across multiple method invocations and calls to disallowed JNI functions in critical regions. Lee et al. [LWH$^+$10] demonstrate how dynamic analyses can be synthesized from state machines to detect FFI violations w.r.t. the JNI. They identified three big classes of rules that enforce safety for the JNI: JVM state constraints, type constraints and resource constraints that each can be represented by a type of state machine.

One thing pointed out in Tan and Croft's study are the vulnerabilities that stem from an incorrect use of exceptions in native code. Therefore, Li and Tan developed several static analyses that can detect incorrect exception throwing and handling [LT09, LT11, LT14].

Furthermore, Tan and Morrisett proposed a framework [TM07] that models C code on an abstract level to help Java analyses understand its behavior and make inter-language analysis more precise. They introduce an extension of the Java Virtual Machine Language to model C code and present an automatic specification extractor for C code.

Gang Tan introduced JNI Light (JNIL) [Tan10] which is a formalization of a common model of (a part of) Java and its native side. The Java-side language is a subset of the JVM Language, the native-side language is a RISC-style assembly language augmented with JNI functions. JNIL models a shared heap, but as Java takes a high-level view on the heap as a map from labels to objects, while native code takes a low-level view as a map from addresses to primitive values, a compromise is needed. To show JNIL's utility, the author formalized a static checking system in JNIL which inspects native code for violations of Java type safety and presented a soundness theorem.

In contrast to our approach from Chapter 4, the bug finding techniques presented here detect rather specific patterns of vulnerabilities. We detect more fundamental unsafe practices, thus gaining a broader spectrum of possible vulnerabilities that we are able to find. Our point of view of classifying JNI functions according to their potential threat, thus, makes novel contributions to this area of research. Existing approaches are focused on their particular bug pattern and cannot find issues not falling into this particular category. Our classification not only gives an overview of where and how many potentially security threatening patterns occur, but can hint a software developer towards packages and methods which, when used directly or through a library, pose a greater security risk than others and thus require more thorough input sanitization and control. The list of functions to be inspected can even be narrowed down further when taking the usage context or the particular capabilities provided into account, as we have shown in Chapter 5.

7.2. Excessive Privileges

When inspecting Java code for security flaws, other issue categories than in C or C++ are important. Because the Java platform already provides guarantees such as memory safety (cf. Section 3.3), issues like buffer overflows do not affect Java code. Issues regarding excessive privilege assignment or usage are more important in this context.

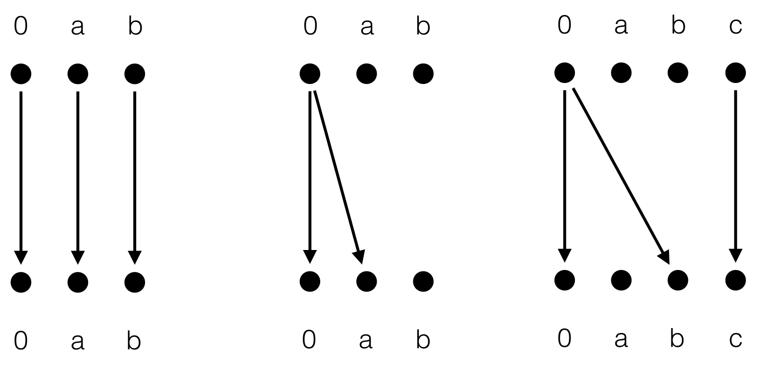
As Java was designed with the idea of loading code dynamically, it is also equipped with an elaborate security mechanism to protect clients from malicious code [Gon11]. It is based on runtime, stack-based access control checks w.r.t. policies provided by the platform, code owners or users. However, writing these policy files has proven to be challenging for developers and thus Koved et al. [KPK02] derived a method (and improved it in [GPT$^+$09]) to infer these access rights so that the resulting policies honor the Principle of Least Privilege [SS75]. Their algorithm traces calls to methods for permission checks in the Java security architecture just as we trace back actual native calls. Hereby, they can effectively create a set of necessary permissions. Nevertheless, they rely on the assumption that every critical resource is indeed effectively protected with a runtime check, as they trace the check and not the resource itself.

The Android platform provides a permission-model for the authorization of apps. When installing an app on an Android-based device, the system explicitly asks the user to confirm a list of permissions granted to the app. Similar to our approach, a coarse-grained permission model is used to represent system resources. Developers have to supply a manifest with their app listing every used permission in form of a list of strings. For example, the string `android.permission.READ_CONTACTS` represents the permission to read the contacts on the device from the app. If the app then calls a platform function related to a permission the Android runtime checks if the permission has indeed been granted to the app.

Similarly to Java policies, it is also hard for developers to write manifests that only include necessary permissions. Accordingly, Bartel et al. [BKLTM12] follow a similar approach to the one provided by Koved et al. With their COPES tool they were able to discover a significant number of apps that suffer from a gap between declared and used permissions.

Intrusion detection works in a similar way to our approach as there system calls are analyzed, which correspond to native calls in our approach. There it has been well established as a useful indicator. Maggi et al. [MMZ08], for instance, use sequence and arguments of system calls with a behavioral Markov model to detect anomalies. Whereas, we want to understand critical resource usage, they go beyond and want to detect abnormal usage patterns.

Of course, developers can use tools like FindBugs [HP04] to determine a footprint of a library, but will only find issues for patterns that have already been included in the bug checker. Other tools like HAVOC-LITE [VL13] or ESC/Java [FLL$^+$02] are able to determine the resource usage of a library, but first have to be configured with the entire set of resources and the methods to reach them.



(a) $id : \lambda S.S$ (b) $\alpha : \lambda S.\{a\}$ (c) $\beta : \lambda S.(S{-}\{a\}) \cup \{b\}$

Figure 7.1.: Modeling Data Flow with the IFDS Framework

7.3. Taint-Flow Analysis

Security vulnerabilities can also stem from complex data-flow problems. A comprehensive, yet complex example are confused-deputy problems (cf. Section 3.4). They consist of an integrity problem, i.e. the security of the operation is jeopardized by a value controlled by an attacker and an optional confidentiality problem, when the vulnerable function returns sensitive information.

To be able to detect such complex problems, analyses must be context- and flow-sensitive, meaning they find data flows over method and field boundaries. An algorithmic framework to formulate such problems is the IFDS framework by Reps et al. [RHS95]. It addresses data-flow problems with distributive functions over finite domains. This implies that the amount of tracked variables in a problem formulation must be finite. So-called flow functions model the generation, propagation and deletion of taint facts as transitions over programming instructions. For instance, an instruction that does not modify the inspected taint facts is depicted in Figure 7.1a. Thus, the currently known facts a and b are propagated to the new state of the program after the instruction. In Figure 7.1b fact a becomes tainted. This is the case, for instance, when the value of a taint source (e.g., user input) is assigned to a local variable a. The taint is generated from the 0 fact, which the tautology fact in the system. In Figure 7.1c, fact a holds before the instruction, but gets deleted, however, the same instruction generates a new taint for fact b. The algorithm evaluates the flow functions over the complete interprocedural control-flow graph (ICFG) of the program and constructs an exploded super graph. A flow between a source and a sink is then expressed as a graph reachability problem from the source fact to the sink node.

Lerch et al. [LHBM14] use the IFDS framework to find confused-deputy vulnerabilities in the Java Class Library (JCL). As confused-deputy problems contain both an integrity and a confidentiality problem, they formulate their analysis as in inside-out analysis starting from calls to caller-sensitive functions. They extended and modified the optimized version of the algorithm [NLR10] to make the algorithm scale to the JCL.

Furthermore, they made adaptations to allow for the coordination of the integrity and confidentiality analysis and for more helpful error reporting.

Arzt et al. [ARF$^+$14] explicitly model the Android lifecycle and use the IFDS algorithm to find leaks of sensitive information like unique identifiers (e.g., IMEI, MAC-addresses), contact data, location or SMS messages. They use a forward and a backward analysis to track different problems and extend it with an on-demand alias analysis. Their FlowDroid tool accurately and efficiently finds flaws with a very low false positive rate.

7.4. Dead Code Detection

Besides the term *dead code* or *dead path*, which we are using in Chapter 6, other closely related terms are also frequently used in related work. For example, Chou et al. [CCK11] use the term *unreachable code*. In their case a fragment of code is unreachable if there is no flow of control into the fragment and is thus never on any path of execution of the program. Their focus is, however, on generating good explanations that facilitate reasoning why the code is dead.

Kasikci et al. [KBC$^+$14] use the term *cold code* to refer to code that is rarely or never executed at runtime. They propose to use a dynamic analysis based on sampling the execution frequencies of instructions to detect such code. As in our case, code that is identified as dead is seen as a threat to software maintenance. In a similar context Eder et al. [EJJ$^+$12] use the term *unused code* and they also point out that unused code constitutes a major threat to the maintainability of the software. As it is common, approaches using dynamic analyses as proposed by Kasikci et al. and our approach which uses static analysis complement each other.

Approaches based on formal methods that rely on SMT solvers are also used to identify variants of dead code. The approach described in [ALS13, AS12, BSS12] for example also determines code that is never executed. They use the term *infeasible code* to describe code that is contained in an infeasible control-flow path. Schäf et al. [SSNW13] take an even wider look at the topic by looking for *Inconsistent Code*, i.e. code that always fails and code that makes conflicting assumptions about the program state. Compared to our approach from Chapter 6 these approaches are trying to prove that the software is free of respective issues. However, they are interested in dead paths of a specific kind while we are interested in dead paths as such and find dead paths related to a variety of issues.

Abstract interpretation [CC77] was already used in the past to detect dead code [CCF$^+$09, PS11]. But, compared to our approach the respective tools try to prove that the code is free of such issues by performing whole-program analyses. The result is that those tools favor precision over scalability and are often not targeting the analysis of libraries. Payet et al. [PS11] for example use the Julia static analyzer to identify – among others – dead code in Android programs.

Though detecting and eliminating code that is dead has been the subject of a lot of research related to compiler techniques [DEMDS00, KRS94], our case study has shown that it is still a prevalent issue in compiled code. Nevertheless, standard techniques that rely on live variable analysis or other data-flow analyses are now an integral part of many modern integrated development environments such as Eclipse or IntelliJ. They help developers to detect some related issues. Compared to these approaches we perform an inter-procedural, context-sensitive analysis based on abstract interpretation which is well beyond the scope of compilers and – as demonstrated – is able to find many more issues.

Other approaches which also try to identify a broader range of issues, such as FindBugs [CHH$^+$06] or JLint [AB01], implement one issue detector for each kind of issue. This limits these approaches to only detect those issues the developer explicitly had in mind which is not the case in our approach. Compared to the presented analysis they can also find issues that are not related to the control-/data-flow.

Part III.

Isolation

Attack Surface Reduction and Sound Privilege Separation

In the previous part of this thesis, we present our contributions regarding monitoring the static state of a program, i.e. its program code. The three analyses we provide consider the full stack of Java programs down to the operating system binding and target very different issues. Developers can use the output of each analysis to harden their system and remove possible vulnerabilities before delivering the software to the end user.

In this part of the thesis, we will extend this work and present concepts for isolation. All of these concepts are directed toward vulnerability remedies and, by this, form isolation mechanisms. Furthermore, all of the concepts follow the language-based security approach as they use program analysis and transformation in order to derive a system that is more secure than the previous.

Isolation for security can be understood in two ways in our case. First, the attack surface of a system can be reduced by removing unnecessary code. The assumption of this method is that by reducing the code base, the number of interfaces to the system is also reduced. As shown before, unnecessary (library) code has been exploited in the past many times. It is, therefore, a direct implication from the Principle of Least Privilege that attack surface reduction should be practiced in any software project prior to delivery.

In Chapter 8, we put forward the idea to reduce third-party code in applications to the necessary minimum. This can either be achieved by inspecting the usage context in the application or the desired capability footprint. We show that usage-context-based slicing can reduce the file size of libraries as well as their capability footprint significantly.

The second way isolation can be understood is by means of privilege separation. In order to follow the Principle of Least Privilege down to the object level, it is necessary to minimize the authority of an object as much as possible. However, it becomes hard to determine the actual authority of an object, when reflection or native calls are used. As we discuss before, a strict adherence of the Object-Capability Model is a clear necessity for a sound inspection of the effectiveness of authority isolation in a program written in an object-oriented language.

In Chapter 9, we take a first step towards full and effective capability safety in Java and present an idea for future work. We start with finding the fundamental violations of the Object-Capability Model of the Java language by systematically inspecting its language specification. We then design a study to find correlations between these violations and recorded or measured vulnerabilities. We find interesting correlations in the first experiments and present ideas for further research on this topic.

8. Library Slicing for Attack Surface Reduction

In this chapter, we explore the idea to reduce the attack surface of applications by slicing the libraries they depend on down to the part that the application actually needs. The reduction can be executed in two different ways – either by leveraging usage information from the application context, or by leveraging a set of target capabilities the application developer deems appropriate for the library.

We begin with a motivation of the addressed problem (Section 8.1) and follow it with a high-level summary of the proposed transformation design in Section 8.2. We present the underlying concept of separation inference in Section 8.3. The technical details of Bytecode emission are discussed in Section 8.4. In Section 8.5 we present the tasks for application developers which are required by our approach. We evaluate the approach in Section 8.6 and show that it can reduce library code by an average of 47% in file size. In this Section we also closely investigate the limitations of the approach w.r.t. reflection. We complete the chapter with a summary of the presented approach in Section 8.7.

8.1. Motivation

As discussed before, library reuse is an integral part of current software engineering. However, the decision to include a library into an application can have a number of consequences unknown to the developer. Most important, the inclusion of library code to the application package can introduce new vulnerabilities to the application. As an example take the vulnerability found in jFin_DateMath (cf. Section 5.3). When using this library in a security critical context, the open class loaders introduced by this library might not be acceptable. The plugin infrastructure in this library that introduced the vulnerability might not be needed in the application's context, however, there is no way to use the library without this infrastructure. The decision to use the library is binary – the developer either includes the library or not.

The library's code is as accessible as any other code in the application and is endowed with the same privileges. Because more functionality is available to an attacker using reflection (or similar mechanisms), the attack surface of the application is expanded with every new library included in the application. In the previous example an attacker could use the open class loader to load arbitrary new code into the current application process.

Libraries can also pose a significant confidentiality risk. As another example take the popular log4j logging library, which provides a wide variety of logging targets ranging from storing log messages in a file, using a syslog server over a network, sending log messages via email, or logging to specific operating system logging facilities. In practice,

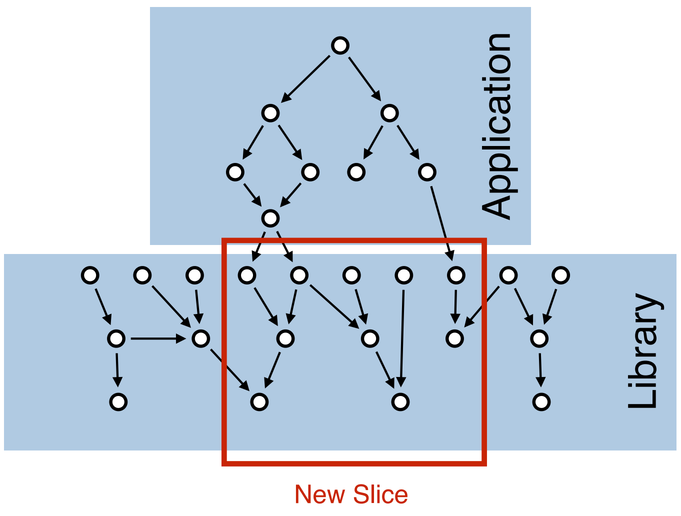


Figure 8.1.: Library Slicing from Application Context

however, most applications use only one of these targets. The other targets still remain reachable by configuration, meaning security-critical data can be sent over the network to unauthorized systems using just a simple configuration change.

Besides the security of the application, the size of the delivered application package is also influenced by used libraries. If the libraries contain unused functionality, code is delivered to the end user which is never used and serves no purpose in this context. As current applications rely heavily on libraries, the amount of library code in applications usually exceeds the amount of application code. For example, the binary JAR file of log4j version 1.2.17 uses approximately 478KB. Current research on Java software [SE13, HDG$^+$11] reveals that most applications are smaller than their combined library code. Furthermore, as libraries may depend on libraries themselves, a reduction techniques must be applicable recursively.

8.2. Concept in a Nutshell

The idea presented in this chapter is to slice the binary version of a library to the parts desired in the application context. The extant slice should contain *all code necessary* to perform the library's tasks in the application. Moreover, it should contain *only the code necessary* for its task in the application context.

We specifically target library users with our approach. Although the technique is also available to library developers, it might not be applicable in their context, as it is hard for them to anticipate all possible usage patterns that emerge in practice. Application developers, on the other hand, have the knowledge on library usage right in the application code and are able to break libraries down to the desired slice using our method.

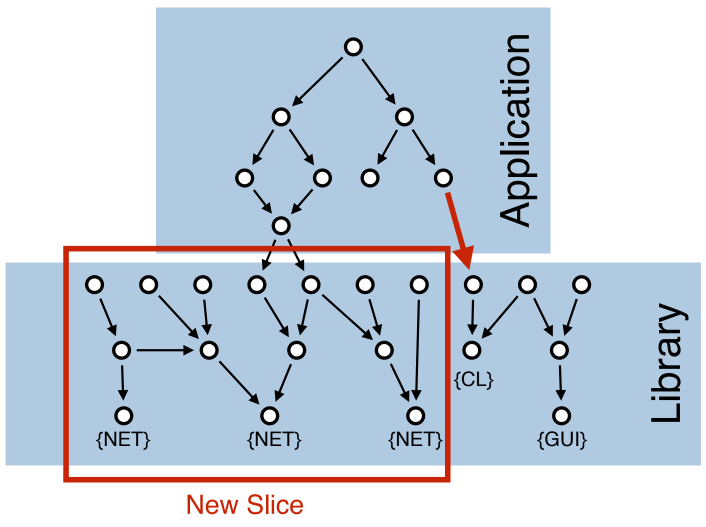


Figure 8.2.: Library Slicing from Capability Context

The first step in creating the new library slice is separation inference which gathers separation information to use in a later slicing step. Information for valid slicing can be obtained using two different sources: (a) The usage context of the library in the application, or (b) a set of desired (or undesired) capabilities for the target slice derived from an analysis as presented in Chapter 5.

In the first scenario the resulting slice only contains the necessary part for the current version of the application (cf. Figure 8.1). First, a call graph of the application and the library is computed. Calls into the library are recognized and followed. Any method not touched by this traversal is not called in the application context and can thus be safely removed. In this scenario, the slice only contains the library code that is necessary to run the application. This also includes methods that use capabilities that might not be allowed in the application's security context.

In the second scenario the resulting slice contains only parts of the library matching a capability set provided by the applications policy (cf. Figure 8.2). The capability footprint of the library is computed with the algorithm presented in Chapter 5. Only those methods of the library are included in the slice, which match the capability set requested from the application. In this scenario, the slice may contain more code than necessary to run the application, however, it might also lack methods that are needed for the application. If this is the case, an application might encounter runtime errors and cannot be compiled against the library slice anymore. The developer has to take action. This is an intented behavior as the usage of some parts of the library in the application is violating the capability policy for the library.

If library size and the capability context are relevant to the application developer, the new library slice can also be built from the intersection of both scenarios. This removes any excess method, but also honors the capability policy of the application.

The final step in the procedure is to emit the Bytecode of the extant library part to a new library file. This library only contains methods (and their enclosing classes), that are included in the separation information computed before. The application is only shipped with code which will eventually be used or which is allowed in the capability policy.

8.3. Separation Inference

The first step in producing a viable library slice is to gather information on the desired subset. In contrast to regular slicing approaches, we are slicing at the method level and leave code inside methods intact. We plan to inspect in future work, whether slicing method blocks would benefit the approach significantly, however, we assume that most Java libraries are decomposed sufficiently at the method level already. Here, we suggest two different approaches to method-level slicing: Using the application context or using a capability policy describing allowed capabilities for the library.

In both cases the challenge is to compute a most precise call graph for the application and the library. Imprecision stemming from incomplete receiver type information may lead to separation information that includes code in the extant library part that is never used during runtime. Any unsoundness in the separation inference will, in contrast, lead to a library slice that is not sufficient to run the application because functionality used at runtime is missing.

8.3.1. Separation by Application Context

Library reuse works in two ways: (1) Black-box reuse, where applications use classes and methods defined in the library, or (2) white-box reuse, where applications extend types from the library. Both methods of library usage have to be taken into account when producing the separation information for the slicing step.

First, a call graph of the application and all of its dependencies is constructed. Any method or class defined in the library that is included in the application-specific subgraph is then added to the set of preserved entities. Constants and field definitions are preserved for each class where at least one method is preserved. Also any used annotation class from the library is preserved. This step ensures the inclusion of any dependencies for black-box reuse.

In white-box reuse application developers inherit from types in the library in order to use their implementations instead of the default ones shipped with the library. Therefore, any supertype of any class added in the previous step or of any class in the application is added, if it is defined in the library. This step ensures the inclusion of any dependencies for white-box reuse.

The precision of this step is depending on the precision of the used call-graph algorithm. Its key element is the resolution of receiver type information for method calls. Therefore, we apply the same pre-analysis steps described in Section 5.2.3 to compute more precise field- and method-return-types.

The output of this step is a set of entities that should be preserved in the extant slice of the library, which captures the usage context of the library in the application. It is a conservative overapproximation of the actually needed part of the library. This ensures the proper execution of the application with the sliced library.

8.3.2. Separation by Capability Policy

In some cases the isolation from unnecessary code in libraries might not be enough. Libraries may still exceed the authority the application developer intends to endow them with and use system capabilities that the developer might not agree with. For instance, a library that performs mathematical operations might store the results of these calculations in the filesystem. In the particular context of some application this might not be permissable. At least, the developer would not expect such a behavior.

So two sources of capability policy emerge: (1) The security context of the application, or (2) the expectations towards a library that an application developer has. When these policies are expressed, they can be used to guide library slicing.

As in the previous case, the call graph of the application and the library is computed first. Furthermore, we compute the capability footprint of the library with the algorithm presented in Chapter 5. We only include those classes and methods in the slicing information, which conform to the defined capability policy.

However, this will result in a slice that might not include everything the application currently depends upon. Runtime binding will fail and the slice will be unusable to compile the application. This behavior is intended, as the removed classes or methods violate the defined capability policy.

The new slice will also include library code, which is not used in the application, but fulfills the capability policy. In order to receive a slice, that represents the usage context and the capability policy, the sets of preserved entities of both approaches just have to be intersected. The slice from this intersected separation information will only include the parts of the library that have been used in the application, but the slice might also be insufficient to run the application, if parts of the library are used, which do not conform to the defined capability policy.

8.4. Bytecode Emission

The final step in the slicing process is the construction of the extant library containing only the Bytecode described in the set of preserved entities. For this step, a new JAR file is composed of all unmodified class files from the library that are included in the separation information. All classes where methods were removed are modified in-memory and then emitted as a new class file. As these transformation happen on the Bytecode level, no recompilation is necessary. A new `META-INF` directory is created including a new manifest with a reference to the ancestry of the slice. The new slice is then packaged as a JAR-file and handed to the application developer.

As the modification of the original JAR-file voids any digital signature it might possess, the original signature must be removed from the JAR-file or in case of a GPG signature the signature file itself. We suggest to resign the new JAR with a new key and to include verifiable information on the ancestry of the library slice. While technically we could preserve the SHA-256 digests of the unmodified classes in the classic `jarsigner` model, we rather move them to the ancestry manifest and create a new signature to make the distinction clear. Using this new signature and the ancestry information, the identity of the library can be verified by end users in order to establish trust.

Another sensitive subject when slicing libraries is the matter of licensing. Third-party libraries might be licensed very diversely. In contrast to commercial licences, that generally forbid the modification of the library, some ”free” licences permit the application developer to alter the library code. While licence models such as Apache License, BSD License, GNU GPL and GNU LGPL allow the free modification and distribution of library code, the Eclipse Public License requires the source code of the modification to be public. Whether the inclusion of ancestry information in the modified binary library fulfills this requirements remains open and a matter of further discussion. However, the careful review of the licence of the library remains an obligation of the application developer.

8.5. Implications for Application Developers

Using our approach, application developers can reduce the amount of unused third-party code. This can either happen at the time, library dependencies are included into the application or at build time. The former case is only applicable in the presence of a capability policy, because the library is not yet used in the application. In this case, our approach becomes a powerful tool for system architects to enforce policies for libraries in a project. The latter case supports both inference sources. It is especially useful if there is no general capability policy or no expectation towards the library yet. However, the application developer can be sure, that she only ships the necessary part of the library she uses.

In order to assess, if the resulting library slices still fulfill the applications requirements, an application needs to provide an effective test suite with a representative coverage. Even though an automated test suite is generally considered a best practice in software development, not every application will fulfill this requirement. We assume that such a test suite exists and encourage developers to create them.

8.6. Evaluation

The goal of our approach is to reduce the amount of unused library code in order to prevent attacks using this code. Therefore, we pose the following research questions:

RQ1 (Soundness) Do the libraries sliced with our application-context approach lead to faulty runtime behavior in the applications?

RQ2 (Precision) How much library code can be removed from libraries in a common application context?

(a) in kilobytes

(b) in number of methods

(c) in number of classes

RQ3 (Precision) How many capabilities are removed from the library with the unused methods?

In order to answer these research questions we have built a set of benchmark application for 60 of the projects used in the evaluation of Chapter 5. The remaining 10 project were either used as platforms or runtime engines (e.g., aspectj) or were mainly designed as applications (e.g., jedit, pooka, and sandmark). We constructed the benchmark applications from the publicly available examples given for the library. If more than one example was provided, we randomly chose one of them. Each application can be successfully executed against the untreated version of the library it depends upon. Furthermore, each example is simple enough to validate a successful execution manually.

Using these applications as the reference, we applied our context-based slicing approach to all 60 libraries and re-ran the applications against the extant slices. Furthermore, we measured the file sizes and the number of classes and methods of the original and extant libraries using OPAL. We also measured the capability footprint before and after treatment using our capability footprint analysis from Chapter 5.

RQ1: Sufficiency of the Sliced Library Part Of all the 60 example applications, only three applications experienced runtime failures. These were the applications for jFin_DateMath, log4j, and sf4j. These failures were in all cases due to removed classes or methods that were accessed using reflection in library code. Our approach was not able to detect these calls and therefore falsely concluced that the missing classes or methods were not used. When inspecting the source code of the three libraries, we found that in the case of jFin_DateMath the reflective call can be easily exchanged with a static call. In case of log4j and sf4j, however, the use of reflection is an integral part of the design of the library.

We can conclude that our approach works sound in the absence of the use of reflection in the library code.

RQ2: Reduction in File Size and Program Elements Using the context of the application benchmark our approach reduces the file size of the libraries in the experiment by an average of 47%. As the detailed results presented in Table 8.1 show, the achieved reduction varies widely depending on the usage in the application and the coupling of the elements inside the library. However, the results clearly show that in every case a reduction can be achieved, which conversely means that there was code in the original library that was not used by the application.

While the number of classes was only reduced by an average of 32%, the number of methods could be reduced by an average of 50%. Also the detailed results shown in Table 8.2 indicate that in some cases (e.g., lucene-join, axion) the number of classes can only be reduced very little or not at all, while the number of methods can be reduced significantly almost always.

RQ3: Removal of Capabilities Table 8.3 shows the capabilities that were effectively removed from the library during the slicing process. This means that the library code using these capabilities was sliced away leaving a library with a smaller capability footprint as before. In our motivating example of the jFin_DateMath library (cf. Section 5.3), the use of the `REFLECTION` capability was removed from the library. The `CLASSLOADING` capability could not be removed as the library code used by the application makes use of it.

While in some cases the footprint could not be reduced (e.g. for axion, log4j and sf4j), other cases show a significant reduction of used capabilities (e.g. jFin_DateMath, checkstyle, lucene-highlighter, maven-core). The results also show that in some cases the use of critical capabilities such as `UNSAFE`, `CLASSLOADING`, or `REFLECTION` can be removed from library code. This means that vulnerabilities which might have been in the original library code using these capabilities were removed as well, automatically.

8.6.1. Limitations

The call-graph algorithms we use cannot compute call edges stemming from reflection or dynamic invocation and cannot detect any calls made from native code (i.e., call-backs), as it is currently state-of-the-art for static analyses for Java. We assume the application code to be free of reflective or native calls to the library. If this assumption does not hold, we cannot guarantee a running application using the slice, because the slice would not represent a sound overapproximation of the library usage in the application's context. We can, however, detect the use of reflection and native calls in the application or library code and warn the developer about this issue.

However, as reflection is a very common mechanism in library configuration (e.g., log4j configuration files), we suggest steps to include the information from other sources. In the following, we describe two approaches to mitigate this issue.

When using reflection to configure libraries, two patterns can be regularly found for object creation or method invocation: (a) The use of string constants in calls to reflection facilities, or (b) the use of configuration files containing these strings. As classes have to be uniquely identified, the use of fully-qualified class names (or a direct mapping to them) can be safely assumed.

We suggest to use a string analysis to extract these class names from Java Bytecode or from configuration files. While configuration files are simply parsed for strings that match the criteria for fully-qualified class names, we suggest to use abstract interpretation for the extraction of class names from Bytecode, as simple operations (e.g., concatenation) can be processed by the analysis. This way, we can extract class names even if they are constructed from multiple constants. For both analyses, classes which can be identified with the extracted fully-qualified class name are added to the set of preserved entities, if they are found in the library. As we cannot infer the called methods with this approach, we add all methods of the class to the set.

Library	Size Before [byte]	Size After [byte]	Reduction
antlr	1543365	735428	52%
axion	433896	391004	10%
cglib	283080	230381	19%
checkstyle	1008473	51297	95%
collections	610259	473494	22%
commons-beanutils	233857	195496	16%
commons-cli	41123	25326	38%
commons-codec	284184	108187	62%
commons-configuration	362679	269622	26%
commons-fileupload	69002	53677	22%
commons-io	185140	81821	56%
commons-lang	412740	164468	60%
commons-logging	61829	43019	30%
easymock	126646	66695	47%
gson	210856	195376	7%
guava	2256213	1808529	20%
guice	642741	582112	9%
gwt	32024115	3143100	90%
jFin_DateMath	114267	38194	67%
jackson-annotations	39817	16517	59%
jackson-core	229860	189830	17%
jackson-databind	1138921	1011345	11%
jasml	116429	44303	62%
java-hamcrest	112929	95702	15%
javassist	723078	461721	36%
javax-mail	571104	418803	27%
jgroups	2311136	1886461	18%
joda-time	589289	332856	44%
jrat	2123190	316277	85%
jsoup	300844	250329	17%
junit	314932	42512	87%
log4j	489883	261539	47%
lucene-analyzers-common	1699874	1404812	17%
lucene-analyzers-icu	84364	28194	67%
lucene-analyzers-kuromoji	4596222	70548	98%
lucene-analyzers-morfologik	20371	6589	68%
lucene-analyzers-smartcn	3602586	35898	99%
lucene-analyzers-stempel	517090	15434	97%
lucene-classification	23790	4105	83%
lucene-codecs	476943	447057	6%
lucene-core	2562626	2339367	9%
lucene-demo	50700	7130	86%
lucene-facet	177221	114521	35%
lucene-grouping	107782	83033	23%
lucene-highlighter	138281	40048	71%
lucene-join	64019	50786	21%
lucene-memory	36076	24022	33%
lucene-misc	97172	62586	36%
lucene-queries	213026	184439	13%
lucene-queryparser	391514	135392	65%
lucene-spatial	126792	90841	28%
lucene-suggest	179007	72641	59%
maven-core	608173	67855	89%
maven-plugin	46108	10681	77%
mockito	1500011	493610	67%
slf4j	32121	15011	53%
sunflow	891444	158986	82%
testng	836806	110064	87%
weka	6555904	3132108	52%
xstream	531571	473181	11%
total: 60			∅ 47%

Table 8.1.: File Size Reduction

Library	**#C before**	**#C after**	**Reduction**	**#M before**	**#M after**	**Reduction**
antlr	937	453	52%	6706	2496	63%
axion	261	251	4%	3108	2186	30%
cglib	229	173	24%	1355	856	37%
checkstyle	477	50	90%	5872	96	98%
collections	431	372	14%	3861	2115	45%
commons-beanutils	137	127	7%	1332	891	33%
commons-cli	22	18	18%	211	110	48%
commons-codec	92	68	26%	718	369	49%
commons-configuration	194	152	22%	2208	1300	41%
commons-fileupload	49	45	8%	299	204	32%
commons-io	110	69	37%	1188	189	84%
commons-lang	217	109	50%	2911	700	76%
commons-logging	28	26	7%	337	209	38%
easymock	95	64	33%	765	137	82%
gson	165	152	8%	949	744	22%
guava	1690	1432	15%	12987	8692	33%
guice	475	420	12%	3100	2494	20%
gwt	5497	2448	55%	46599	18783	60%
jFin_DateMath	62	30	52%	628	127	80%
jackson-annotations	52	17	67%	154	75	51%
jackson-core	85	69	19%	1597	1094	31%
jackson-databind	570	506	11%	6465	4581	29%
jasml	50	38	24%	265	78	71%
java-hamcrest	101	93	8%	719	362	50%
javassist	398	275	31%	3848	2172	44%
javax-mail	319	261	18%	2803	1508	46%
jgroups	1020	894	12%	9789	6584	33%
joda-time	245	220	10%	4422	2212	50%
jrat	1222	250	80%	8873	974	89%
jsoup	227	206	9%	1400	944	33%
junit	286	35	88%	1627	114	93%
log4j	314	184	41%	2358	816	65%
lucene-analyzers-common	526	372	29%	3245	1956	40%
lucene-analyzers-icu	28	23	18%	154	95	38%
lucene-analyzers-kuromoji	49	40	18%	271	217	20%
lucene-analyzers-morfologik	5	4	20%	32	30	6%
lucene-analyzers-smartcn	23	20	13%	119	108	9%
lucene-analyzers-stempel	20	11	45%	101	41	59%
lucene-classification	6	3	50%	31	7	77%
lucene-codecs	255	236	7%	1656	1540	7%
lucene-core	1631	1510	7%	10964	9162	16%
lucene-demo	12	4	67%	64	7	89%
lucene-facet	105	80	24%	613	315	49%
lucene-grouping	62	51	18%	272	211	22%
lucene-highlighter	81	36	56%	491	121	75%
lucene-join	35	35	0%	182	162	11%
lucene-memory	14	14	0%	135	123	9%
lucene-misc	56	47	16%	346	250	28%
lucene-queries	162	153	6%	977	681	30%
lucene-queryparser	248	94	62%	1514	388	74%
lucene-spatial	74	65	12%	436	287	34%
lucene-suggest	88	45	49%	581	108	81%
maven-core	372	48	87%	2840	144	95%
maven-plugin	25	9	64%	298	32	89%
mockito	647	353	45%	3750	1893	50%
slf4j	28	13	54%	332	78	77%
sunflow	255	113	56%	1811	495	73%
testng	546	69	87%	4059	221	95%
weka	2138	1103	48%	21255	9421	56%
xstream	437	388	11%	2664	1969	26%
total: 60			∅ 32%			∅ 50%

Table 8.2.: Element Reduction

Library	**Removed Capability Usages**
antlr	NT, PR, GU
axion	∅
cglib	FS
checkstyle	CL, UN, FS, SY, GU
collections	FS
commons-beanutils	FS
commons-cli	CL, UN
commons-codec	CL, UN
commons-configuration	∅
commons-fileupload	RF, SY
commons-io	CL, RF, SY
commons-lang	CL, FS
commons-logging	∅
easymock	∅
gson	∅
guava	∅
guice	∅
gwt	NT, FS
jFin_DateMath	RF, FS, SY
jackson-annotations	∅
jackson-core	CL, RF
jackson-databind	∅
jasml	UN
java-hamcrest	FS
javassist	UN
javax-mail	∅
jgroups	PR
joda-time	∅
jrat	CB, PR
jsoup	RF
junit	CL, FS
log4j	∅
lucene-analyzers-common	FS
lucene-analyzers-icu	∅
lucene-analyzers-kuromoji	SC
lucene-analyzers-morfologik	∅
lucene-analyzers-smartcn	∅
lucene-analyzers-stempel	SC
lucene-classification	∅
lucene-codecs	∅
lucene-core	NT
lucene-demo	CL, UN, FS, SY, SC, NA
lucene-facet	FS, SC
lucene-grouping	∅
lucene-highlighter	RF, NT, UN, FS, SY, SC, NA
lucene-join	∅
lucene-memory	∅
lucene-misc	RF
lucene-queries	∅
lucene-queryparser	RF, FS
lucene-spatial	CL, UN, FS, SC, NA
lucene-suggest	UN
maven-core	CL, RF, UN, FS, SY, SC, NA
maven-plugin	CL, UN, FS, SC
mockito	SY
slf4j	∅
sunflow	CL, RF, UN, SY
testng	NT
weka	CB, NT, PR
xstream	∅

Table 8.3.: Removed Capability Usages

Another source for reflection usage information is runtime monitoring. If the application uses reflection facilities to retrieve instances from classes or to call methods located in the library, we can monitor these calls and add the respective methods to the set of methods to be preserved in the slice. Ideally, the application has a set of representative tests that can be run automatically. If the coverage of the tests is high enough to perform any library usage, the runtime monitor will eventually collect all used classes and methods. This approach is very similar to the Play-out Agent from TamiFlex [BSS^{+}11]. The collected methods from the library, that were used during test execution, are added to the set of preserved entities. After this step, the set of preserved entities should contain any class and method necessary to run the application.

While the first approach uses purely static information from configurarion files or program code, the second approach uses purely dynamic information collected from application or test runs. In the first approach, class or method names constructed during runtime cannot be collected, but might detect names that are not part of every execution of the program and can be missed by the second approach. To provide a conservative, yet safe, overapproximation of the necessary classes and methods for a sound slice w.r.t. reflection we, therefore, suggest to use both approaches and construct a union set of results.

8.6.2. Threats to Validity

As the slicing in the context-based setting is based on the usage of some library in an application, the extent of usage inside the application is key to the effectiveness of the slicing. In our study we use a set of applications constructed from examples given by the original library authors which showcase the use of a particular feature or scenario for the library. This means that the extent of usage of libraries in real-world application might be different than in our set of benchmark applications. However, as we can safely assume both cases (a lesser or a more intensive use) in real-world scenarios, the effect might not be significant.

Also the set of removed capability usages is highly dependent on the usage scenario of the library. The applications used for this study might not reflect common usage scenarios for the respective libraries in practice.

8.7. Summary

In this chapter, we presented a concept for library slicing based on application usage-context and on capability policy. We presented these two approaches in detail and discussed the effects of over- and under-approximation on the resulting library slice. Library slicing requires the developer of an application to develop a representative test suite to cover all execution cases. While this is a good practice in general for maintaining correct application behavior despite change, it can also be an efficient check whether the library slice contains all the necessary original library code. We have shown that in a library showcase scenario the approach can reduce libraries by a mean of 47% in file size while removing critical capability usages if they are not on the possible execution path. The concept of library slicing can aid developers to mitigate the risk of shipping unnecessary code that might be exploited and helps them actively reducing the attack surface of their application.

9. Future Work: Towards Full Object-Capability Safety in Java

In order to perform operations, which have a measurable effect, a part of a program needs two things: (1) control (i.e., the current program counter is set to an instruction in the part), and (2) authority (i.e., the privileges necessary to perform the operation). While control is only transferred explicitly through calls, branches, and throw operations, authority can also be transferred implicitly using global state (so-called *ambient authority*). Reasoning over isolation in the presence of ambient authority is hard bordering to undecidable.

As Miller et al. [MYSI03] wrote:

> The question of ambient authority determines whether subjects can identify the authorities they are using. If subjects cannot identify the authorities they are using, then they cannot associate with each authority the purpose for which it is to be used. Without such knowledge, a subject cannot safely use an authority on another party's behalf.

Applied to a static assessment of authority isolation in a program, this means that given a specific program point an analysis will not be able to enumerate the authority an object possesses at this point. This can lead to excessive and implicit authority transfer between objects and can foster vulnerabilities such as the Confused Deputy (cf. Section 3.4).

We advocate to make authority transfer as explicit as possible. In order to achieve this in Java, a strict application of the Object-Capability Model [MS03, Mil06] establishes rules, which disallow implicit authority transfer. In the Object-Capability Model authority can only be obtained via four mechanisms: (1) By an initial state of the system, (2) by parenthood (i.e., creating another object), (3) by endowment (i.e., passing a capability via a parameter during creation of another object), or (4) by introduction (i.e., passing a capability via a method call).

Java respects the Object-Capability Model to some extent, but violates it in some language features. We systematically examine the Java language specification [GJSB09] for these violations and derive a comprehensive list. As instances of these violations can be found in almost every Java project including the Java Class Library (JCL), most Java code is not object-capability safe. However, the amount of existing code is so large that a cold start from zero would forbiddingly set back the current state of software engineering. Therefore, we conclude that migration technologies towards more secure software engineering methods are necessary.

As a first step towards this migration, we implemented an analysis to detect instances of *Object-Capability-Safety Violations* (OCaps Violations) in Java Bytecode. It reports all direct instances of violations, which are: (1) The use of intrusive reflection (cf. Section 3.3.2), (2) the use of static mutable state, and (3) the use of native methods. We also implemented the ability for the analysis to compute the inverse transitive hull in the call graph in order to find transitive effects of these violations. As Java programs need to cross the boundary to the native implementation of the JCL in order to achieve any effect on the system, transitive native calls are unavoidable. Therefore, the transitive analysis should be understood more as a means to study the extent of possible vulnerabilities.

The Object-Capability Model has been covered to some length in scientific literature. Many aspects of the model have been inspected and we show the current state of research in Chapter 10. In this chapter, we present the design for a study on the Object-Capability Model with the goal of discovering possible correlations between OCaps violations and possible or known vulnerabilities. We conducted an exploratory experiment based on this study design and find interestingly high correlations. However, the scope of this first experiment is limited and, therefore, the findings can only provide motivation for further research.

We also take a first look at possible program transformations in order to remove capability violations from Java programs. Furthermore, we take a look at the entailing maintainability problem when making every authority transfer in the program explicit.

The chapter is structured as follows. We first describe our inspection of the Java Language Specification w.r.t. OCaps violations in Section 9.1 and present the resulting analysis in Section 9.2. In Section 9.3, we explain our study design and our preliminary experiment to motivate this study. We briefly take a look at possible program transformations to foster explicit authority transfer in Section 9.4 and summarize the chapter in Section 9.5.

9.1. Java's Capability Safety

At first sight, Java fulfills the requirements of the Object-Capability Model. References in Java are unforgeable from programs written in the Java language or any other language compiled to Java Bytecode. However, when the control flow of the program leaves the fenced garden of the JVM's heap control (i.e., native methods are being called) this guarantee can not be given anymore. This is also important for the second requirement of the Object-Capability Model: effective private state encapsulation. Programs written in Java generally conform to this requirement, however, not only with direct memory manipulations through native methods but also through reflection violating low-level access control (cf. Section 3.3.2) encapsulation can be broken.

The third requirement for object-capability safety is mandated by four rules of authority transfer in the model. Globally shared state is forbidden in an object-capability safe language, as it creates ambient authority if objects can subvert the four transfer rules.

We systematically reviewed the Java Language Specification (JLS) [GJSB09] and the Java Virtual Machine Specification [LYBB14] to find violations of the requirements and rules of the Object-Capability Model. We found three fundamental violations, which we explain in the following.

9.1.1. Native Method Calls

Java allows the definition of methods that defer their implementation to code written in another language (§8.4.3.4 in the JLS). The Java Class Library makes heavy use of this language feature. It declares over 4,000 `native` methods, which bind Java applications to operating system facilities, perform mathematical operations, or implement reflection. Native implementations are not bound to the restrictions the virtual machine puts in place to ensure unforgeable references and encapsulation. The complete heap can be read and modified. Object references can be constructed.

9.1.2. Shared Mutable State

Java allows the definition of static fields of any type or visibility in a class (§8.3.1.1 in the JLS). If static fields have the visibility modifier `public`, they can be read and written from anywhere in the program. The field has become an ambient resource in the program. Even if the field is marked with the `final` keyword and cannot be overwritten with another object it might still be an ambient resource, as its type might be mutable. For instance, a linked list shared this way allows the concatenation of new elements, which can be read by different parts of the program.

Its visibility modifier only limits the scope of the availability of the ambient resource. Even when set to `private` different objects of the same class can still collude using the field. Therefore, we consider a field an ambient resource, if it is declared `static` and is either not `final` or declared using mutable reference type.

9.1.3. Intrusive Reflection

Reflection is not defined in the Java Language Specification. It is handled as a part of the Java Class Library in the package `java.lang.reflect` and is used for retrieving reflective information on classes and objects. Additionally, package `java.lang.invoke` is available for implementing dynamic invocation. While the former package has been available in Java since version 1.1, the latter was added in version 1.7 for the support of dynamic languages in the JVM. Both APIs allow the subversion of the visiblity access control model of Java. Once a reflective object was retrieved it is possible to use the `setAccessible` method defined in `java.lang.reflect.AccessibleObject` to deactivate the runtime checks for access modifiers inside the reflection handling code. Subsequent read, write, and invoke operations are not constrained by the visibility declared at the class, field or method definition. This breaks the encapsulation requirement of the Object-Capability Model and we, therefore, consider any instance of this practice as a violation.

9.2. Detecting Violations

We implemented a static analysis based on Soot [VRCG+10] to find these violations in Java Bytecode. It is designed as a whole-program analysis taking any public method as an entry point.

The detection of native calls is the most straightforward of the three analysis modules. We construct a call graph and report any caller of any native method.

For the detection of shared mutable state, we iterate over any class in the analysis set and check the mutability of any static field declared in the class. We report all non-`final` fields, as they are mutable. For all fields declared `final`, we inspect the declared type of the field. We consider `final` fields of a primitive type as immutable. Furthermore, we consider any `final` field as immutable, if the type is either a known immutable type[1] or composed of known immutable types and all methods of this type are free of side effects. We report any static field that by this definition is either non-`final` or uses a mutable type.

Our analysis module for intrusive reflection is applied to any method where a call to the `setAccessible` method defined in `java.lang.reflect.AccessibleObject` is found. This call is necessary to deactivate the runtime checks for visibility access control. We then perform a simple flow analysis to determine two things: (1) Whether the object, which was set accessible, depends on a parameter value of that method, and (2) for private methods, whether a flow can be constructed from a public method to the parameter value of the private method. Using these criteria we only address objects which are controllable from a caller and not those created internally. We, thus, filter out any use of intrusive reflection which depends on constant values (e.g., class name constants) and which can effortlessly be transformed into a change in the design of the program.

We also developed the ability to propagate the results of the three analysis modules backwards along call edges in the call graph's inverse transitive hull of each reported method. This is especially interesting for native calls, because the strong dependency of the Java platform towards its native implementation can be observed this way. For example, we found that over 66% of the methods in the `rt.jar` file of the JCL[2] transitively use native functions.

9.3. A Study on Object-Capability-Safety Violations in Java Code

The ruleset of the Object-Capability Model promises to foster programs, where authority transfer is made explicit and, therefore, traceable. Together with tools for static anal-

[1] `java.lang.String`, `java.lang.Character`, `java.lang.Byte`, `java.lang.Short`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Float`, `java.lang.Double`, `java.lang.Boolean`, `java.math.BigInteger`, `java.math.BigDecimal`, `java.lang.StackTraceElement`, and all primitive types

[2] Oracle Java version 1.7.0.21

ysis and rigerous code review the adherence to these rules should produce more secure software than current state-of-the-art. Conversely, we assume that vulnerabilities have their root cause in violations to these rules.

In order to validate this assumption we designed a study to find correlations between vulnerabilities and violations. While the data for violations is generated by our own analysis, the data for vulnerabilities can be retrieved from multiple sources such as vulnerability databases and static checking tools. If a strong correlation between vulnerabilities and violations of the Object-Capability Model exists, further studies on the causility of this conjunction can be devised.

To ensure a sufficiently large experiment set, we use the Qualitas Corpus [TAD$^+$10] in the complete version 20130901. It consists of 111 Java libraries in different versions (752 in total), 39,846 Java archive files, and 5,886,775 classes. Additionally, we created a smaller corpus consisting of the JCL (version 1.2.2 update 6 until version 1.7.0 update 65 – 151 in total), the Apache Struts framework (version 1.1 to 2.3.16.3 – 48 in total), and the Tomcat application server (version 3.0.0 to 8.0.9 – 199 in total). This corpus consists of 12,281 Java archives and 3,188,372 classes. We created the second corpus as for the projects included a sufficiently large dataset of known vulnerabilities can be obtained, while for the first corpus this information is not available.

Datasets of known vulnerabilities (e.g., the National Vulnerability Database (NVD)) document the insecurity of a system over the course of multiple versions. As version information was not consistently given in the machine readable part of the NVD, we implemented a retrieval algorithm to enrich the dataset from the textual information [GSWH15]. However, these datasets mostly capture the vulnerabilities for larger projects such as the JCL. Information regarding vulnerabilities in smaller Java libraries is rare. Therefore, we decided to include the results of static vulnerability checks into the study. Tools such as FindBugs or Coverity search for known vulnerability patterns in code. We use both data sources in the study to have a strong comparison in the absence of a gold standard.

9.3.1. Preliminary Experiments

Using this study design, we conducted three preliminary experiments. In the first experiment we used the smaller corpus containing only the JCL, Apache Struts, and Tomcat to determine the correlation between the number of measured violations and the number of known vulnerabilities from the NVD over the version history of the included projects. We measured a moderate correlation of $r_s \approx 0.41$.

Using the same data set we conducted the second experiment, but used the number of issues found by the FindBugs tool instead of the number of known vulnerabilities. We configured FindBugs to only use the `MALICIOUS CODE VULNERABILITY`, `CORRECTNESS`, `MULTITHREADED_CORRECTNESS`, `SECURITY`, and `DODGY_CODE` categories, as they include the bug patterns for known vulnerabilities. We measured a strong correlation of $r_s \approx 0.65$ in this case.

In the third experiment we used the same configuration for FindBugs, but used the Qualitas Corpus as the code base. In this experiment we measured a very strong correlation of $r_s \approx 0.85$.

9.3.2. Threats to Validity

Measuring the security or the insecurity of a system can only capture the "known". Unknown vulnerabilities in the code are not included in our base data, thus, if our analysis finds Object-Capability Model violations which may lead to vulnerabilities that are yet unknown, the correlation would be lower. The indirection over bug finding tools is also not optimal, as vulnerability data gets biased towards the recognized patterns. The creation of a gold standard to evaluate against seems to be inevitable. This gold standard should consist of carefully developed systems with known vulnerabilities but are small enough to ensure the absence of unknown vulnerabilities.

We inspect the evolution of software projects over the course of multiple versions. All of the inspected code bases grow in this process and, therefore, could include more vulnerabilities as well as more violations than before. In the preliminary experiments we could not exclude this effect so far. Normalizing the used data over the project size would counter this effect.

9.4. Migrating Towards Explicit Authority Transfer

The violations we found in the Java language foster the creation of implicit authority transfers that are hard to understand and hard to analyze. In order to fulfill the requirements of the Object-Capability Model instances of these violation have to removed from the codebase. In the following we give advice how to proceed with these.

Given the current architecture of Java and current operating systems[3] it would be impossible to completely remove the ability to call native methods. However, we propose to remove the ability to execute Java applications that use native methods not included in the standard platform. This is already enforced for Java applets and could easily be configured into the standard policy. The residual risk of exploitation of native code from the Java side could be addressed with defensive programming and automated or manual analysis of the native codebase of Java. Especially functions that reveal low-level details such as pointer to the Java side should be redesigned. Application developers can refrain from using custom native code or libraries that ship with native code.

The use of reflection can easily be limited to non-intrusive reflection (i.e. reflection not violating encapsulation). Two options are available to the developer:

1. She either lifts the field or method accessed through intrusive reflection to an accessiblity level visible to the caller or introduces a new method which provides access to the requested field or method. For example, if a private field is read through the use of intrusive reflection, she could introduce a getter to access the field. Further access restrictions can be addressed at this point.

[3]A new approach to this problem is the Singularity project by Hunt et al. [GH07].

2. She removes the access, if it is not permitted. For example, if a private field is read through the use of intrusive reflection which should be available to the piece of code accessing the information, she removes the field access.

We argue that in a well-designed system, the need for intrusive reflection does not exist and instances of this practice should be considered as design smells w.r.t. security. The presence of intrusive reflection always indicates a loophole in effective encapsulation and can, as just presented, always be dealt with.

We found that static context and shared mutable state is prevalent in state-of-the-art Java code. Removing it from Java code is, therefore, a sizeable task. In general, it is used to transfer some form of authority from an object that created a capability to an unknown set of other objects. For instance, the internal class `sun.misc.Unsafe` has a static method `getUnsafe()` that provides the current instance of the class. Callers of this method can be rewritten in such a way that they request the instance as a method parameter or an internal field of the class. Applying this refactoring makes the authority transfer explicit and traceable. Callers of the modified method will in turn be modified to transfer the authority until its original creation is reached. However, this refactoring also impairs the maintainability of the codebase, as new method (or constructor) arguments have to be introduced and parameter values have to be passed along. The transfer of authority gets muddled with the functional parameters and behavior of the program. In the current language definition, however, it is the price to pay for explicit authority transfer.

An alternative for explicit authority transfer might be `cflow`-based aspects as implemented in AspectJ by Kiczales et al. [KHH$^+$01]. An aspect can manage the transfer of authority between parts of the program which are bound to the aspect by the use of `cflow` pointcut designators. In this design the transfer of authority is explicit and localized. It can easily be changed and adapted, yet, still can be controlled. To determine the parts of the program where authority is transferred, the pointcut designators simply need to be evaluated against the complete program. Any point that matches is a point where authority is transferred.

9.5. Summary

The Java language and runtime in its current state is not capability safe, although it is pretty close. We found three fundamental violations in the language and API specification and developed a static program analysis to find instances of these violations in Java Bytecode. We present a study design to explore the correlation between those violations and vulnerabilies in the codebase. We use two different codebases in the design of the study: The full Qualitas Corpus and a smaller corpus composed of multiple versions of the JCL, Apache Struts, and the Tomcat application server. In three preliminary experiments we found an indication for some correlation, but argue that the results are not yet conclusive. In summary, the goal of object-capability safe Java applications presents itself as desirable and achievable. However, the removal of implicit authority transfer from Java code may entail maintainability issues.

10. Related Work for Isolation

In this chapter we present related work that can provide isolation defenses for software. We start with a review on approaches for program slicing, which can separate code that supplies a specific behavior from the rest. We follow this with a review of current work in the field of object-capability security. We conclude the chapter with a review of sandboxing approaches. Its goal is to isolate possibly untrusted mobile or native code from the rest of the system.

10.1. Slicing

The automated reduction of a program to its necessary parts has been a long standing matter of interest. In our contribution presented in Chapter 8 we use it to achieve effective isolation from code that is not used in an application context.

A classic approach to remove code is program slicing which is a rich field of research and, therefore, we only present foundational work and works closely related to our use of it.

The concept of program slicing was introduced by Mark Weiser [Wei81]. Its general idea is to cut a program into parts (slices) based on subsets of program behavior in order to find a set of statements that affect a particular point in the program. Automatic slicing requires the specification of a slicing criterion, which describes what to include in the new slice. Slicing has found many applications ranging from testing and debugging to program comprehension and refactoring.

As Weiser was working on complete programs without functional decomposition, in current terms his approach could be described as intraprocedural slicing. Horwitz et al. [HRB90] extend Weiser's concept to interprocedural slicing. This means that slices can cross the boundary of procedure calls. As to be expected their main challenge is the correct representation of calling context, which Horwitz et al. address with system dependence graphs. These graphs are an extension of program dependence graphs which adds information on procedures, parameters of procedures, calls, and returns.

In a recent study Kashima et al. [KII15] compared four different program slicing approaches. They examine Static Execution Before [JBGR08], Context-insensitive Slicing, a hybrid approach of Static Execute Before and Context-insensitive Slicing, and Improved Slicing [HS04] using applications from the Qualitas Corpus [TAD$^+$10]. They find that the hybrid approach scales best of the four inspected approaches and produces smaller slices than Context-insensitive Slicing. The best approach for precision in their evaluation is the Improved Slicing approach, yet, it does not scale as well as the other approaches. Specifically it does not scale to the size of the Java Class Library.

The Improved Slicing approach inspected by Kashima et al. was developed by Hammer and Snelting [HS04]. They improve on previous algorithms for slicing programs in object-oriented languages by adding sound support for object parameters. Their approach deals with recursive data structures and is field- and object-sensitive. This makes their algorithm very precise when compared to other approaches, as Kashima et al. have shown.

Implementations for program slicing typically are limited to the scope of a specific language. However, current software systems are composed of components written in various languages. Binkley et al. [BGH+14] address this with Observation-Based Slicing (ORBS). Their approach is based on actual statement removal and execution observation. It observes a variable's behavior at a specific program point according to some slicing criterion. In contrast to dynamic slicing, which bases its criterion on an observation of a untampered execution, observation-based slicing removes a statement first and then observes the behavior during execution. They illustrate four algorithm variants and apply them to real world cross-language code successfully.

Our slicing approach from Chapter 8 uses method-level slicing which is much more coarse-grained than the approaches presented here. We made this choice in the assumption that state-of-the-art Java libraries are already decomposed sufficiently, but plan to investigate this assumption further. As in the ORBS approach, we also use execution observation to validate possible slices. ORBS already closes the circle by using slicing and execution repeatedly to produce a viable slice.

10.2. Object-Capability Security

The separation of privileges can help to make isolation provable in object-oriented programs. Miller et al. have demonstrated this with the Object-Capability Model [MS03, Mil06] which we discuss in Section 2.4. Here, we will concentrate on work regarding its application to isolation.

Two different object-capability secure language subsets for Java have been proposed that support isolation. First, there is the older J-Kernel system by Hawblitzel et al. [HCC+98, vECC+99] and, secondly, the newer Joe-E system by Mettler et al. [MWC10]. Since J-Kernel was constructed before the formulation of the Object-Capability Model it was designed with a different goal and does not honor all object-capability rules. Joe-E removes ambient authority from Java by only allowing `static` fields if they are `final`, prohibiting the introduction of native methods, and restricting reflection. Both approaches are constructive in their nature and, thus, require excessive change to the complete system including the JCL, which might not be possible in industry environments.

In another article Miller et al. [MYSI03] present the differences of the Object-Capability model compared to Access Control Lists (ACL) and clarify existing misconceptions about object-capability security. They illustrate that object-capability security is very different from ACLs by constructing and using two intermediary models. They also show that object-capability systems can enforce confinement and revocability.

A powerful property of object-capability-based systems is that abstractions can be built over the primitives in the model to express more complex scenarios such as revocation. Toby Murray inspected the implementations of popular abstractions (or patterns) in object-capability security [Mur08]. He used the CSP process algebra to formalize the implementations of three patterns for object-capability safe languages and object-capability-based operating systems. While the security properties of one pattern hold entirely in both systems, another pattern had subtle differences and another pattern fails to ensure its guarantees when moving it from the language to the operating system. As the formalization only covers the current implementations it may still be possible to implement this pattern for operating systems correctly. However, he was able to prove this particular implementation insecure.

Maffeis et al. [MMT10] analyzed multiple JavaScript subsets for their isolation properties. Their goal is to secure so-called mashups, which are web applications composed from code from multiple sources. In these applications the different components are free to interact with the user and the originating website, but not with each other. The authors show that Yahoo's ADsafe and Facebook's FBJS were vulnerable to attacks breaking isolation. They developed language-based foundations on operational semantics to show isolation proofs for the Cajita language in Google's Caja system. They found that a subset of object-capability safety they call authority safety is sufficient to provide isolation. In authority safety, two rules are sufficient: (1) Only connectivity begets connectivity and (2) the authority of a subject can only be changed by another subject inside the bounds of the acting subjects authority. They formalize these rules and prove them for a subset of JavaScript. Furthermore, they describe the first formal model for capability safety and also show that capability safety implies authority safety. Caja is the first fully object-capability safe system which is used in real-world software systems and, therefore, highly relevant to compare against.

Drossopoulou and Noble [DN13, DN14] advocate capability policies to express how capabilities in a system should be used. They observe that in current implementations of object-capability language (e.g., Joe-E or E) the policies are implicit and interwoven with the functional behavior of the program. They present a framework to specify and reason about these policies in order to make policy definition and treatment explicit. The authors use this framework to reformulate the Escrow Example, a common example where two mutually untrusting parties can exchange authority over a trusted third party [ND14]. The example shows the power of the Object-Capability Model to construct abstractions and still guarantee isolation.

Drossopoulou, Noble, and Miller [DNM15] extend the example with specifications that allow for the expression of trust and risk. Trust in this context means that an object trusts another object to adhere to a specification. Reasoning over trust here is hypothetical in such a way that both options of trust and distrust are explored. For example, if a specification describes the behavior of some method `m`, then it is only called if the object is trusted to obey this specification. The authors model risk introducing two predicates expressing "may access" and "may affect" relationships. With specifications using these new expressivity they are able to prove policy fulfillment of the implementation of the escrow example.

In contrast to these constructive approaches presented in this Section, we chose an analytical viewpoint in Chapter 9 and aim to collect motivation from practice for the wider application of the Object-Capability Model to software and language design. The work from Maffeis et al. and Drossopoulou et al. will help us in future work to describe violations in a more formal fashion, while the works of Murray and Miller provide general motivation that even complex scenarios can be addressed within the Object-Capability Model.

10.3. Other Isolation Mechanisms

Several alternative mechanisms have been proposed in order provide isolation inside the Java platform or inside of Java programs. Some of them have found their way into commonly used products. We present them here to put our contributions into perspective and to provide a comprehensive view on the security of the platform.

The general goal of sandboxing is to provide isolation of a trusted core from possibly harmful code. For instance, the Java platform provides a sophisticated sandbox to protect the Java Virtual Machine (JVM) from code running on it (cf. Section 3.3). We will start specifically with approaches that have been proposed to protect the JVM from native code (including its own) and then extend our view on sandboxes to other platforms.

Yuji Chiba [Chi08] introduced a feature that detects and locates invalid memory references to the JVM heap. The feature uses page protection, which prevents threads executing native methods from referring to the JVM heap directly, as this should only happen through the JNI. This is mandated by most common JVMs like Hotspot, which is used in the OpenJDK. As opposed to mainstream operating systems, the protection feature controls permissions by thread and not by process.

Siefers et al.'s "Robusta" [STM10] is a framework to provide safety and security for Java applications and applets through software-based fault isolation which places the native code into a sandbox. Their sandboxing is based on Google's Native Client extending it with support for dynamic linking and loading of native libraries. Sun and Tan [ST12] introduced "Arabica", an enhancement of Robusta that is portable between different Java VMs by avoiding modification of VM internals. Arabica relies on a combination of the Java Virtual Machine Tool Interface (JVMTI) and a layer of stub libraries. With SafeJNI, Tan et al. [TCSW06] developed a system which, by static and dynamic checks, retro-fits native C methods to enhance their safety and captures safety invariants whose satisfaction aims at guaranteeing a safe inter-operation between Java and native code.

Also, as Sun et al. [ST14] point out, the majority of the Top 50 Android apps are shipped with native libraries. Building on their outstanding work for the Java Native Interface [ST12, TCSW06, TC08, STM10], they build an isolation model for native libraries in Android apps effectively protecting users from malicious or faulty native code.

Cappos et al. [CDR$^+$10] construct a Python-based sandbox with a small, isolated kernel library. By this, they prevent attackers from leveraging bugs in the kernel library for privilege escalation. Moreover, De Groef et al. [DGNYP10] are concerned with the integrity of the execution process itself. As they point out, Write-XOR-Execute protection mechanisms of operating systems (cf. Section 3.1) cannot be applied for applications with Just-in-time compilation as the runtime needs to write into and execute the same memory block. Their system separates sensitive from non-sensitive code and protect the system by blocking sensitive code.

Part IV.

Conclusion and Outlook

11. Conclusion

In this chapter, we will present a brief overview over the findings from this thesis. We start with a review of the results and contributions of this thesis and follow this with a closing discussion.

11.1. Summary of Results

We contribute three analyses to the field of monitoring defense techniques in this thesis. They cover the full stack of a Java application from the application code down to the native parts of the Java Class Library. However, we use rather different analysis approaches and have made different findings on the layers of the application stack.

Automated Detection of Unsafe Programming Practices Our analysis to detect unsafe programming practices reports the use of pointer arithmetic, pointer type casts, dynamic memory management, and impure programming to the developer. The analysis scales to the size of the native codebase of the Java Class Library on commodity hardware. We are, therefore, confident to use it in large-scale scenarios or when response time is a key asset. It reliably points out hot spots that justify closer inspection and, therefore, efficiently guides code reviews.

Prevalence of Unsafe Programming Practices in the JCL We used the previous analysis on the Java Class Library and found interesting results. While there were packages of functions using all of the detected unsafe programming practices, we also found several that were just using impure functions. As the main purpose of the JCL's native part is operating system binding, which naturally entails side effects, this was to be expected, yet, the absence of other such practices is interesting. As our evaluation shows, there seems to be a strong connection between vulnerabilities and the use of these practices.

Automated Detection of Capability Usage in Java Libraries We demonstrate an analysis to detect coarse-grained capability usages in Java libraries. These capabilities include file system or network socket access, but also Java facilities such as reflection. With this analysis developers can make educated choices on library usage in their specific security context and decide whether to trust a library or not. As part of this analysis, we provide a manually-created data set of native functions and their respective capability provision for Java.

Capability Usage in Common Java Libraries In our evaluation we applied the analysis to 70 widely used Java libraries and found interesting discrepancies to their documentation. Almost all of them were using features from `sun.misc.Unsafe` and reflection facilities, which is in line with findings from Mastrangelo et al. [MPM+15]. Some of them used class loading, which is unexpected for most libraries. We inspected these results closer and found several open class loader vulnerabilities in libraries.

Efficient Infeasible Path Detection Our third analysis is targeting infeasible (or dead) paths in Java code. These paths are either a consequence of technical necessities (such as compilation), which makes them benign, and therefore irrelevant, or pointing towards implementation defects stemming from various sources. In a study on the Java Class Library, we identified ten categories of defects for the 258 bugs we found. As our approach in not targeted towards the Java Class Library we are confident we find more categories in other code as we are not detecting the cause, but the effect of the programming mistake. This enables programmers to find various, very complex bugs using our analysis.

Detection of 258 Complex Data-Flow Dependent Bugs in the JCL As we applied our analysis to the Java Class Library to categorize our findings, we found 258 previously undetected bugs inside the JCL. Some of these bugs only impede the maintainability of the code base, such as excessive checks, but some of them deactivate functionality or lead to miscalculations.

Library Slicing for Attack Surface Reduction We extend our analysis for library capability footprints and present a method to slice libraries down the necessary part in the context of an application. With this method we can reduce the size of a library on average by 47% in an experiment using example code. Moreover, the approach removes the use of capabilities and provides libraries with a smaller footprint, thus, honoring the Principle of Least Privilege.

11.2. Closing Discussion

After summarizing the contributions and findings of this thesis, we will briefly relate them to current trends and take a look at future challenges.

The areas of research where programming languages, software engineering, and security intersect and converge have a huge potential to shape the security of future software systems. Language and platform guarantees such as Java's memory safety have already helped to secure the operation of current systems and pushed buffer-overflow attacks off the first place in vulnerability rankings. Also specialized analyses have been constructed to address several buffer overflow related vulnerabilities. Yet, with the evolution of software development new threats and challenges have to be tackled.

The execution of mobile code is still prominent, even with the drop in Java applet usage. Cisco's Annual Security Report 2016 [Cis16] names browser extensions as a new but prominent source of vulnerabilities, yet, criminals still seems to exploit new vulnerabilities in Adobe Flash. We assume that mobile code will remain to be a desired feature of software systems as the plugin paradigm remains to be a prestigious selling point. As we have learned, sandboxing mobile code is important, yet also a process to be expected to contain faults (cf. Section 3.3, Chapter 10, Chapter 4). Thus, second tier monitoring and isolation schemes have to be used to provide effective protection from malicious mobile code.

Another interesting observation Cisco's report makes, is that web platforms (e.g., WordPress) get exploited more often than in the past. Exploiters use constructed values as attack vectors to use vulnerabilities in the same way that Bratus et al. [BLP+11] describe weird machines. As we also demonstrated in this thesis (cf. Chapter 4), attacks against platforms even without the use of mobile code are possible and realistic. The importance of input guarding and sanitization as well as the use of decidable input languages remains to be an interesting topic.

A new trend in the commercialization of exploits is hard to overlook. Remotely controlled ransomware uses a vulnerability of the target machine to achieve control in order to encrypt the hard drive of the machine. The decryption key is only available on payment of a ransom over the untraceable Bitcoin currency. This new commercialization scheme has given a new rise to machine exploitation and puts an even larger emphasis on research of new defenses. As machine control has become a more valuable asset than before, control-flow related issues should be prioritized over data-flow related issues.

Special care should be given to the topic of security when creating a new language or runtime platform. As current compilers include much more checks than those created in the beginning of the C language, so should compilers for new languages allow for more and more code security guarantees when translating software to machine code. Type systems play an important role in this endeavor and should enjoy a major emphasis in language-based security research.

12. Future Work

In this chapter we present further ideas for future work which are in an earlier stage as those presented in Chapter 9. Nevertheless, we present sound and promising ideas that provide interesting challenges. Naturally the ideas span the complete program stack visited in this thesis.

Reliably Finding Guards and Sanitizers The precision of our approach presented in Chapter 4 would greatly improve if we could reliably detect guard clauses and sanitizer functions. Guard clauses are the conditions that protect the execution of sensitive operations. For instance, offset calculations on pointers can be used to direct the pointer to memory locations that should not be accessible for the current function. A guard clause is a condition that only permits the offset calculation if the offset is in well-defined bounds. A sanitizer function in this example would take the provided offset as its input and always return a valid offset regardless of the validity of the input value. A detection may build on these observations and track the data flow of these input values in order to find out if they are treated properly. Detecting these guards and sanitizers reliably and rating their effectiveness in protecting a sensitive operation may reveal cases where these safeguards are missing or are ineffective.

Analyze Native Binaries using Decompilation Recent advances in decompilation technology [YEGS15] opens the possibility for static analysis of programs in native binary form. We plan to use such a decompiler as a first step and then use our analysis techniques based on the LLVM platform to find possible vulnerabilities and used capabilities. This enables us to apply our analyses on software where the source code is not available. However, the complete chain from binary code to LLVM bitcode must faithfully reflect the operations found in the native binary otherwise it would render the findings of the analysis unsound. The resulting analyses could also help in the identification and inspection of malware and the security assessment of native implementations delivered with Java libraries.

Analyze Critical Data Flows over Language Boundaries Despite its age and maturity the Java platform still seems to be a rich source of exploitable vulnerabilities. As we discussed in Chapter 4 some of these exploits target vulnerabilities in the native part of the JCL but use delivery mechanisms that only allow them to use Java code. This means that vulnerabilities in the native part have to be exploited using prepared values that are passed as arguments to these functions. We plan to track these data flows in order to find access paths that do not use proper access control checks and do not implement necessary guard clauses or sanitization routines. The output of such an analysis would greatly help in the identification of loopholes in the security mechanisms of the Java platform and would have a major impact on many users.

Two ideas can help to implement this analysis. First, as the callers of the native functions in the JCL can be enumerated, it is possible to use a stored data flow abstraction when crossing the language border. This allows to split the data flow analysis into parts for each language in order to avoid having to build a common abstraction over both programming languages. This in turn helps to fully leverage the guarantees given in each language. Second, the detection of guards and sanitizer presented before as well as the detection of unsafe programming practices can aid to produce meaningful results that lead to reliable vulnerability detection.

Outlier Detection for FFI Calls Another idea is closely related to the previous idea. Calls to foreign function interfaces (FFI) such as native functions in Java only work safely on a subset on the values that can be passed to them not violating type safety rules. For this reason callers employ guard clauses and sanitization functions to narrow the set of possible values passed to these functions. However, callers might use different techniques that result in different value sets for the same function. We performed a preliminary study in the JCL and found interesting outliers for integer type parameter ranges in calling native functions. We plan to implement an analysis that performs outlier detection for FFI calls for all possible types. Developers can find possibly vulnerable call path from these outliers by finding missing checks or faulty callers.

Extracting Capabilities from Native Code In our capability inference presented in Chapter 5 we manually bootstrapped the algorithm with data extracted from the native code base of the JCL. As this process has to be repeated for each release of Java we would like to automate it. We plan to implement this in an automated analysis of the JCL's native source code using LLVM. Usage and proliferation of operating system capabilities can be detected using this analysis making the process not only repeatable for new versions of Java, but also applicable to native libraries shipped with other Java libraries or to implement a capability analysis for completely native libraries.

Studies on the Practical Value of Our Work In all of our work presented in this thesis, we target developers in their pursuit of developing secure systems. Up to this point we evaluated our approaches by means of case studies, experiments regarding the precision of the analyses, and proxies for developer expectations. After these necessary steps to prove the validity of our approaches, we plan to conduct user studies with professional developers in order to measure effects in real-world projects. All our our analyses have in common, that they should make the detection of software flaws easier for the developer to find. Through a set of bug finding scenarios we would like to find out if this is actually the case.

Automatic Vulnerability Remedies Recent work in automatic program repair [LSDR14, LR15, LR16] has sparked our interest whether the same principles could be a applied to automated vulnerability remedies as well. The idea is to take our findings from static analysis and generate guard clauses or sanitizer functions removing the vulnerability from the program code. While there is work on sanitizer placement [LC13], we found no work on automated generation and placement of guards and sanitizers.

Contributed Implementations and Data

In the course of the projects presented in this thesis, research prototypes have been implemented and data has been sourced or collected. We provide these implementations and data sets to enable other researchers to validate our work and to build new research upon them. We believe this to be good scientific practice and encourage other researchers to do the same.

Analysis for Unsafe Languages

The implementation of the analysis presented in Chapter 4 is provided as a module to be compiled into LLVM. We provide the source code of the analysis here:

`https://github.com/stg-tud/peaks-native`

The `StaticAnalyses` folder contains the analysis code and a Makefile to include these analyses in the compilation process of the `opt` tool. This is in compliance with the standard process for LLVM version 3.5. The repository also contains a `Tests` folder containing test cases for the analysis. Finally, the `NativeCompilation.gmk` file contains the modified version of the OpenJDK8 build process based on the file shipped with build 132.

The full artifact can be inspected here:

`http://www.st.informatik.tu-darmstadt.de/artifacts/peaks-native/`

Analysis for High-Level Capabilities

We provide the implementation of the analysis presented in Chapter 5 here:

`https://github.com/stg-tud/peaks-capmodel`

The analysis and the evaluation tooling is written in Scala and is based on the OPAL analysis framework [EH14]. It compiles down to Java Bytecode and will run on any machine with the Java 1.8 runtime installed. We were successfully able to run it using the Windows 8 and MacOS X 10.10.3 operating systems. It analyses any provided library in Java Bytecode format and allows many command-line arguments. It can be integrated into other project easily by means of white-box reuse.

We provide the full replication artifact at the following URL:

`http://www.st.informatik.tu-darmstadt.de/artifacts/peaks-capmodel/`

For reproducing the evaluation charts and statistics you may need R[1] and a standard browser (we recommend using Firefox). The reproduction package contains the full set of libraries from our evaluation but may need to connect to various websites in the process to collect library documentation.

It also features the dataset created in our manual bootstrapping process in the file `NativeMethodsRT.csv`.

Dead Path Analysis

We provide the source code of the analysis as part of the OPAL project here:

`https://bitbucket.org/delors/opal`

A binary version of the tool is available for download at:

`http://www.opal-project.de/tools/bugpicker`

It is able to operate on any artifact in Java Bytecode and presents the found issues in a graphical user interface with further explanations. The tool is the frontend for many different analyses. In order to replicate our results presented here, chose the Dead Path Analysis.

Slicing Java Libraries

We provide the implementation of the slicing approach presented in Chapter 8 in the same repository as the High-Level Capability Analysis:

`https://github.com/stg-tud/peaks-capmodel`

The implementation is written in Scala and is based on the OPAL analysis framework [EH14]. It compiles down to Java Bytecode and will run on any machine with the Java 1.8 runtime installed.

[1] `http://www.r-project.org/`

Bibliography

[AB01] Cyrille Artho and Armin Biere. Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs. In *Proceedings of the 13th Australian Conference on Software Engineering*, ASWEC '01, pages 68–75, Washington, DC, USA, 2001. IEEE Computer Society.

[ABB$^+$10] José Bacelar Almeida, Endre Bangerter, Manuel Barbosa, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. *Computer Security – ESORICS 2010: 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*, chapter A Certifying Compiler for Zero-Knowledge Proofs of Knowledge Based on Σ-Protocols, pages 151–167. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[ABEL05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[ADL$^+$05] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, Monica S. Lam, V. Benjamin, Livshits Monica, and S. Lam. Improving software security with a c pointer analysis. In *In ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 332–341. ACM Press, 2005.

[AF03] Martín Abadi and Cédric Fournet. Access control based on execution history. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society, 2003.

[Akr10] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[ALS13] Stephan Arlt, Zhiming Liu, and Martin Schäf. *Formal Methods and Software Engineering: 15th International Conference on Formal Engineering Methods, ICFEM 2013, Queenstown, New Zealand, October 29 – November 1, 2013, Proceedings*, chapter Reconstructing Paths for Reachable Code, pages 431–446. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[AP10] Nathaniel Ayewah and William Pugh. The Google FindBugs Fixit. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 241–252, New York, NY, USA, 2010. ACM.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[AS12] Stephan Arlt and Martin Schäf. Joogie: Infeasible Code Detection for Java. In *CAV'12: Proceedings of the 24th international conference on Computer Aided Verification.* Springer-Verlag, July 2012.

[Bac02] Michael Bacarella. Taking advantage of linux capabilities. *Linux Journal*, 2002(97):4–, May 2002.

[BB13] Alberto Bacchelli and Christian Bird. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the International Conference on Software Engineering.* IEEE, May 2013.

[BBC+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM*, 53(2):66–75, February 2010.

[BG09] Ioannis G. Baltopoulos and Andrew D. Gordon. Secure compilation of a multi-tier web language. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 27–38, New York, NY, USA, 2009. ACM.

[BGH+14] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. Orbs: Language-independent program slicing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 109–120, New York, NY, USA, 2014. ACM.

[BKLTM12] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 274–277, 2012.

[Bla14] Mike Bland. Finding more than one worm in the apple. *Communications of the ACM*, 57(7):58–64, July 2014.

[BLP$^+$11] Sergey Bratus, Michael E Locasto, Meredith L Patterson, Len Sassaman, and Anna Shubina. Exploit programming: From buffer overflows to weird machines and theory of computation. *USENIX; login*, pages 13–21, 2011.

[BN05] Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005.

[BNR06] G. Barthe, D. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for java. In *Security and Privacy, 2006 IEEE Symposium on*, pages 13 pp.–242, May 2006.

[Boe99] Barry W Boehm. Managing Software Productivity and Reuse. *IEEE Computer*, 32(9):111–113, 1999.

[BSS$^+$11] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM.

[BSS12] Cristiano Bertolini, Martin Schäf, and Pascal Schweitzer. Infeasible code detection. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE'12, pages 310–325, Berlin, Heidelberg, 2012. Springer-Verlag.

[CBP$^+$15] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., August 2015. USENIX Association.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[CCF$^+$09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[CCK11] Hong-Zu Chou, Kai-Hui Chang, and Sy-Yen Kuo. Facilitating Unreachable Code Diagnosis and Debugging. *2011 16th Asia and South Pacific Design Automation Conference ASP-DAC 2011*, pages 1–6, February 2011.

[CDR+10] Justin Cappos, Armon Dadgar, Jeff Rasley, Justin Samuel, Ivan Beschastnikh, Cosmin Barsan, Arvind Krishnamurthy, and Thomas Anderson. Retaining sandbox containment despite bugs in privileged memory-safe code. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, 2010.

[CGK15] Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding caller-sensitive method vulnerabilities: A class of access control vulnerabilities in the java platform. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2015, pages 7–12, New York, NY, USA, 2015. ACM.

[CHH+06] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. Improving Your Software Using Static Analysis to Find Bugs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 673–674, New York, NY, USA, 2006. ACM.

[Chi08] Yuji Chiba. Java heap protection for debugging native methods. *Science of Computer Programming*, 70(2–3):149 – 167, 2008. Special Issue on Principles and Practices of Programming in Java (PPPJ 2006).

[Cis16] Cisco Systems, Inc. Cisco 2016 annual security report. `http://www.cisco.com/c/m/en_us/offers/sc04/2016-annual-security-report/index.html`, 2016.

[CLN+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 95–107, New York, NY, USA, 2000. ACM.

[CPM+98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[CW14] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, August 2014. USENIX Association.

[CZY+14] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. Ropecker: A generic and practical approach for defending against ROP attacks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[DEMDS00] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler Techniques for Code Compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, March 2000.

[DGNYP10] Willem De Groef, Nick Nikiforakis, Yves Younan, and Frank Piessens. Jitsec: Just-in-time security for code injection attacks. In *Benelux Workshop on Information and System Security (WISSEC 2010)*, 2010.

[DKA$^+$14] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[DN13] Sophia Drossopoulou and James Noble. The need for capability policies. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*, FTfJP '13, pages 6:1–6:7, New York, NY, USA, 2013. ACM.

[DN14] Sophia Drossopoulou and James Noble. *Integrated Formal Methods: 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*, chapter How to Break the Bank: Semantics of Capability Policies, pages 18–35. Springer International Publishing, Cham, 2014.

[DNM15] Sophia Drossopoulou, James Noble, and Mark S. Miller. Swapsies on the internet: First steps towards reasoning about risk and trust in an open world. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*, PLAS'15, pages 2–15, New York, NY, USA, 2015. ACM.

[DVH66] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, March 1966.

[DWW$^+$10] Yu Ding, Tao Wei, TieLei Wang, Zhenkai Liang, and Wei Zou. Heap taichi: Exploiting memory allocation granularity in heap-spraying attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 327–336, New York, NY, USA, 2010. ACM.

[EH14] Michael Eichberg and Ben Hermann. A Software Product Line for Static Analyses: The OPAL Framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, pages 1–6, New York, NY, USA, 2014. ACM.

[EHMG15] Michael Eichberg, Ben Hermann, Mira Mezini, and Leonid Glanz. Hidden truths in dead software paths. In *Proceedings of the 2015 10th Joint Meeting* *on Foundations of Software Engineering*, ESEC/FSE 2015, pages 474–484, New York, NY, USA, 2015. ACM.

[EJJ$^+$12] Sebastian Eder, Maximilian Junker, Elmar Jürgens, Benedikt Hauptmann, Rudolf Vaas, and Karl-Heinz Prommer. How Much Does Unused Code Matter for Maintenance? In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1102–1111, Piscataway, NJ, USA, 2012. IEEE Press.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, Jan 2002.

[FF06] Michael Furr and Jeffrey S. Foster. Polymorphic type inference for the jni. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 309–324, Berlin, Heidelberg, 2006. Springer-Verlag.

[FL02] Andrew Forward and Timothy C Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33. ACM, 2002.

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[FMSW08] Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. Verifiable functional purity in java. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 161–174, New York, NY, USA, 2008. ACM.

[Fra12] Ivan Fratric. Ropguard: runtime prevention of return-oriented programming attacks. `https://github.com/ivanfratric/ropguard`, 2012.

[GE03] Li Gong and Gary Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation.* Pearson Education, 2nd edition, 2003.

[GH07] Jim Larus Galen Hunt. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41:37–49, April 2007.

[GJSB09] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, 2000.* Sun Microsystems, 2009.

[Gon98] Li Gong. Java security architecture (jdk1.2). Technical report, 1998.

[Gon11] Li Gong. Java security architecture revisited. *Commun. ACM*, 54(11):48–52, November 2011.

[Gos95] James Gosling. Java intermediate bytecodes. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, IR '95, pages 111–118, New York, NY, USA, 1995. ACM.

[GPT+09] Emmanuel Geay, Marco Pistoia, Takaaki Tateishi, Barbara G Ryder, and Julian Dolby. Modular string-sensitive permission analysis with demand-driven precision. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 177–187. IEEE Computer Society, May 2009.

[Gri93] Martin L Griss. Software Reuse: From Library to Factory. *IBM Systems Journal*, 32(4):548–566, 1993.

[GSWH15] Leonid Glanz, Sebastian Schmidt, Sebastian Wollny, and Ben Hermann. A vulnerability's lifetime: Enhancing version information in cve databases. In *Proceedings of the 15th International Conference on Knowledge Technologies and Data-driven Business*, i-KNOW '15, pages 28:1–28:4, New York, NY, USA, 2015. ACM.

[GYF06] Emmanuel Geay, Eran Yahav, and Stephen Fink. Continuous Code-quality Assurance with SAFE. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pages 145–149, New York, NY, USA, 2006. ACM.

[Har85] Norman Hardy. Keykos architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, October 1985.

[Har88] Norm Hardy. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[HCC+98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing Multiple Protection Domains in Java. *USENIX Annual Technical Conference 1998*, 1998.

[HDG+11] Lars Heinemann, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, and Maximilian Irlbeck. *Top Productivity through Software Reuse: 12th International Conference on Software Reuse, ICSR 2011, Pohang, South Korea, June 13-17, 2011. Proceedings*, chapter On the Extent and Nature of Software Reuse in Open Source Java Projects, pages 207–222. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2004.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[HREM15] Ben Hermann, Michael Reif, Michael Eichberg, and Mira Mezini. Getting to know you: Towards a capability model for java. In *Proceeding of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE'15, pages 758–769, New York, NY, USA, 2015. ACM.

[HS04] Christian Hammer and Gregor Snelting. An improved slicer for java. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04, pages 17–22, New York, NY, USA, 2004. ACM.

[Jax13] Jaxin. A java 6 killer – cve-2013-2465 (update, now with cve-2013-2463), 27.08.2013. http://wraithhacker.com/a-java-6-killer-cve-2013-2465/.

[JBGR08] J. Jasz, A. Beszedes, T. Gyimothy, and V. Rajlich. Static execute after/before as a replacement of traditional software dependencies. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 137–146, Sept 2008.

[JSMHB13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.

[KBC$^+$14] Baris Kasikci, Thomas Ball, George Candea, John Erickson, and Madanlal Musuvathi. Efficient Tracing of Cold Code via Bias-Free Sampling. *USENIX Annual Technical Conference*, pages 243–254, 2014.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[KII15] Yu Kashima, Takashi Ishio, and Katsuro Inoue. Comparison of backward slicing techniques for java. *IEICE TRANSACTIONS on Information and Systems*, 98(1):119–130, 2015.

[KO08] Goh Kondoh and Tamiya Onodera. Finding bugs in java native interface programs. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 109–118, New York, NY, USA, 2008. ACM.

[Koz99] Dexter Kozen. *Mathematical Foundations of Computer Science 1999: 24th International Symposium, MFCS'99 Szklarska Poreba, Poland, September 6–10,1999 Proceedings*, chapter Language-Based Security, pages 284–298. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[KPK02] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access rights analysis for Java. *ACM Sigplan Notices*, 37(11):359, November 2002.

[KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial Dead Code Elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 147–158, New York, NY, USA, 1994. ACM.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[LC13] Benjamin Livshits and Stephen Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 385–398, New York, NY, USA, 2013. ACM.

[Ler01] Xavier Leroy. *Computer Aided Verification: 13th International Conference, CAV 2001 Paris, France, July 18–22, 2001 Proceedings*, chapter Java Bytecode Verification: An Overview, pages 265–285. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[Lev96] Elias Levy. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[LHBM14] Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. Flowtwist: Efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 98–108, New York, NY, USA, 2014. ACM.

[LR15] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015. ACM.

[LR16] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 298–312, New York, NY, USA, 2016. ACM.

[LSDR14] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings* *of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 227–238, New York, NY, USA, 2014. ACM.

[LT09] Siliang Li and Gang Tan. Finding bugs in exceptional situations of jni programs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 442–452, New York, NY, USA, 2009. ACM.

[LT11] Siliang Li and Gang Tan. Jet: Exception checking in the java native interface. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 345–358, New York, NY, USA, 2011. ACM.

[LT14] Siliang Li and Gang Tan. Exception analysis in the java native interface. *Science of Computer Programming*, 89, Part C(0):273 – 297, 2014.

[LWH$^+$10] Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. Jinn: Synthesizing dynamic bug detectors for foreign language interfaces. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 36–49, New York, NY, USA, 2010. ACM.

[LYBB14] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification.* Pearson Education, 2014.

[LZC$^+$10] Z. Li, Z. Zhuang, Y. Chen, S. Yang, Z. Zhang, and D. Fan. A certifying compiler for clike subset of c language. In *Theoretical Aspects of Software Engineering (TASE), 2010 4th IEEE International Symposium on*, pages 47–56, Aug 2010.

[MCG$^+$99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talx86: A realistic typed assembly language. In *In Second Workshop on Compiler Support for System Software*, pages 25–35, 1999.

[Mee10] Haroon Meer. Memory corruption attacks the (almost) complete history. *Blackhat USA.(Jul. 2010)*, 2010.

[Mil06] Mark Samuel Miller. *Robust composition: towards a unified approach to access control and concurrency control.* PhD thesis, 2006.

[MM07] Fabio Martinelli and Paolo Mori. Enhancing java security with history based access control. In Alessandro Aldini and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design IV*, volume 4677 of *Lecture Notes in Computer Science*, pages 135–159. Springer Berlin Heidelberg, 2007.

[MMT10] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 125–140, May 2010.

[MMZ08] Federico Maggi, Matteo Matteucci, and Stefano Zanero. Detecting Intrusions through System Call Sequence and Argument Analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(4):381–395, 2008.

[MNZZ] Andrew C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. `http://www.cs.cornell.edu/jif/`.

[MPM+15] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: The java unsafe api in the wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 695–710, New York, NY, USA, 2015. ACM.

[MS03] Mark S. Miller and Jonathan S. Shapiro. *Advances in Computing Science – ASIAN 2003. Progamming Languages and Distributed Computation Programming Languages and Distributed Computation: 8th Asian Computing Science Conference, Mumbai, India, December 10-12, 2003. Proceedings*, chapter Paradigm Regained: Abstraction Mechanisms for Access Control, pages 224–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[MS13] Tilo Müller and Michael Spreitzenbarth. Frost: Forensic recovery of scrambled telephones. In Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security*, volume 7954 of *Lecture Notes in Computer Science*, pages 373–388. Springer Berlin Heidelberg, 2013.

[MSL+08] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Safe active content in sanitized javascript. Technical report, 2008.

[Mur08] Toby Murray. Analysing object-capability security, in. In *Proceedings of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS'08*, pages 177–194, 2008.

[MW10] Adrian Mettler and David Wagner. Class properties for security review in an object-capability subset of java: (short paper). In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 7:1–7:7, New York, NY, USA, 2010. ACM.

[MWC10] Adrian Mettler, David Wagner, and Tyler Close. Joe-e: A security-oriented subset of java. In *NDSS*, volume 10, pages 357–374, 2010.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, May 1999.

[Mye99] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[MYSI03] Mark Miller, Ka-Ping Yee, Jonathan Shapiro, and Combex Inc. Capability myths demolished. Technical report, 2003.

[ND14] James Noble and Sophia Drossopoulou. Rationally reconstructing the escrow example. In *Proceedings of 16th Workshop on Formal Techniques for Java-like Programs*, FTfJP'14, pages 5:1–5:6, New York, NY, USA, 2014. ACM.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.

[Ner01] Nergal. Advanced return-into-lib(c) exploits: Pax case study. *Phrack*, 58, 2001.

[NL04] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. *SIGPLAN Not.*, 39(4):612–625, April 2004.

[NLR10] Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical extensions to the ifds algorithm. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 124–144, Berlin, Heidelberg, 2010. Springer-Verlag.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[ora14a] Secure coding guidelines for java se. `http://www.oracle.com/technetwork/java/seccodeguide-139067.html`, 2014.

[Ora14b] Oracle. Java native interface specification, 06.08.2014. http://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html.

[Pap12] Vasilis Pappas. kbouncer: Efficient and transparent rop mitigation. Technical report, 2012.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[PBN07] M. Pistoia, A. Banerjee, and D.A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 149–163, May 2007.

[Pro14] LLVM Project. Llvm language reference manual, 30.07.2014. http://llvm.org/docs/LangRef.html.

[PS11] Étienne Payet and Fausto Spoto. Static Analysis of Android Programs. In *Proceedings of the 23rd International Conference on Automated Deduction*, CADE'11, pages 439–445, Berlin, Heidelberg, 2011. Springer-Verlag.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[RTC] John Rose, Christian Thalinger, and Mandy Chung. Jep 176: Mechanical checking of caller-sensitive methods. `http://openjdk.java.net/jeps/176`.

[Rus81] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM.

[Sch11] Fred B. Schneider. Blueprint for a science of cybersecurity. Technical report, Cornell University, 2011.

[SE13] Widura Schwittek and Stefan Eicker. A study on third party component reuse in java enterprise open source software. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, pages 75–80, New York, NY, USA, 2013. ACM.

[SHR$^+$00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 264–280, New York, NY, USA, 2000. ACM.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003.

[SMH01] Fred B Schneider, Greg Morrisett, and Robert Harper. A Language-Based Approach to Security. In *Computer Aided Verification*, pages 86–101. Springer Berlin Heidelberg, Berlin, Heidelberg, March 2001.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[SPBL13] L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto. Security applications of formal language theory. *IEEE Systems Journal*, 7(3):489–500, Sept 2013.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, November 1984.

[SS75] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[SSNW13] Martin Schäf, Daniel Schwartz-Narbonne, and Thomas Wies. Explaining Inconsistent Code. *the 2013 9th Joint Meeting*, pages 1–11, August 2013.

[ST12] Mengtao Sun and Gang Tan. Jvm-portable sandboxing of java's native libraries. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security – ESORICS 2012*, volume 7459 of *Lecture Notes in Computer Science*, pages 842–858. Springer Berlin Heidelberg, 2012.

[ST14] Mengtao Sun and Gang Tan. NativeGuard: protecting android applications from third-party native libraries. In *WiSec '14: Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks.* ACM Request Permissions, July 2014.

[STM10] Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: Taming the native beast of the jvm. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 201–211, New York, NY, USA, 2010. ACM.

[STP$^+$14] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. *Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, chapter Evaluating the Effectiveness of Current Anti-ROP Defenses, pages 88–108. Springer International Publishing, Cham, 2014.

[SVR05] Fred Spiessens and Peter Van Roy. The oz-e project: Design guidelines for a secure multiparadigm programming language. In *Multiparadigm Programming in Mozart/Oz*, pages 21–40. Springer, 2005.

[TAD$^+$10] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, December 2010.

[Tan10] Gang Tan. Jni light: An operational model for the core jni. In *Proceedings of the 8th Asian Conference on Programming Languages and Systems*, APLAS'10, pages 114–130, Berlin, Heidelberg, 2010. Springer-Verlag.

[TC08] Gang Tan and Jason Croft. An empirical security study of the native code in the jdk. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 365–377, Berkeley, CA, USA, 2008. USENIX Association.

[TCSW06] Gang Tan, Srimat Chakradhar, Raghunathan Srivaths, and Ravi Daniel Wang. Safe java native interface. In *In Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*, pages 97–106, 2006.

[TM07] Gang Tan and Greg Morrisett. Ilea: Inter-language analysis across java and c. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 39–56, New York, NY, USA, 2007. ACM.

[VBKM00] John Viega, J. T. Bloch, Yoshi Kohno, and Gary McGraw. Its4: a static vulnerability scanner for c and c++ code. In *Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference*, pages 257–267, Dec 2000.

[vECC$^+$99] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: A Capability-Based Operating System for Java. In *Computer Aided Verification*, pages 369–393. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[VL13] J Vanegue and S K Lahiri. Towards Practical Reactive Security Audit Using Extended Static Checkers. In *2013 IEEE Symposium on Security and Privacy (SP) Conference*, pages 33–47. IEEE, 2013.

[VRCG$^+$10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[WAF00] Dan S. Wallach, Andrew W. Appel, and Edward W. Felten. Safkasi: A security mechanism for language-based systems. *ACM Trans. Softw. Eng. Methodol.*, 9(4):341–378, October 2000.

[WALK10] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 249–257, New York, NY, USA, 1998. ACM.

[YEGS15] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*. The Internet Society, 2015.

[Yel95] Frank Yellin. Low level security in java. In *In Proceedings of the 4th International World Wide Web Conference*, 1995.

[YPC$^+$10] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. Paricheck: An efficient pointer arithmetic checker for c programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 145–156, New York, NY, USA, 2010. ACM.