A Theory and Tools for Applying Sandboxes Effectively

Michael Maass

CMU-ISR-16-105
March 2016

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich (Co-chair)
William L. Scherlis (Co-chair)
Lujo Bauer
Bruno Amizic (The Boeing Company)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This material is based upon work supported by the US Department of Defense through the Office of the Assistant Secretary of Defense for Research and Engineering (ASD(R&E)) under Contract HQ0034-13-D-0004, the Army Research Office under Award No. W911NF-09-1-0273, the Air Force Research Laboratory under Award No. FA87501220139, the National Security Agency under Lablet Contract H98230-14-C-0140, the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1252522. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of ARL, ARO, ASD (R&E), NSA, or NSF. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Keywords: Sandboxing, software protection, supply chain security, Java sandbox, software engineering

Abstract

It is more expensive and time consuming to build modern software without extensive supply chains. Supply chains decrease these development risks, but typically at the cost of increased security risk. In particular, it is often difficult to understand or verify what a software component delivered by a third party does or could do. Such a component could contain unwanted behaviors, vulnerabilities, or malicious code, many of which become incorporated in applications utilizing the component.

Sandboxes provide relief by encapsulating a component and imposing a security policy on it. This limits the operations the component can perform without as much need to trust or verify the component. Instead, a component user must trust or verify the relatively simple sandbox. Given this appealing prospect, researchers have spent the last few decades developing new sandboxing techniques and sandboxes. However, while sandboxes have been adopted in practice, they are not as pervasive as they could be. Why are sandboxes not achieving ubiquity at the same rate as extensive supply chains? This thesis advances our understanding of and overcomes some barriers to sandbox adoption.

We systematically analyze ten years (2004 – 2014) of sandboxing research from top-tier security and systems conferences. We uncover two barriers: (1) sandboxes are often validated using relatively subjective techniques and (2) usability for sandbox deployers is often ignored by the studied community.

We then focus on the Java sandbox to empirically study its use within the open source community. We find features in the sandbox that benign applications do not use, which have promoted a thriving exploit landscape. We develop run time monitors for the Java Virtual Machine (JVM) to turn off these features, stopping all known sandbox escaping JVM exploits without breaking benign applications. Furthermore, we find that the sandbox contains a high degree of complexity benign applications need that hampers sandbox use.

When studying the sandbox's use, we did not find a single application that successfully deployed the sandbox for security purposes, which motivated us to overcome benignly-used complexity via tooling. We develop and evaluate a series of tools to automate the most complex tasks, which currently require error-prone manual effort. Our tools help users derive, express, and refine a security policy and impose it on targeted Java application JARs and classes. This tooling is evaluated through case studies with industrial collaborators where we sandbox components that were previously difficult to sandbox securely.

Finally, we observe that design and implementation complexity causes sandbox developers to accidentally create vulnerable sandboxes. Thus, we develop and evaluate a sandboxing technique that leverages existing cloud computing environments to execute untrusted computations. Malicious outcomes produced by the computations are contained by ephemeral virtual machines. We describe a field trial using this technique with Adobe Reader and compare the new sandbox to existing sandboxes using a qualitative framework we developed.

For my father, whose early hacking efforts brought an indelible passion for technology into our family.

Acknowledgments

It is impossible to acknowledge all of the people that helped me find my way into and through graduate school. Growing up, I never dreamed of, or even really considered, attending graduate school, primarily due to a lack of role models. I decided to attend a university after high school thanks to the encouragement of two science and math teachers: Mary Davis and Jennifer Anderson. While studying Electrical Engineering I figured I would graduate, get a software security job somewhere, and carve out a career for myself entirely in industry. However, my views changed when I was invited to lunch with one of my professors, Sandip Roy, who encouraged me to consider graduate school and gave me a peek at what it is all about.

After I graduated I got a software security job at Boeing, where I met and later worked for a leader that encouraged me to finally apply for admission to a PhD program: Ron Gabel. Ron gave me the opportunity to travel to universities and conferences to establish research collaborations, and it was through these engagements that I decided I wanted to pursue a PhD at Carnegie Mellon. Shortly before I left for Carnegie Mellon, a PhD Candidate named Bruno Amizic joined our team and offered his wisdom regarding keeping a foot in both academia and industry. Ron and Bruno have been my regular companions and supporters on this journey.

Carnegie Mellon is bursting at the seams with influential people, many of whom have had an impact on me and my time here. Bill Scherlis and Jonathan Aldrich have supervised and guided my research progress throughout the entire process. They have both given me unique opportunities outside of typical research experiences to make connections, learn how research is managed, and generally understand how the sausage is made. I can never repay the trust they have placed in me. Josh Sunshine has acted as a third advisor; I cannot emphasize enough the impact he has had on the research in this thesis. Josh has a keen eye for both message and methodology that has improved my work well beyond what I could have achieved without his guidance. I am also thankful for my co-authors, without whom many of the chapters in this thesis would be worse off:

1. **Chapter 2:** Adam Sales, Benjamin Chung, and Joshua Sunshine. Published in "PeerJ": `https://peerj.com/articles/cs-43/`.
2. **Chapter 3:** Zach Coker, Tianyuan Ding, Claire Le Goues, and Joshua Sunshine. Published in "Annual Computer Security Applications Conference" 2015.
3. **Chapter 4:** Jonathan Aldrich, William Scherlis, and Joshua Sunshine
4. **Chapter 5:** William Scherlis and Jonathan Aldrich. Published in "Symposium and Bootcamp on the Science of Security" 2014.

Many others have motivated, inspired, and educated me including office mates I have shared long discussions with on many topics: Ben Towne and Gabriel Ferreira in particular. I give extra thanks to Ben and Vishal Dwivedi,

both of whom have patiently reviewed several drafts of this document to improve my clarity and reduce the quantity of mistakes. Mary Shaw taught me foundational skills in research writing that I will always rely on. Charlie Garrod showed me what true passion in teaching is and what it can accomplish. Christian Kastner, Claire Le Goues (the best person at venting frustration I have ever met!), and Stephanie Balzer have all improved my communication skills through careful reviews of early paper drafts and practice talks. Others have pushed me to work harder while managing to also keep me social: Michael McCord, Sven Stork, Thomas LaToza, Alex Corn, Ivan Ruchkin, Moses James, Nathan Fulton, David Naylor, Ilari Shafer, Roykrong Sukkerd, Ashwini Rao, Jason Tsay, Mauricio Soto, Darya Kurilova, Blase Ur, Hanan Hibshi, Laurie Jones, Peter Landwehr, Kenny Joseph, Hemank Lamba, Alain Forget, Sudhi Wadkar, Vittorio Perera, and Daniel Smullen. I will not even try to list the many people I have met through service in Dec/5 that could easily make this list as well.

I thank my family, particularly my father. My father developed a knack for tinkering with electronics as a child and never gave it up. I credit him with inspiring my passion for learning about and contributing to the advancement of technology. My brothers have brought me a constant sense of support, multiple nieces and nephews to watch grow up, and have taken on the burden of hosting me and my fiancée, Erin, for several holidays. My fiancée's parents also deserve acknowledgment for hosting us on several visits and for supporting Erin while she finished law school and sought employment.

Finally, I thank my fiancée, partner, and constant companion for most of graduate school. Erin, I love you and look forward to taking the many remaining steps in life with you.

Contents

Chapter 1

Introduction

Software engineers have long worked to realize a world where software production is primarily a matter of component assembly (Krueger, 1992). While there is still work to be done in this space, the software engineering community has made notable progress in this effort. Building software through component assembly measurably reduces development risk (Gabel and Su, 2010; Lim, 1994). Indeed, many modern application developers quite reasonably refuse to incur the costs of writing a component from scratch, making it robust, and maintaining it, instead opting to acquire a component that fulfills some or all of their requirements. This outsources the development risk to the component author, allowing the developers to fill in gaps and focus on their core competencies. For example, web application developers do not typically write the code to send, receive, and process HTTP requests. Instead, they use a framework that performs these operations for them. Similarly, developers do not worry about writing the code to protect their site's communication channels because there are already popular libraries that do this. However, as the common saying goes, "There is no such thing as a free lunch." We pay for decreased development risk by increasing our security risk.

1.1 The State of Practice: Loose Supply Chains

Large manufacturing companies, such as Boeing, Lockheed Martin, and Airbus, maintain strong ties to their physical parts suppliers through contracts, on-site inspections, and other practices aimed at ensuring first-time and continued quality. The architectures of these relationships aim to ensure suppliers will solve problems quickly and at little expense to the buyer. In spite of these arrangements, manufacturers still have problems with defective (Knauth, 2011) and counterfeit parts (Capaccio, 2011; Lim, 2011; McCormack, 2012). While the software community also has extensive supply chains, in general, most software agreements firmly favor the supplier over the buyer.

Oracle, a supplier of enterprise software, recently censored a blog post by one of their executives (Davidson, 2015). This post decried even the automated inspection of their software by customers as a violation of Oracle's intellectual property rights. The ability

to ensure quality marginally improves when we reuse open source components because we avoid these types of legal and political barriers. However, we are still stuck with a seemingly intractable problem: It is extremely difficult to understand what these components do or could do. They may contain unwanted behavior, vulnerabilities, or even malicious attacks. When problematic external code is incorporated into an application, its vulnerabilities or malice become the application's.

Consider the Spring Framework,[1] which is commonly used to construct web applications in Java. This framework contains hundreds of features implemented in more than a million lines of code (OpenHUB, 2015b). How do we trust or verify this quantity of code from the standpoint of even common security attributes, much less all of its possible behaviors? This is not an isolated case. OpenSSL,[2] a library commonly used to authenticate communicating parties and secure their communication channel, constitutes 450,000 lines of code (OpenHUB, 2015a). Most of this code is written in C, which provides essentially no guarantees about the security attributes of software it is used to produce. OpenSSL has recently suffered from a number of serious vulnerabilities (OpenSSL, 2015), some of which could not be detected using existing analysis tools without *a priori* knowledge of the problem (Chou, 2014). These are not isolated problems: An entire thesis-length document could be written documenting similar cases (Neumann, 1995).

To complicate matters, many of the components we use contain their own supply chains. Firefox, for example, depends on a third party GUI toolkit called GTK+. GTK+ in turn depends on a third party 2D drawing library known as Cairo, which itself depends on yet another third party component for picture parsing known as libpng. This is a small subset of Firefox's full supply chain. If trusting and verifying one component from one supplier is often intractable, how do we deal with the even greater degree of complexity often seen in practice?

1.2 Sandboxes: A Path Forward

The complexity present in software supply chains currently creates an uncomfortable situation: Developers must use components (1) that they cannot verify and (2) that they cannot and do not fully trust. Both of these gaps create security risk. When we must use a component we cannot trust or verify we are generally left with one remaining option to defend against unwanted behaviors: encapsulate it. Sandboxes emerged as a promising software protection approach with the publication of Wahbe et al. (1993)'s Software-based Fault Isolation.[3] Sandboxes are encapsulation mechanisms that impose a security policy on software components. Instead of trusting or verifying the component, we trust or verify the relatively simple sandbox. Sandboxes intermediate between an encapsulated component and a surrounding system, but in ways that are distinct from a gateway, diode, or other regulatory device between peer components or systems as seen in system-of-systems environments. Applying a sandbox often leads to an outcome where the behaviors of the

[1] `http://projects.spring.io/spring-framework/`
[2] `https://www.openssl.org/`
[3] This lineage is more thoroughly traced in Chapter 3.

encapsulated component are restricted to a limited set of well understood operations, the component is protected against certain classes of attacks, or both. More abstractly, these cases cover both common reasons to encapsulate a component: It may be malicious or it may be vulnerable to attack.

Given the promise presented by sandboxing technologies, it is little surprise that the last few decades have seen both wide research into new sandboxing techniques and the adoption of sandboxes in practice. In a limited number of cases, sandboxes have been applied to widely used software systems where attacks have been prominent. Google's web browser, Chrome, is sandboxed and also provides a sandbox for running high performance native code as part of a web page. Microsoft sandboxes Internet Explorer and Word, and Adobe Systems sandboxes the Adobe Acrobat family of PDF readers. Modern versions of Linux provide a number of sandboxing mechanisms, including compsec, AppArmor, and SELinux. Some Linux distributions, such as Debian, supply security policies to use with these sandboxes to encapsulate risky applications such web servers, browsers, and email clients.

These sandboxed applications generally have several points in common: they are extremely complicated, with code bases well into the hundreds of thousands if not millions of lines of code[4], and, aside from perhaps Microsoft, all of their vendors make use of third-party code, thus introducing supply chain security risks. Furthermore, aside from Chrome, all of these applications were subject to hundreds of attacks targeting their parsers and renders (the components sandboxed in each of these cases to protect the rest of the system from their vulnerabilities). Google decided to sandbox Chrome citing the many attacks against popular browsers of the time (2008), none of which were sandboxed. In short, all of these applications have been sandboxed because their security states were so poor that applying a sandbox became a necessity. This raises the question: Why are sandboxes not ubiquitous given the pervasiveness of complex software systems and components?

Firefox provides one potential clue. As of late 2015, Firefox remains one of the last unsandboxed mainstream web browsers. One obvious way to sandbox Firefox is to focus on separating out and sandboxing the components in which we have the least confidence. In Firefox, user built addons are an obvious target for sandboxing because anyone can write them in JavaScript, existing review processes to ensure addons are correct and not malicious are relatively weak, and addons can currently alter the execution of the browser in almost any way they like. Unfortunately, the addons are highly coupled with the rest of the browser. Mozilla cannot sandbox Firefox without placing a well-defined API between the browser and addons to decouple the two enough that one sandbox can be imposed on the addons and another (or no) sandbox can be applied to the rest of the browser. Without this decoupling, the browser and addons must have the same sandbox, which does little to prevent an addon from compromising the browser. Furthermore, coupling ensures the security policy imposed is either under-constrained for the addons or over-constrained for the entire browser. This coupling concern is often cited by Mozilla when they explain why Firefox is behind all other popular browsers in terms of isolation, although they are

[4]Chromium, the open source subset of Google Chrome, currently contains 7.5 million lines of code.

making progress in catching up (Mozilla Security Team, 2013).[5] As shown by Firefox and other cases discussed below, coupling can be a serious impediment to sandbox adoption in practice. What other hurdles exist?

1.3 Sandboxes and Unfulfilled Promises

Given the compelling promises sandboxes exhibit, we would like to remove hurdles where possible to encourage adoption. Sandboxes have long been a promising avenue to **decrease security risk**[6] by redirecting the verification and trust problem from components to trustworthy encapsulation mechanisms. This redirection promises to **decrease complexity** for engineers making software development trade-offs.

The software security community has long pursued trustworthy security measures through a strategy of using small, easily analyzed components to act as Trusted Computing Bases (TCB) (Rushby, 1981). Instead of having to trust full computing systems, we only need to trust a much smaller set of code (the TCB) where the scope leaves us better prepared to establish trust. Sandboxes are often TCBs[7] (although not all TCBs are sandboxes): They perform a relatively small set of specialized operations that *should* be easy to establish trust in through inspection, testing, program analysis, and even proof. This **reduces risk** by allowing us to mitigate vulnerabilities and approach the ideal of least privilege (Saltzer and Schroeder, 1975) through the use of small, trustworthy mechanisms.

Small, trustworthy mechanisms *should* **decrease complexity** for engineers developing secure systems under two conditions: (1) the mechanisms are easily applied locally to specific problem areas (e.g. Firefox addons), and (2) are simple enough to understand that they can be applied effectively (e.g. policies are neither over- or under-constrained and are applied to the right set of code). It is often stated that "complexity is the enemy of security"[8] because complexity leads to component misuse and bugs. Encapsulation mechanisms that cannot be easily narrowly targeted lead to cases like Firefox's where applications must be re-architected to encapsulate just untrusted components. (However, the architectures of encapsulated components and their host applications can equally contribute to this failure.) Mechanisms that are hard to understand get accidentally applied incorrectly and thus do not lead to **reduced risk**.

Given the hurdles discussed to this point, to what degree are the promises of **decreased security risk** and **decreased complexity** fulfilled? This thesis takes a big picture approach to answer this question. We analyze the sandboxing landscape, where the majority of existing work has focused on creating or enhancing encapsulation techniques and evaluating each in isolation. In particular, we analyze the literature and study practical,

[5]Mozilla's encapsulation exercise is still not complete two years after the interview cited above. Current status is available at: `https://wiki.mozilla.org/Security/Sandbox`.

[6]Bold phrases in this chapter are intended to help the reader follow the consistent motivation behind this work as we introduce the topic of this thesis.

[7]A sandbox is not a TCB if it is being used for purposes other than security or if it is not being utilized as a primary security mechanism.

[8]While the origins of this phrase are not clear, it seems to have been popularized by Bruce Schneier e.g. `https://www.schneier.com/news/archives/2001/09/three_minutes_with_s.html`.

mainstream mechanisms to explore the current baseline for sandboxing's ability to **reduce risk** without falling prey to excessive **complexity**. After exploring the baseline, which helps us establish barriers to sandbox adoption, we work to reduce barriers in an existing sandbox.

1.3.1 Sandbox Complexity Causes Failures in Practice

Our years of industrial experience with the use of SELinux, AppArmor, and the Java sandbox strongly suggested that sandboxes are often difficult, tedious, and error prone to apply. These challenges represent a few of the technical, social, and human-factors complexity points presented by sandboxes. We empirically confirm sandboxes are difficult and error prone to apply, and evaluate a number of cases where sandbox complexity led to both vulnerabilities and sandbox-deployer misery due to poor usability. Some of these cases were widely reported. For example, the complexity of Google Chrome's sandbox requires intricate re-architecting of existing systems re-using the Chrome sandbox to encapsulate components. Adobe Systems' developers were bitten by this complexity when they failed to understand intricacies that lead to the misapplication of Chrome's sandbox to Reader's renderer. This misapplication lead to severe security vulnerabilities (Delugre, 2012). The deployment of Chrome's sandbox in Google's browser has also been bypassed to compromise the host machine at hacking competitions (Goodin, 2012). Furthermore, Chrome's Native Client sandbox project has paid out several vulnerability bounties and requires intricate porting work to apply to applications in the first place.

While some complexity is necessary, it is up the developers of sandboxing mechanisms to ensure their implementations are verifiable and broadly usable by their intended users (i.e. that any complexity is in fact necessary). In a later chapter we find, empirically validate, and remove unnecessary complexity in the Java sandbox whereby the sandbox can be reconfigured and even disabled at runtime by untrusted code. This complexity led to a number of widely reported exploit campaigns. Turning these "features" off maintains necessary complexity that creates usability issues hampering manual applications of the sandbox, which we overcome through the creation and evaluation of tools, as discussed further below.

1.3.2 Practical Challenges are Mirrored in Sandbox Literature

Given the practical challenges apparent in the sandboxing landscape, does the literature fulfill the promises? We find that the complexity challenges discussed above with respect to mainstream sandboxes are repeated in research sandboxes, where usability is rarely a consideration in any form. This observation is explainable given that usable privacy and security only emerged as a field big enough to have its own venue in 2005.[9] We use a number of statistical techniques to show that the validation strategies used to support claims about sandboxes have not improved since 2004. Sandbox validation is often multifaceted but with

[9]The "Symposium On Usable Privacy and Security" was founded to fill what was perceived as a gap in research pertaining to human-factors in the privacy and security fields: `https://cups.cs.cmu.edu/soups/`.

an emphasis on more subjective evaluation strategies. This is a key point because we lose the benefit of having a small TCB if the TCB is not objectively evaluated against strong security claims. These findings, combined with our complexity findings, demonstrate that there is room to **increase sandbox security** while **decreasing technical complexity** for people applying sandboxes to encapsulate components. We believe that the approaches discussed in this thesis could make engineers more confident about the decisions they make while building secure systems with complex supply chains.

More specifically, our approaches constitute and contribute the following:

- A systematic analysis of sandbox research literature to find general barriers to sandbox adoption.
- An empirical study of how developers use the Java sandbox to find additional barriers to sandbox adoption and to observe how barriers evidenced by the research literature manifest in practice (if at all).
- Tools to mitigate adoption barriers in the Java sandbox that we can not simply remove.
- An architecture for mitigating the consequences of sandbox failures.

1.4 Thesis Approach: Systematic Sandbox Literature Analysis

Chapter 2 systematically analyzes ten years of sandboxing literature from top tier security and systems conferences. This study uses a rigorous qualitative content analysis to interpret relevant papers. We statistically analyze these interpretations to draw conclusions about the *science of sandboxing*. We use this data to define a general and concise definition of the term *sandbox* that we believe is the first to cover most mechanisms that are uncontroversially called "sandboxes." We find a trade-off between the requirements of the application being sandboxed and the requirements of the user applying the sandbox with interesting implications for user-facing complexity in sandboxes. We also substantiate our claim that sandbox validation has not improved in the ten years since 2004 and that sandbox-deployer usability is almost entirely ignored in this space. We provide concrete recommendations for improving sandbox validation. Our recommendations include (1) the use of structure to reduce subjectivity in arguments and (2) the analysis of usability requirements to improve the odds that sandbox deployers will avoid mistakes. Chapter 2 provides broad, detailed evidence of reasons sandboxing promises are unfulfilled. In later chapters we look at specific instances of sandboxes to further explore ways to enhance encapsulation mechanisms.

1.5 Thesis Approach: Tool Assistance for Applying Sandboxes

In Chapter 3 we narrow our focus from sandboxes in general to the Java sandbox. We chose the Java sandbox because it has existed for two decades, it is used in practice to encapsulate potentially malicious or vulnerable components that could damage the rest of the system, and it has suffered a number of problems both in generally containing malice and in being effectively deployed by security conscious users. It is thus a convenient case study to further explore where sandboxes can better **decrease security risk and complexity**.

First, we carefully deconstruct existing Java sandbox exploits and analyze benign Java applications that use the sandbox to compare and contrast malicious versus benign sandbox interactions. We find sharp differences between the operations performed by exploits and benign applications. These differences represent unnecessary complexity in the Java sandbox that we effectively remove through the use of runtime monitors, and without breaking backwards compatibility with benign applications. Our monitors were built solely to evaluate the approach and therefore incur overhead that is not suitable for adoption in practice. This overhead can be reduced to levels acceptable for adoption if the monitors are built into the Java Virtual Machine. In addition to unnecessary complexity, while studying the benign applications we observe a number of sandbox concepts that consistently throw off developers, ranging from severe misunderstandings of the Java security model to simple configuration mistakes applying the sandbox. Furthermore, we observe that the sandbox is rarely deployed, likely due this complexity, and that it is even more rarely used for fine-grained access control. The latter is a stated goal of the sandbox's design (Gong, 2009).

Having fortified the sandbox, we turn to solving developer-facing complexity in Chapter 4. We explain precisely how the sandbox is applied in practice to illustrate the pain suffered by developers that must sandbox entire Java applications or their components. This pain is felt throughout the entire process from defining a security policy, to refining the policy, and finally applying it. Our tools automate most of these steps and double check the user where automation is not possible. Collectively, we believe these various findings and contributions bring Java's sandbox substantially closer to a state acceptable for day-to-day use by regular developers.

1.6 Thesis Approach: Ephemeral Environments

Chapters 3 and 4 show how to reduce unnecessary complexity in the Java sandbox and use tooling to overcome necessary complexity. However, many sandboxes contain necessary complexity that can lead to vulnerabilities in the sandbox itself. In Chapter 5 we define a possibly ideal architecture for mitigating sandbox failures by managing attack surface complexity. Our architecture leverages ephemeral environments to (1) execute potentially vulnerable or malicious computations, (2) carefully persist required outputs, and (3) throw away all remaining side-effects after the computation completes. We realize an approxima-

tion of this ideal using virtual machines running in cloud computing environments. We use this approximation to build a new sandbox for an application that has been successfully attacked in spite of encapsulation (Adobe Reader). We evaluate our sandbox through a field trial and use of a qualitative framework for comparing sandboxes. This framework walks analysts through the process of ranking sandboxes (without quantification) along several dimensions to come to a risk determination in a structured way. The structure ensures rationale is documented and important dimensions of a risk assessment are considered. The qualitative nature of the framework is an acknowledgement that we need some way to measure our progress in **reducing security risk**, but quantitative measurements remain elusive.

1.7 Contributions

The contributions of this thesis focus on sandbox *epistemology*, *architecture*, and *analysis*. The contributions in each category are elaborated below:

Epistemology:

- Systematization of a decade's worth of sandbox research. (Chapter 2)
- A statistical analysis of research sandbox validation techniques and their evolution over a decade. (Chapter 2)
- Derivation of the first concise definition for "sandbox" that consistently describes research sandboxes. (Chapter 2)
- Identification and proposed solutions to (1) an over-reliance on ad hoc arguments for security validation and (2) the neglect of sandbox and policy usability. (Chapter 2)

Architecture:

- Systematization of how sandboxes built for research fit into existing architectures and their outcomes when correctly applied. (Chapter 2)
- The design and evaluation of runtime monitors to fortify the Java sandbox against sandbox escaping exploits. (Chapter 3)
- A small Integrated Development Environment to help developers refine a security policy for subsets of Java applications while accounting for their operational environments. (Chapter 4)
- A proposed sandbox architecture to increase attack surface design flexibility and decrease attack surface extent and consequences of attack success, and an evaluation of the architecture in an industrial case study. (Chapter 5)

Analysis:

- An empirical analysis of use and abuse of the Java sandbox. (Chapter 3)
- A detailed exposition of the complexity involved in manually applying the Java sandbox. (Chapter 4)
- A context sensitive control- and dataflow analysis for recovering required Java permissions from Java bytecode. (Chapter 4)

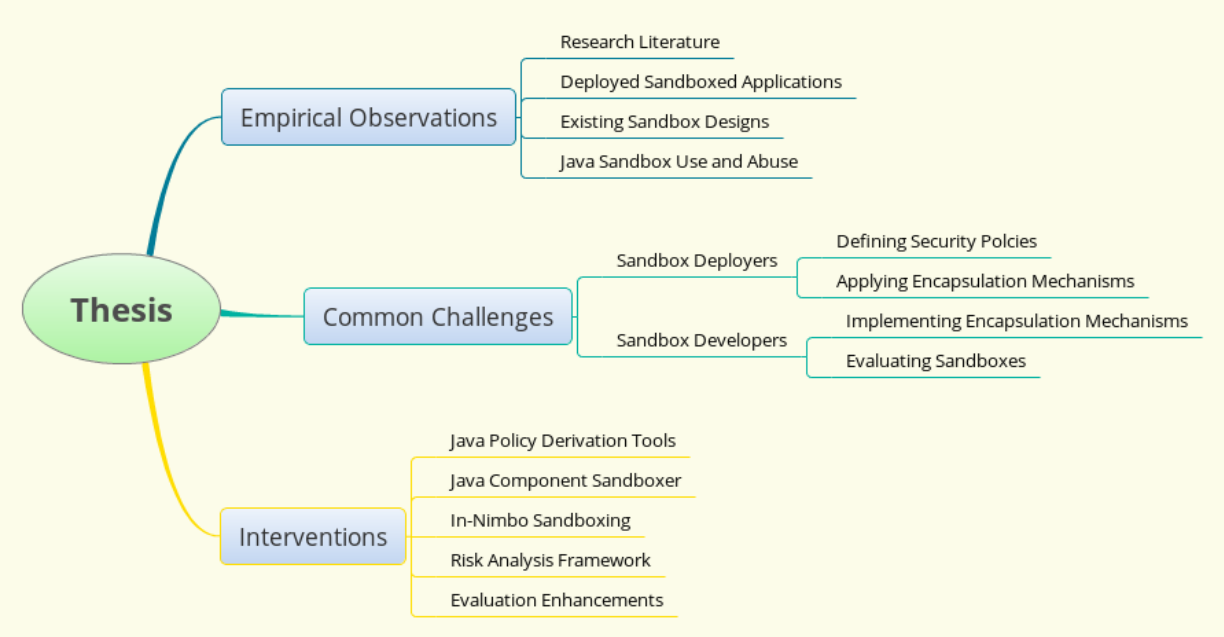


Figure 1.1: A mind map summarizing key features of this thesis.

- A dynamic analysis for recovering required Java permissions from Java applications. (Chapter 4)
- A formalism for merging Java policies. (Chapter 4)
- An algorithm to re-write Java bytecode to apply the sandbox to subsets of Java applications. (Chapter 4)
- Evaluation of the above four contributions through a series of industrial case studies. (Chapter 4)
- A qualitative framework for comparing sandboxes. (Chapter 5)

1.8 Thesis

This thesis exposes problems preventing sandboxes from reaching their potential as software protection mechanisms. Figure 1.1 shows a mind map summarizing important features of this thesis. We take a big a picture look at the sandboxing landscape, only narrowing the field of view when necessary to develop and evaluate solutions to identified problems. We find that while sandboxes have room for improvement, these improvements are both tractable and worth performing as one approach to secure our software future.

Chapter 2

A Systematic Analysis of the Science of Sandboxing[1]

Researchers have spent the last several decades building sandboxes capable of containing computations ranging from fully featured desktop applications to subsets of nearly every kind of application in existence, from third party libraries in Java programs to ads on web sites. Sandboxes have been built to stop memory corruption exploits, ensure control- and data-flow integrity, enforce information flow constraints, introduce diversity where monocultures previously existed, and much more. What more can the research community do to bring value?

In this chapter, we use multidisciplinary techniques from software engineering, statistics, the social sciences, and graph analysis to systematically analyze the sandboxing landscape as it is reflected by five top-tier security and systems conferences. We aim to answer questions about what sandboxes can already do, how they do it, what it takes to use them, what claims sandbox inventors make about their creations, and how those claims are validated. We identify and resolve ambiguity in definitions for "sandbox", systematize ten years of sandbox research, and point out gaps in our current practices and propose ways forward in resolving them.

This systematic analysis was motivated by the lack of a clear common understanding of what the baseline is for the encapsulation option we could rely on when we must use a component we cannot trust or verify. As we will support later, there is a lack of coherence in the sandboxing landscape, which makes it difficult to establish this baseline. Without a baseline, we would not have a basis for identifying the general and pervasive technical and usability barriers to sandbox adoption this thesis aims to mitigate. This chapter's analysis was carried out to overcome the challenges presented by a lack of coherence to ultimately establish a baseline we can rely on for the rest of this thesis.

This chapter contributes the following:

- A multi-disciplinary methodology for systematically analyzing the state of practice in a research domain (Section 2.2).

[1]This chapter was adapted from a paper written with help from Adam Sales, Benjamin Chung, and Joshua Sunshine. It is published in "PeerJ": `https://peerj.com/articles/cs-43/`.

- An improved concise definition for "sandbox" that consistently describes research sandboxes (Section 2.1).
- Systemization of the research sandboxing landscape (Section 2.3).
- Identification of and proposed solutions to (1) an over-reliance on *ad hoc* arguments for security validation and (2) the neglect of sandbox and policy usability (Section 2.4).

We find a general reliance on subjective validation strategies that is concerning. If sandboxes are to be adopted as an effective measure for reducing security risk in the presence of the components that can neither be verified or trusted, we must have strong claims about sandboxes that are backed up by objective science.

2.1 What is a sandbox?

In order to systematically analyze the "sandboxing" landscape we need to clarify the meaning of the term. We reviewed definitions used by practitioners and in papers within the field, both in the substance of the definitions and in their quality as definitions. This section reviews those definitions and establishes a definition for our use here, which we advance as an improved definition for the field.

A definition should be a concise statement of the exact meaning of a word and may be accompanied by narration of some properties implied by the definition. In this case, it should clearly distinguish between mechanisms that are and are not sandboxes. To gain widespread use, a new definition must include all mechanisms that are already widely considered to be sandboxes.

In software security contexts, the term "sandboxing" has grown ambiguous. In an early published use, it described an approach for achieving fault isolation (Wahbe et al., 1993). Discussions where practicing programmers are trying to understand what sandboxing is often fail to achieve a precise resolution and instead describe the term by listing products that are typically considered to be sandboxes or cases where sandboxes are often used.[2] However, we did find cases where attempts were made at a concise and general definition. A representative and accepted StackOverflow answer[3] started with, "In the context of IT security, 'sandboxing' means isolating some piece of software in such a way that whatever it does, it will not spread havoc elsewhere"—a definition that is not sufficiently precise to separate sandboxes from other defensive measures.

Even recently published surveys of sandbox literature have either acknowledged the ambiguity, then used overly-broad definitions that include mechanisms not traditionally considered to be sandboxes (Schreuders et al., 2013a), or have relied entirely on the use of examples instead of a precise definition (Al Ameiri and Salah, 2011). Schreuders writes, "Although the terminology in use varies, in general a sandbox is separate from the access

[2] `http://stackoverflow.com/questions/2126174/what-is-sandboxing`
`http://security.stackexchange.com/questions/16291/are-sandboxes-overrated`
`http://en.wikipedia.org/w/index.php?title=Sandbox_(computer_security)&oldid=596038515`

[3] `http://security.stackexchange.com/questions/5334/what-is-sandboxing`

controls applied to all running programs. Typically sandboxes only apply to programs explicitly launched into or from within a sandbox. In most cases no security context changes take place when a new process is started, and all programs in a particular sandbox run with the same set of rights. Sandboxes can either be permanent where resource changes persist after the programs finish running, or ephemeral where changes are discarded after the sandbox is no longer in use. ..." This definition suffers from three problems. First, it is still overly reliant on examples and thus is unlikely to capture all security mechanisms that are uncontroversially called sandboxes. Along the same lines, characterizations prefaced with, "In most cases...", are not precise enough to reliably separate sandboxes from non-sandboxes. Finally, the comparison to access controls is not conclusive because it does not clarify which, if any, access control mechanisms applied to a subset of running programs are not sandboxes.

In this section we aim to resolve this ambiguity to lay the groundwork for our analysis's inclusion criteria. While this definition serves our purposes, we believe it can strengthen future attempts to communicate scientifically about sandboxes by adding additional precision. We derive a clear, concise definition for what a "sandbox" is using papers that appear in five top-tier security and operating system conferences, selected because their topics of interest are broad enough to include sandboxing papers most years. While we do not attempt to thoroughly validate our definition using commercial and open source sandboxes, it does encompass the tools with which we are most familiar.

We found 101 potential sandboxing papers. Out of these papers, 49 use the term "sandbox" at least once, and 14 provide either an explicit or implicit definition of the term that is clear enough to characterize. The remaining papers that use the term make no attempt at a definition or provide an ambiguous explanation, intertwined with other ideas, and spread over multiple sentences. Within the set of definitions we identify two themes: *sandboxing as encapsulation* and *sandboxing as policy enforcement*.

Sandboxing as encapsulation has a natural analogy: sandboxes on playgrounds provide a place for children to play with indisputably-defined bounds, making the children easier to watch, and where they are less likely to get hurt or hurt someone else. They also contain the sand, thus preventing it from getting strewn across neighboring surfaces. A similar analogy is used in an answer on the Security StackExchange to the question, "What is a sandbox?" Indeed, Wahbe was working to solve the problem of encapsulating software modules (to keep a fault in a distrusted module from affecting other modules) when he popularized the term in this domain.[4]

Table 2.1 lists the definitions we found that we characterize as falling within the theme of sandboxing as isolation. Many of these definitions use the term "isolation," but we prefer the use of *encapsulation*. In Object Oriented Programming, an object *encapsulates* related components and *selectively* restricts access to some of those components. Isolation, on the

[4]While it is clear from at least one publication that the term *sandbox* was used in computer security earlier than Wahbe's paper (Neumann, 1990), many early software protection papers cite Wahbe as the origin of the "sandbox" method (Schneider, 1997; Wallach et al., 1997; Zhong et al., 1997). At least one early commentator felt that this use of the term "sandbox" was merely renaming "trusted computing bases" (TCB) (McLean, 1997). We believe this section makes it clear that sandboxes meet common TCB definitions, but that not all TCBs are sandboxes.

other hand, sometimes refers to a stronger property in which modules use entirely different resources and therefore cannot interfere with each other *at all*. Sandboxed components often need to cooperate to be useful. Cooperation and the idea of disjoint resources are present in Wahbe's original use of the term "sandbox": Wahbe was trying to reduce the communication overhead present in hardware fault isolation by instead creating software domains that run in the same hardware resources, but that do not interfere when faulty. One potential counterpoint to our use of "encapsulation" is that the term typically is used to refer to cases where the inside (e.g. of an object) is protected from the outside, but sandboxes often protect the external system from the contents of the sandbox. While this is a fair point, this chapter does discuss sandboxes that protect their contents from the outside and sandboxes exist that simultaneously defend the inside from the outside and *vice versa* (Li et al., 2014a). Given these points, we maintain that encapsulation's recognition of cooperation is important enough to use the term over isolation. Nevertheless, we retain the use of *isolation* when discussing existing definitions.

Table 2.2 presents seven quotes that discuss sandboxing in terms of restrictions or policy enforcement. These definitions reflect different dimensions of the same idea: A *security policy* can state what is allowed, verboten, or both. The "sandbox" is the subject that enforces the policy or "sandboxing" is the act of enforcing a policy. In short, these quotes cast *sandboxing as policy enforcement*.

Careful inspection of our definition tables shows that the same technique, Software-based Fault Isolation (SFI), appears in both tables. Zhang explicitly states that hardening is not used in SFI, but McCamant very clearly refers to operations being "allowed" and the existence of a policy. While it could seem that the *sandboxing as isolation* and *sandboxing as policy enforcement* camps are disjoint, we claim they are talking about different dimensions of the same idea. Isolation refers to the *what*: An isolated environment where a module cannot do harm or be harmed. Policy enforcement typically refers to the *how*[5]: By clearly defining what is or is not allowed. To use an analogy, we often sandbox prisoners when we place them in a cell. We isolate them by moving them away from everyone else and placing them in a specific, bounded location, then we impose a security policy on them by imposing curfews, monitoring their communications with the outside world, etc. We resolve ambiguity in the use of the term "sandbox" by combining these themes:

Sandbox An encapsulation mechanism that is used to impose a security policy on software components.

This definition concisely and consistently describes the research sandboxes we identify in the remainder of this chapter. It intentionally leaves ambiguity about whether the inside or outside of the sandbox is protected to remain consistent with the discussion of the various approaches above.

Table 2.1: Definitions that speak about "sandboxing" in terms of isolation.

Reference	**Quote**
(Zhang et al., 2013)	"SFI (Software(-based) Fault Isolation) uses instruction rewriting but provides isolation (sandboxing) rather than hardening, typically allowing jumps anywhere within a sandboxed code region."
(Zeng et al., 2013)	"It is a code-sandboxing technique that isolates untrusted modules from trusted environments. ... In SFI, checks are inserted before memory-access and control-flow instructions to ensure memory access and control flow stay in a sandbox. A carefully designed interface is the only pathway through which sandboxed modules interact with the rest of the system."
(Geneiatakis et al., 2012)	"Others works have also focused on shrinking the attack surface of applications by reducing the parts that are exposed to attack, and isolating the most vulnerable parts, using techniques like sandboxing and privilege separation."
(De Groef et al., 2012)	"Isolation or sandboxing based approaches develop techniques where scripts can be included in web pages without giving them (full) access to the surrounding page and the browser API."
(Cappos et al., 2010a)	"Such sandboxes have gained widespread adoption with web browsers, within which they are used for untrusted code execution, to safely host plug-ins, and to control application behavior on closed platforms such as mobile phones. Despite the fact that program containment is their primary goal, flaws in these sandboxes represent a major risk to computer security."
(Reis et al., 2006)	"Wagner et al. use system call interposition in Janus to confine untrusted applications to a secure sandbox environment."
(Cox et al., 2006)	"Our work uses VMs to provide strong sandboxes for Web browser instances, but our contribution is much broader than the containment this provides."

Table 2.2: Definitions that speak about "sandboxing" in terms of policy enforcement.

Reference	**Quote**
(Xu et al., 2012)	"We automatically repackage arbitrary applications to attach user-level sandboxing and policy enforcement code, which closely watches the applications behavior for security and privacy violations such as attempts to retrieve a users sensitive information, send SMS covertly to premium numbers, or access malicious IP addresses."
(Chandra et al., 2011)	"The re-executed browser runs in a sandbox, and only has access to the clients HTTP cookie, ensuring that it gets no additional privileges despite running on the server."
(Politz et al., 2011)	"ADsafe, like all Web sandboxes, consists of two interdependent components: (1) a static verifier, called JSLint, which filters out widgets not in a safe subset of JavaScript, and (2) a runtime library, adsafe.js, which implements DOM wrappers and other runtime checks."
(Tang et al., 2010)	"Fundamentally, rule-based OS sandboxing is about restricting unused or overly permissive interfaces exposed by todays operating systems."
(Sun et al., 2008)	"Sandboxing is a commonly deployed proactive defense against untrusted (and hence potentially malicious) software. It restricts the set of resources (such as files) that can be written by an untrusted process, and also limits communication with other processes on the system."
(McCamant and Morrisett, 2006)	"Executing untrusted code while preserving security requires that the code be prevented from modifying memory or executing instructions except as explicitly allowed. Software-based fault isolation (SFI) or sandboxing enforces such a policy by rewriting the untrusted code at the instruction level."
(Provos, 2003)	"For an application executing in the sandbox, the system call gateway requests a policy decision from Systrace for every system call."

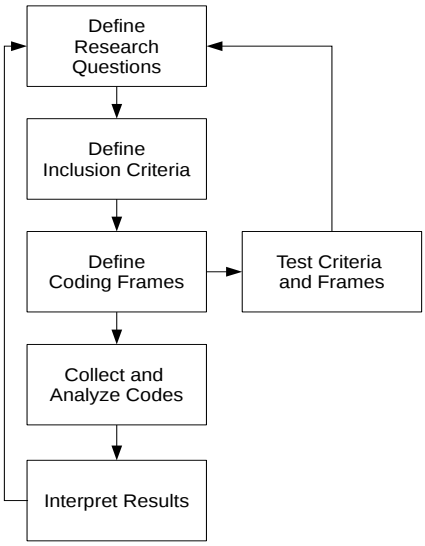


Figure 2.1: The iterative process used to define research questions, build a dataset, and interpret the set to answer the questions. This process is inspired by QCA (Schreier, 2012)

2.2 Methodology

In this section, we discuss the steps we took in order to select and analyze sandboxing papers and the sandboxes they describe. Our methodology is primarily based on the book "Qualitative Content Analysis in Practice" (QCA) (Schreier, 2012). Barnes (2013) provides a succinct summary of the methodology in Section 5.3 of his dissertation. This methodology originates in the social sciences (Berelson, 1952; Denzin and Lincoln, 2011; Krippendorff, 2013) and is intended to repeatably interpret qualitative data to answer a set of research questions. Figure 2.1 summarizes the iterative process we used to define our questions, pick and interpret papers (Sections 2.2.1 and 2.2.2), and develop our results (Section 2.2.3).

We decided to use explicit methods (e.g. QCA) in this analysis to reduce ambiguity in our results. We attempt to clearly define both the steps we carried out and the intermediate terms and data used to get to the results. A typical review would skip these steps, leaving a reasonable belief that important data may have been accidentally missed or inconsistently utilized. Such limitations would have made it difficult for us to achieve the higher levels of confidence we need to point out general barriers to sandbox adoption, a primary goal of this thesis. While the reduction in ambiguity is a benefit for the types of questions we want to answer, the strict methods we use limit our ability to make the types of low-level connections between research threads that are often useful in traditional literature reviews.

QCA goes well beyond a systematic literature review (Budgen and Brereton, 2006;

[5]Exceptions potentially exist where sandboxes also verify that a computation does not violate a given security policy before execution.

Kitchenham et al., 2009). While both QCA and systematic reviews require the definition of research questions and repeatable processes for collecting source material, reviews stop short of detailed analysis. QCA carries on where reviews end. When performing QCA, researchers define coding frames to clearly and repeatably establish how the source material will be interpreted to answer the research questions. The frames contain codes that summarize blocks of data and definitions for each code. Furthermore, QCA methodologies dictate how the coding frames are to be applied, by segmenting the entirety of the data such that each segment can labeled with at most one code. This ensures that the data is coded without missing relevant data and while reducing the researcher's bias towards some bits of data. Finally, QCA requires researchers to test their full process before carrying out the analysis.[6] Together, these steps allow researchers to reliably and effectively interpret text to answer research questions that are not possible to answer using a purely quantitative analysis. For example, Schreier points out that a quantitative analysis can determine how many women appear in magazine advertisements relative to men, but a qualitative analysis (e.g. QCA) is required to determine whether or not women are more likely to be placed within trivial contexts than men in those ads (Schreier, 2012, p. 2).

The sandboxes we describe in this chapter were selected from the proceedings of five conferences: IEEE Symposium on Security and Privacy (Oakland), Usenix Security, ACM Conference on Computer and Communications Security (CCS), ACM Symposium on Operating System Principles (SOSP), and Usenix Symposium on Operating System Design and Implementation (OSDI). We restricted our selection to particular conferences to improve reproducibility—because of this choice, the set of papers evaluated against our inclusion criteria is very well defined. To select these conferences, we collected all of the sandboxing papers we were aware of and the selected five venues contained far more sandboxing papers than any other venue.[7]

The selected conferences are widely regarded as the top-tier conferences in software security and operating systems.[8] Therefore, our data reflects the consensus of large communities.

Table 2.3 presents our twelve research questions, the areas each question attempts to illuminate, and a comprehensive list of their answers as manifested by our paper corpus. We derived an initial set of questions by considering which broad aspects of sandboxes

[6]We followed the QCA methodology specified by Schreier with one major deviation. We did not segment the text because the vast majority of the content in the papers is irrelevant to our needs. Most uses of QCA attempt to capture content of a text in its entirety. This was not our goal so we analyzed text more selectively.

[7]Based on earlier criticism of the paper version of this chapter, we reevaluated our data set by looking at the past four years of proceedings at unselected venues such as the USENIX Annual Technical Conference (ATC), Programming Language Design and Implementation (PLDI), and Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). These venues contained fewer sandboxing papers than our selected venues, and those that appeared were not significantly different in form or content from those in selected venues. In fact, with rare exceptions, the sandboxing papers at the unselected venues were written by the same authors as one or more paper in our data set.

[8] `http://www.core.edu.au/index.php/conference-rankings`
`https://personal.cis.strath.ac.uk/changyu.dong/ranking.html`
`http://faculty.cs.tamu.edu/guofei/sec_conf_stat.htm`
`http://webdocs.cs.ualberta.ca/~zaiane/htmldocs/ConfRanking.html`

are poorly understood and where better understanding may change how the community performs research in this space. As a result, the questions are necessarily biased by our own backgrounds and personal experiences. In particular, this led to an emphasis on questions about how mechanisms and policies are derived, applied, and evaluated. We added questions while we performed the analysis when we found that we had the data to answer new and interesting questions. Overall, these questions aim to capture a comprehensive snapshot of the current state of sandboxing research, with an emphasis on where sandboxes fit into the process of securing software systems, what policies are enforced and how they are defined and constructed, and what claims are made about sandboxes and how those claims are validated.

2.2.1 Picking Papers

We selected papers from 10 years worth of proceedings at the five conferences mentioned above. We decided whether a paper was included in our sample based on rigorous inclusion criteria so the process of including/excluding papers is repeatable. The most important criterion is that the paper describes a sandbox that meets the definition given in Section 2.1. The remaining criteria were added as we carried out the study to exclude papers that are incapable of answering the research questions and to clarify relevant nuances in our definition.

Papers were included if they met the following criteria:

- The paper documents the design of a novel tool or technique that falls under the *sandbox* definition
- The paper is a full conference paper
- The paper is about an instance of a sandbox (e.g. not a component for building new sandbox tools, theoretical constructs for sandboxes, etc.)
- Techniques are applied using some form of automation (e.g. not through entirely manual re-architecting)
- A policy is imposed on an identifiable category of applications or application subsets
  - The policy is imposed locally on an application (e.g. not on the principal the application executes as, not on network packets in-transit, etc.)
  - The category encompasses a reasonable number of real-world applications (e.g. doesn't require the use of (1) a research programming language, (2) extensive annotations, or (3) non-standard hardware)

We gathered papers by reading each title in the conference proceedings for a given year. We included a paper in our initial dataset if the title gave any indication that the paper could meet the criteria. We refined the criteria by reviewing papers in the initial dataset from Oakland before inspecting the proceedings from other venues. We read the remaining papers' abstracts, introductions, and conclusions and excluded papers as they were being interpreted if they did not meet the criteria. We maintained notes about why individual

Table 2.3: Our research questions, the areas each question attempts to illuminate, and potential answers. The answers are codes in the content analysis process we apply. Answers are not necessarily mutually exclusive. Definitions for the terms in this table appear in our coding frames with examples: `http://goo.gl/cVHdzZ`.

Question area	**Question**	**Possible answers**
Sandbox Lifecycle	What architectural components enforce the policies?	System (e.g. OS), Application, Application Host
	How and when are policies imposed?	Statically, Dynamically, Hybrid
Security outcomes	What resources do the sandboxes protect?	Memory, Code/Instructions, Files, User Data, Communications
	Which components do the sandboxes protect?	Component, Application, Application Class
	At what point will sandboxes catch exploits?	Pre-exploit, Post-exploit
Effort and applicability	What must be done to apply the sandboxes?	Nothing, Select Pre-made Policy, Write Policy, Run Tool, Install Tool
	What are the requirements on sandboxed components?	None, Source Code, Annotated Source Code, Special Compiler, Compiler-introduced Metadata, Sandbox Framework/Library Components
Policy provenance and manifestation	Who defines policies?	Sandbox Developer (Fixed), Sandbox User (User-defined), Application Developer (Application-defined)
	How are policies managed?	Central Policy Repository, No Management
	How are policies constructed?	Encoded in Sandbox Logic, Encoded in Application Logic, User Written
Research claims and validation	What claims are made about sandboxes?	Performance, Security, Applicability
	How are claims validated?	Proof, Analytical Analysis, Benchmark Suite, Case Studies, Argumentation, Using Public Data
	How are sandboxes released for review?	Source Code, Binaries, Not Available

papers were excluded from the final set.[9]

2.2.2 Categorizing the Dataset

To interpret papers we developed coding frames[10] where a *category* is a research question and a *code* is a possible answer to the question. To ensure consistency in coding, our frames include detailed definitions and examples for each category and code. Our codes are not mutually exclusive: A question may have multiple answers. We developed the majority of our frames before performing a detailed analysis of the data, but with consideration for what we learned about sandboxing papers while testing the inclusion criteria above on our data from Oakland. We learned that evaluative questions were quite interesting while coding papers, thus frames concerning what claims were made about a sandbox and how those claims were validated became more fine-grained as the process progressed. Whenever we modified a frame, we updated the interpretations of all previously coded papers.

We tested the frames by having two coders interpret different subsets of the Oakland segment of the initial dataset. To interpret a paper, each category was assigned the appropriate code(s) and a quote justifying each code selection was highlighted and tagged in the paper's PDF.[11] While testing, the coders swapped quotes sans codes and independently reassigned codes to ensure consistency, but we did not measure inter-rater reliability. Code definitions were revised where they were ambiguous. While there is still some risk that different coders would select different quotes or assign codes to the same quote, we believe our methodology sufficiently mitigated the risk without substantially burdening the process given the large scope of this effort.

After coding every paper, we organized the codes for each paper by category in a unified machine-readable file[12] (hereafter referred to as the summary of coded papers) for further processing.

2.2.3 Analyzing the Dataset

To summarize the differences and similarities between sandboxing papers, we attempted to identify clusters of similar sandboxing techniques. To do so, we first calculated a dissimilarity matrix for the sandboxes. For category k, let p_{ijk} be the number of codes that sandboxes i and j share, divided by the total number of codes in that category they *could* share. For categories in which each sandbox is interpreted with one and only one code, p_{ijk} is either 1 or 0; for other categories, it falls in the interval $[0, 1]$. Then the dissimilarity between i and j is $d_{ij} = \sum_k (1 - p_{ijk})$. We fed the resulting dissimilarity matrix into a hierarchical agglomerative clustering algorithm (Kaufman and Rousseeuw, 2009) (implemented in `R` with the `cluster` package (Maechler et al., 2014; R Core Team, 2014)). This algorithm begins by treating each sandbox as its own cluster, and then iteratively merges

[9] Our full list of papers with exclusion notes is available at: `https://goo.gl/SfcN0K`.

[10] Our full coding frames are available at: `http://goo.gl/cVHdzZ`.

[11] A full list of quotes with code assignments is available at: `http://goo.gl/d3Sf5J`.

[12] The summarized version of our dataset is available at: `http://goo.gl/q0cGV8`. This spreadsheet was converted to a CSV to perform statistical and graph-based analyses.

the clusters that are nearest to each other, where distance between two clusters is defined as the average dissimilarity between the clusters' members. The agglomerative clustering process is displayed in dendrograms. We stopped the agglomerative process at the point at which there were two clusters remaining, producing two lists of sandboxes, one list for each cluster. To interpret the resulting clusters, we produced bar charts displaying the code membership by cluster. We conducted this analysis three times: once using all of the categories to define dissimilarity, once using using all categories except those for claims, validation, and availability, and once using the validation categories. We do not present the plots from the analysis that ignored claims, validation, and availability because it did not produce results different from those generated using all categories.

We conducted correlational analyses to learn whether sandbox validation techniques have improved or worsened over time, or whether sandbox publications with better (or worse) validation received more citations. The validation codes were ordered in the following way: proof > analytical analysis > benchmarks > case study > argumentation > none. This ordering favors validation techniques that are less subjective. While it is possible for a highly ranked technique to be applied less effectively than a lower ranked technique (e.g. a proof that relies on unrealistic assumptions relative to a thorough case study) this ranking was devised after coding the papers and is motivated by the real world applications of each technique in our dataset. Each claim type (security, performance, and applicability), then, was an ordinal random variable, so rank-based methods were appropriate. When a sandbox paper belonged to two codes in a particular validation category, we used its highest-ordered code to define its rank, and lower-ordered codes to break ties. So, for instance, if paper A and paper B both included proofs, and paper A also included benchmarks, paper A would be ranked higher than paper B. To test if a claim type was improving over time, we estimated the Spearman correlation (Spearman, 1904) between its codes and the year of publication, and hence tested for a monotonic trend. Testing if papers with better validation, in a particular category, received more citations necessitated accounting for year of publication, since earlier papers typically have higher citation counts. To do so, we regressed paper citation rank against both publication year and category rank. (We used the rank of papers' citation counts as the dependent variable, as opposed to the citation counts themselves, due to the presence of an influential outlier—Terra (Garfinkel et al., 2003). Scatterplots show the relationship between citation ranks and publication year to be approximately linear, so a linear adjustment should suffice.) There was a "validation effect" if the coefficient on the validation measure was significantly different from zero. We conducted four separate regression analyses: one in which citation ranks were regressed on publication year and category ranks of all three validation criteria, and one in which citation ranks were regressed on publication year and security validation only, one in which citation ranks were regressed on publication year and performance validation only, and one in which citation ranks were regressed on publication year and applicability validation only.

We constructed a citation graph using the papers in our set as nodes and citations as edges as a final means of better understanding the sandboxing landscape. We clustered the nodes in this graph using the same clusters found statistically, using the process describe above, and using common topics of interest we observed. The topics of interest are typically

based on the techniques the sandboxes apply (e.g. Control Flow Integrity (CFI), artificial diversity, etc.). We evaluate these clusters using the modularity metric, which enables us to compare the quality of the different categorizations. Modularity is the fraction of edges that lie within a partition, above the number that would be expected if edges were distributed randomly.

2.3 Results

We derived our results from the various statistical clusters of our summary of coded papers, trends explicit in this dataset, and observations made while reading the papers or analyzing our summarized data. As our dataset is public, we encourage readers to explore the data themselves. Note while interpreting the statistical clusters that they are not representative of how papers are related in terms of broad topics of interest. When we applied the statistical clusters to the citation graph of the papers in our set the modularity scores were -0.04 and 0.02 when papers were clustered based on all of the attributes we coded and just validation attributes respectively. These modularity scores mean that the statistical clusters are no better than randomly clustering papers when considering how they cite each other.

These poor modularity scores make sense because authors are much more likely to cite papers that use similar techniques or tackle similar problems than use similar validation strategies. Indeed, the lack of coherence in the sandboxing landscape, caused by authors primarily citing papers that use related techniques, is a prominent reason for carrying out this study in the first place. We confirmed papers are not related through citations due to applied validation strategies by using a modularity measure that ranks the partitions of papers based on validation techniques and topics into overlapping groups (Lázár et al., 2009). This measure allows us to quantify how clusters built based on validation strategies relate to those built based on topics. We measured an overlapping modularity of -0.198, which confirms that partitions built from the validation techniques do not direct citation graph structure. Indeed, when we clustered papers in the citation graph based on topics of interest we observed while interpreting the set, the modularity score, 0.33, is significantly better than a random cluster. The citation graph with topic clusters is shown in Figure 2.2. While these clusters are potentially of sociotechnical interest to the community, we must look at lower-level attributes to understand how sandboxes are to be applied in practice and how they improve the security posture of real systems. The statistical clusters fill that role.

Figures 2.3 and 2.4 show the codes that are members of the fixed policy and user-defined policy clusters respectively when all categories are considered. The dendrogram for these clusters appears in Figure 2.5. Many of our results are interpretations of these charts. Table 2.4 succinctly describes our results per research question and references later sections where more details are found. The remainder of this section presents those details.

Table 2.4: Summary of our research questions and results.

Research Question	Results	Section
Where in a system's architecture are policies enforced?	There is an emphasis on enforcing policies in the operating system or transforming applications to enforce a policy over using application hosts (e.g. language-hosting virtual machines, browsers, etc.).	2.3.1
When are policies imposed?	Static, dynamic, and hybrid strategies are roughly equally favored in all domains but with a slight preference for strictly static or dynamic approaches.	2.3.1
What application resources are protected by sandboxes?	Sandboxes with fixed policies tend to prevent memory corruption or protect properties of application code (e.g. control flow). User-defined policies are correlated with policies that are more diverse and cover the gamut of application-managed resources.	2.3.1
What types of components are protected by sandboxes?	Sandboxes that use fixed policies tend to require the user to target specific components, while those with user-defined policies tend to allow for broader targeting.	2.3.1
At what point in the process of an attack will an exploit violate sandbox policies?	Sandboxes are primarily pro-active by disrupting exploits before a payload can be executed. Where users must define a policy, sandboxes tend to be pro-active in attempting to stop exploits, but also limit the range of possible behaviors a payload can exhibit.	2.3.1
What are the requirements of people applying sandboxes?	Sandboxes that have fewer requirements for people tend to have more requirements for the application. Similarly, having a fixed policy is correlated with more requirements of the application, while user-defined policies are correlated with more requirements of the user.	2.3.2
What are the requirements of components being sandboxed?	Sandboxes with fixed policies most-often require that applications be compiled using a special compiler.	2.3.2
Who defines sandbox policies?	Policies are most often defined by the sandbox developer at design time.	2.3.2
How are policies managed?	Policy management is largely ignored, even where users must write their own policies.	2.3.2
How are policies constructed?	Most policies are hardcoded in the sandbox.	2.3.2
What claims are made about sandboxes?	Applicability to new cases is often the impetus for improving existing techniques, but strong security and better performance are more often claimed.	2.3.3
How are claims validated?	Benchmarks and case studies are the most favored validation techniques for all types of claims. Where security claims are not validated using both benchmarks and case studies, ad-hoc arguments are heavily favored.	2.3.3
In what forms are sandboxes made available for review?	There is a recent slight increase in the release of sandbox source code, but generally no implementation artifacts are made available for review.	2.3.3

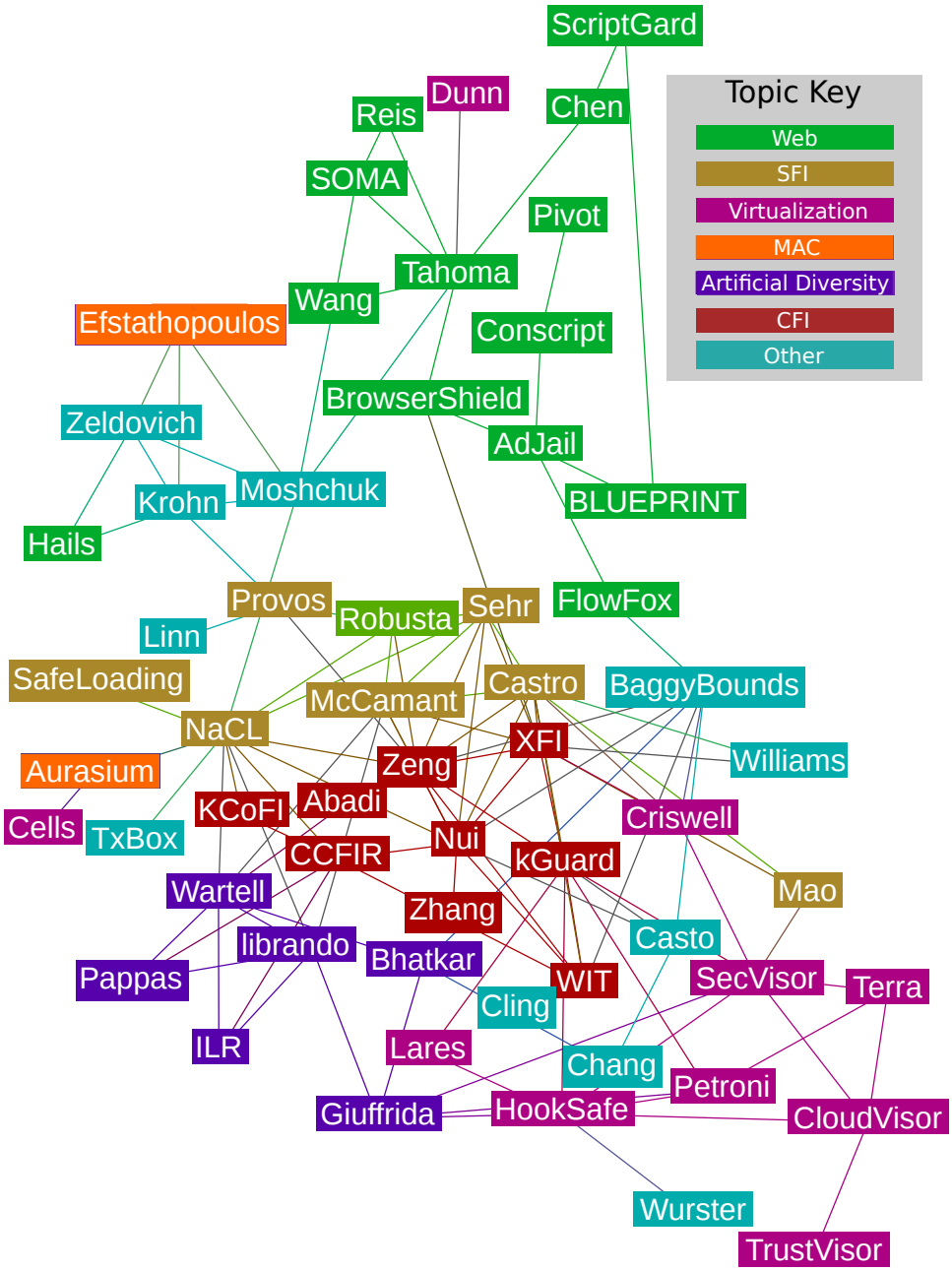


Figure 2.2: The citation graph for the papers in our set. The intention of this chart is to show where citation relationships exist, not to show explicitly which papers cited other papers. The colors represent clusters based on topics of interest (modularity = 0.33). Papers cluster based on topics of interest, not necessarily their technical attributes or validation stratgies, thus we must look at lower level attributes to gain a broad understanding of the sandboxing landscape. Papers that were not linked to any of the other papers in the set are not shown. Categories bridging Mandatory Integrity and Access Control (MI/AC) were collapsed to simply Mandatory Access Control (MAC) for this graph. Our citation data can be found at `http://goo.gl/fgk4LG`.

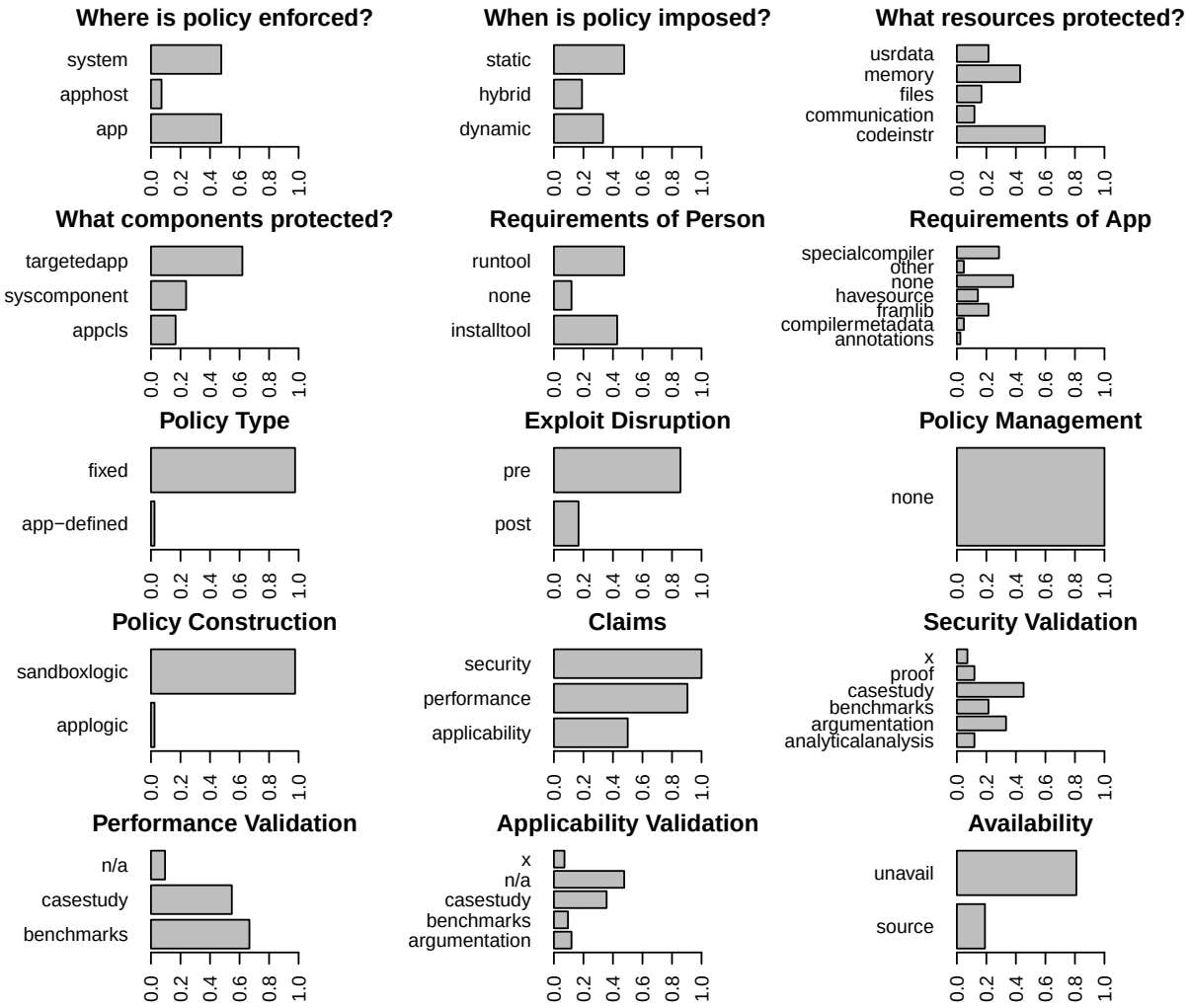


Figure 2.3: Breakdown of the representation of all codes for papers that emphasize fixed policies. Cases where a claim was made but not validated are labeled with an "x".

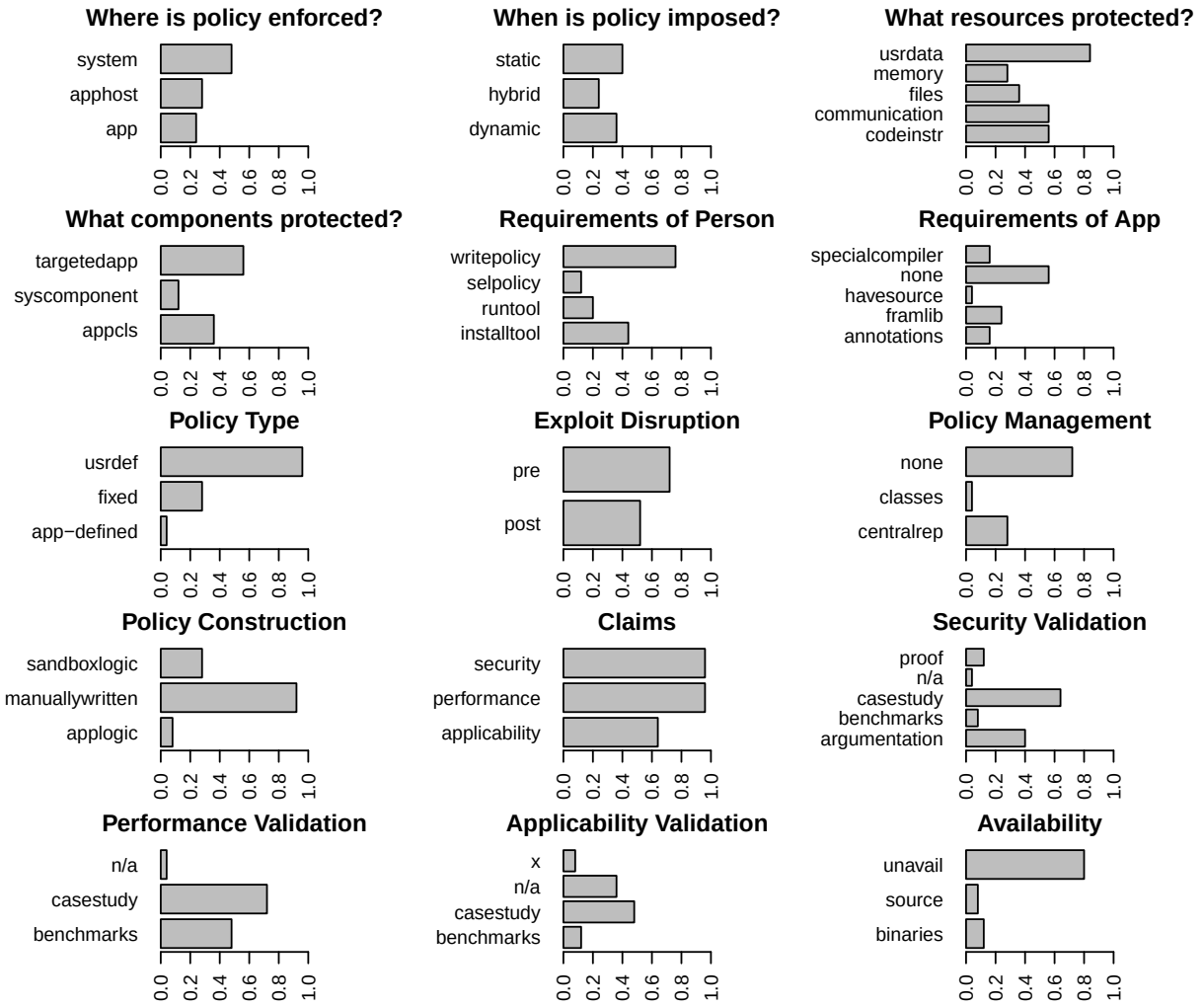


Figure 2.4: Breakdown of the representation of all codes for papers that emphasize user-defined policies. Some sandboxes support a fixed-policy with an optional user-defined policy (e.g. (Siefers et al., 2010a)). Cases where a claim was made but not validated are labeled with an "x".

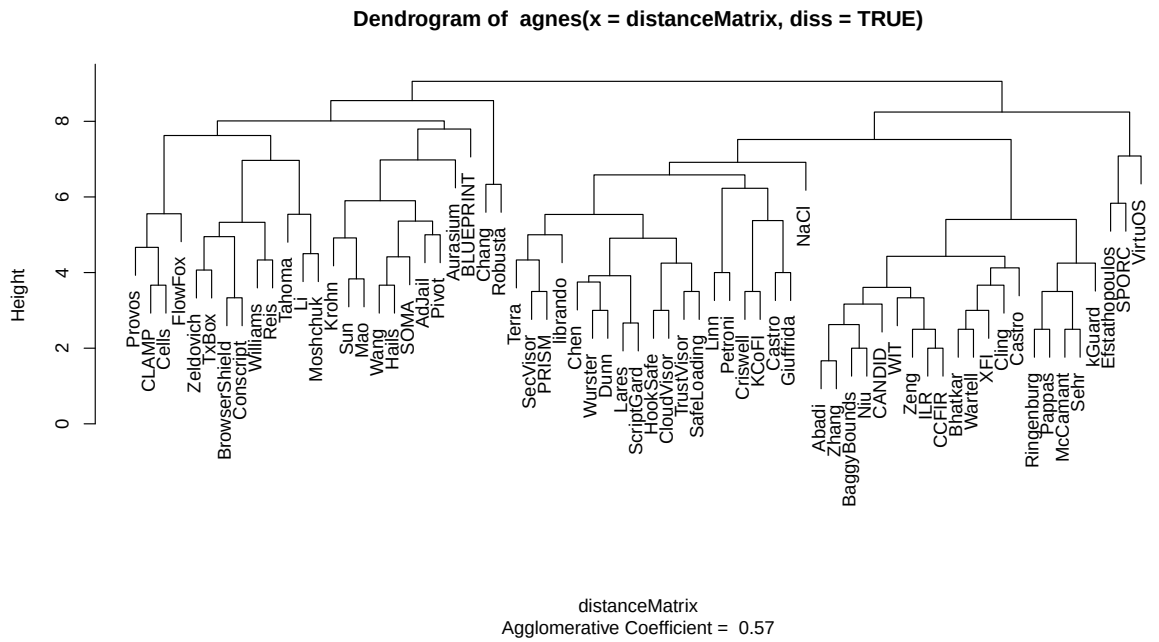


Figure 2.5: A dendrogram displaying the clusters for sandboxing papers taking into account all categories. At the top most level, where two clusters exist, the clusters respectively represent sandboxes that use fixed policies and those that use user-defined policies.

2.3.1 Sandboxes: Building Materials for Secure Systems

Sandboxes are flexible security layers ready to improve the security posture of nearly any type of application. While the deployment requirements and details vary from sandbox to sandbox, collectively they can be applied at many different points in a system's architecture and may be introduced at any phase in an application's development lifecycle, starting with the initial implementation. In fact, sandboxes can even be applied well after an application has been abandoned by its maintainer to secure legacy systems.

In our dataset, the policy enforcement mechanism for a sandbox is always deployed as a system component, as a component of an application host, or by insertion directly into the component that is being encapsulated. While application hosts are becoming more popular as many applications are moved into web browsers and mobile environments, they are currently the least popular place to deploy policy enforcement mechanisms for research sandboxes. Our set includes ten sandboxes where policies are enforced in the application host, twenty-six in the component being encapsulated,[13] and thirty-two in a system component.

We believe that application hosts are less represented because many existing hosts come with a sandbox (e.g. the Java sandbox, Android's application sandbox, NaCl in Google Chrome, etc.). Indeed, all but one of the sandboxes deployed in application hosts are for

[13](Sehr et al., 2010) is counted twice because the enforcement mechanism is spread across the application and its host.

the web, where applications can gain substantial benefits from further encapsulation and there is currently no *de facto* sandbox. The one exception is Robusta (Siefers et al., 2010a), which enhances the Java sandbox to encapsulate additional non-web computations.

System components are heavily represented because any sandbox that is to encapsulate a kernel, driver, or other system component must necessarily enforce the policy in a system component. Fifteen of the sandboxes fall into this category because they are encapsulating either a kernel or hypervisor. The remainder could potentially enforce their policies from a less privileged position, but take advantage of the full access to data and transparency to user-mode applications available to system components. This power is useful when enforcing information flow across applications, when preventing memory corruption, or when otherwise enforcing the same policy on every user-mode application.

Research sandboxes almost universally embed their enforcement mechanism in the application that is being encapsulated when the application runs in user-mode. Application deployment is correlated with fixed policies where modifying the application itself can lead to higher performance and where it makes sense to ensure the enforcement mechanisms exist anywhere the application is, even if the application moves to a different environment. Fixed-policies with embedded enforcement mechanisms are correlated with another important deployment concern: statically imposed policies.

Imposing a policy statically, most often using a special compiler or program re-writer, is advantageous because the policy and its enforcement mechanism can travel with the application and overhead can be lower as enforcement is tailored to the targeted code. There are some cons to this approach. For example, the process of imposing the policy cannot be dependent on information that is only available at run-time and the policy is relatively unadaptable after it is set. Furthermore, because the policies are less adaptable, sandboxes that statically impose security policies typically only encapsulate components that are targeted by the person applying the sandbox. These are cases where dynamic mechanisms shine. Given these trade-offs, it makes sense that papers in our set fall into one of two clusters when all codes are considered: Those that are protecting memory and software code, which are relatively easy to encapsulate with a fixed policy, and those managing behaviors manifested in external application communications or interactions with user-data and files that are more easily encapsulated with an adaptable (typically user-defined) policy.

Generally hybrid deployments are used when the approach is necessarily dynamic but static pre-processing lowers overhead. Sometimes, techniques begin as hybrid approaches and evolve to fully dynamic approaches as they gain traction. For example, early papers that introduce diversity in binaries to make reliable exploits harder to write (e.g. code randomization), tend to rely on compiler-introduced metadata, while later papers did not need the extra help. This evolution broadens the applicability of the sandboxing technique. We observed other techniques such as SFI and CFI evolve by reducing the number of requirements on the application, the person applying the sandbox, or both.

2.3.2 Policy Flexibility as a Usability Bellwether

Requiring more work out of the user or more specific attributes of an application lowers the odds that a sandbox will be applied, thus it is natural that research on specific techniques reduce these burdens over time. We find that the nature of the policy has an influence on how burdensome a sandbox is. About half of sandboxes with fixed policies require the application be compiled using a special compiler or uses a sandbox-specific framework or library. Many fixed-policy sandboxes also require the user to run a tool, often a program re-writer, or to install some sandbox component. In comparison, nearly all sandboxes with flexible policies require the user to write a policy manually, but few have additional requirements for the application. Given the burdens involved in manually writing a security policy, the message is clear—easy to use sandboxes reduce the user-facing flexibility of the policies they impose.

Forty-eight sandboxes, more than two-thirds of our sample, use a fixed policy. In all of these cases the policy itself exists within the logic of the sandbox. In the remaining cases, the policy is encoded in the logic of the application twice (e.g. through the use of the sandbox as a framework), and the remaining seventeen cases require the user to manually write a policy.

In cases where the user must manually write the policy, it would help the user if the sandbox supported a mechanism for managing policies—to ensure policies do not have to be duplicated repeatedly for the same application, to generate starter policies for specific cases, to ensure policies can apply to multiple applications, etc. This type of management reduces the burden of having to manually write policies in potentially complex custom policy languages. Support for the policy writer is also important because the policies themselves can be a source of vulnerabilities (Rosenberg, 2012). Eight out of twenty-six cases where policy management is appropriate offered some central mechanism for storing existing policies, where they could potentially be shared among users. However, none of the papers in our sample list policy management as a contribution, nor do any of the papers attempt to validate any management constructs that are present. However, it is possible that there are papers outside of our target conferences that explicitly discuss management. For example, programming languages and software engineering conferences are more focused on policy authoring concerns and management may therefore be the focus of a paper that appears in one of those conferences. However, we are not aware of any such paper.

2.3.3 The State of Practice in Sandbox Validation

There is little variation in the claims that are made about sandboxes. Most claim to either encapsulate a set of threats or to increase the difficulty of writing successful exploits for code-level vulnerabilities. All but four measure the performance overhead introduced by the sandbox. Thirty-seven papers, more than half, make claims about the types of components the sandbox applies to, typically because the paper applies an existing technique to a different domain or extends it to additional components.

While there is wide variety in how these claims are validated, we observe measurable

patterns. In our data set, proof and analytical analysis were, by far, the least used techniques. The lack of analytical analysis is due to the fact that the technique is primarily useful when the security of the mechanism depends on randomness, which is true of few sandboxes in our set. However, proof appears in two cases: (1) to prove properties of data flows and (2) six papers prove the correctness of a mechanism enforcing a fixed policy. The rarity of proof in the sandboxing domain is not surprising given the difficulty involved. Proof is particularly difficult in cases where one would ideally prove that a policy enforcement mechanism is capable of enforcing all possible policies a user can define, which we did not see attempted. Instead, claims are often validated empirically or in ways that are *ad hoc* and qualitative.

In empirical evaluations, case studies are the most common technique for all claims, often because proof was not attempted and there is no existing benchmark suite that highlights the novel aspects of the sandbox. For example, papers for sandboxes with fixed policies often want to show a particular class of vulnerabilities can no longer be exploited in sandboxed code, thus examples of vulnerable applications and exploits for their vulnerabilities must be gathered or, very rarely, synthesized. When claims were empirically validated, the results were not comparable in fifteen out of sixty-two cases for performance, twenty-two out of forty-two cases for security, and twenty-four out of thirty-one cases for applicability because non-public data was used in the discussed experiments. Non-public data takes the form of unlabeled exploits, undisclosed changes to public applications, and unreleased custom example cases (e.g. applications built using a sandbox's framework where the examples were not released).

Security claims are notoriously difficult to formalize – and even harder to prove, hence the pervasive lack of proof. Many papers instead vet their security claims using multi-faceted strategies, often including both common empirical approaches: case studies and experiments using benchmark suites. However, Figures 2.6 and 2.7 illustrate an interesting finding: When we look at the bottom two clusters in Figure 2.7, where multi-faceted strategies are not used, and cross-reference the papers named in the corresponding clusters in Figure 2.6 with the underlying summary of coded papers, we find that in twenty-nine papers authors simply pick one empirical tactic and argue that their claims are true. Argumentation in this space is problematic because all of the arguments are *ad hoc*, which makes evaluations that should be comparable difficult to compare at best but more often incomparable. Furthermore, we observed many cases where arguments essentially summarize as, "Our sandbox is secure because the design is secure," with details of the design occupying most of the paper in entirely qualitative form. Not only are these types of arguments difficult to compare in cases where sandboxes are otherwise quite similar, it is even harder to see if they are complete in the sense that every sub-claim is adequately addressed.

Our correlational analyses show no significant trends in security or applicability analyses, however performance validation has improved over time. Table 2.5 summarizes the Spearman correlations and their p-values per validation category. Spearman correlations fall in the range [-1,1], where a value of 0 is interpreted as no correlation, positive values show a positive correlation, and negative values a negative correlation. The magnitude of the coefficient grows towards 1 as time and the validation rank become closer to perfect

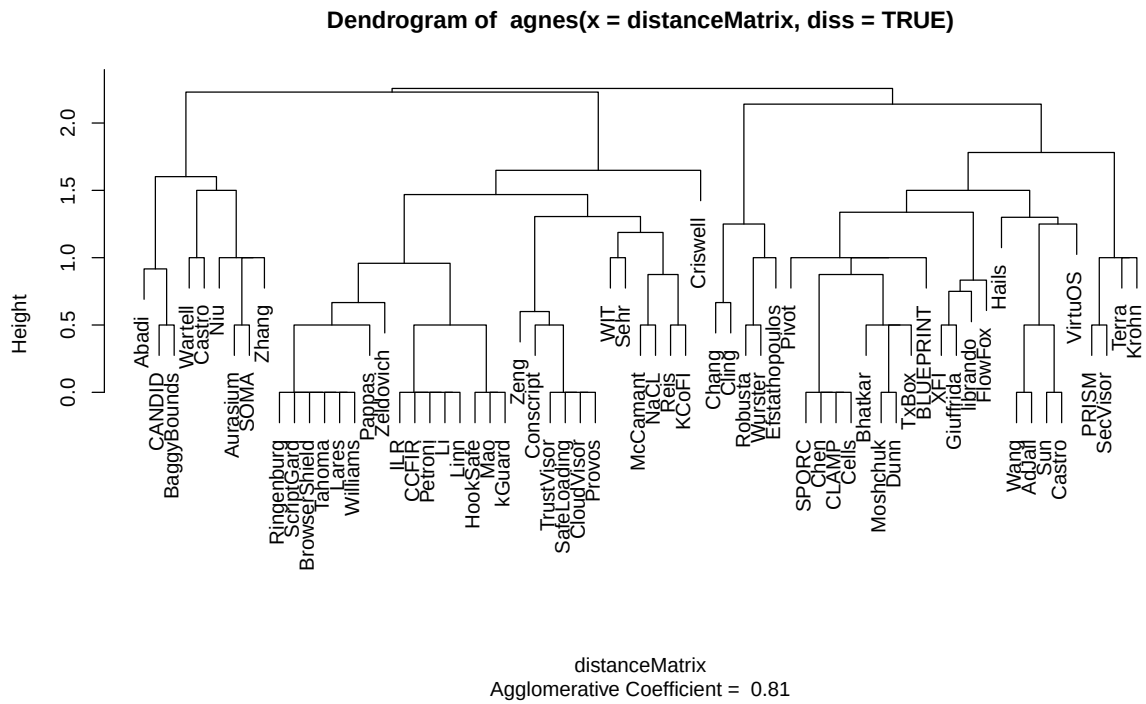


Figure 2.6: A dendrogram displaying the clusters for sandboxing papers taking into account validation categories. At the top most level, where two clusters exist, the clusters respectively represent sandboxes that emphasize multi-faceted empirical security validation and those that do not.

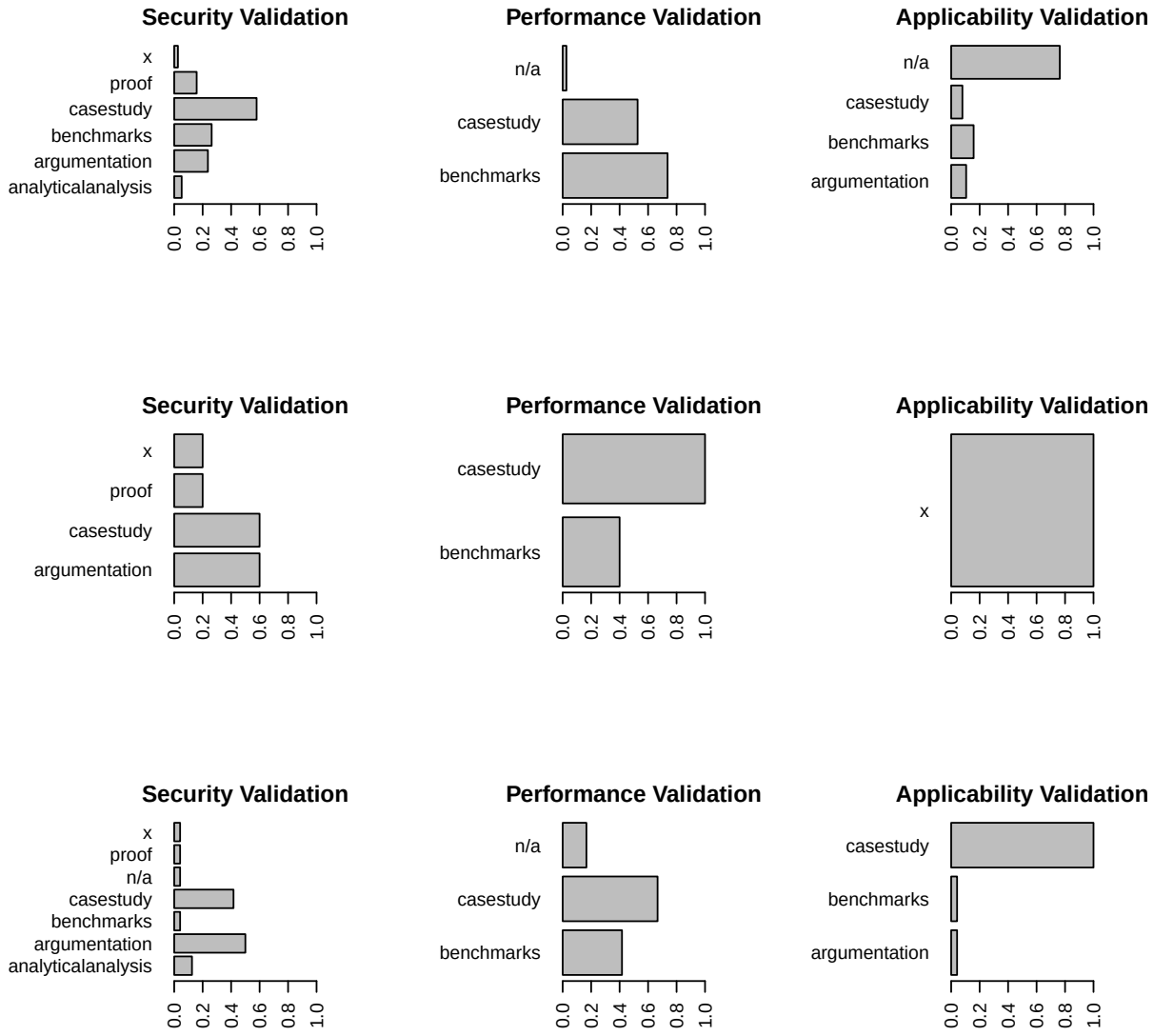


Figure 2.7: Breakdown of the representation of validation codes per claim type for the three validation clusters found in our dataset. Each row contains the data for one cluster. The bottom two clusters include papers that do not emphasize multi-faceted security validation strategies, instead relying on case studies and arguments that security claims are true. The characteristics of the two clusters are similar overall. However, while the papers in both clusters make applicability claims, the papers in the middle cluster, unlike the bottom cluster, did not validate those claims. Cases where a claim was made but not validated are labeled with an "x".

Table 2.5: The Spearman correlations and their statistical significances per validation category. Data with correlation coefficients closer to 1 have stronger correlations.

	Correlation (ρ)	**p-value**
Security Validation	-0.02	0.894
Performance Validation	0.30	0.014
Applicability Validation	0.20	0.105

monotonic functions (i.e. when a positive and perfect monotonic relationship exists, the Spearman correlation is 1).

Performance validation, when quantified as described in Section 2.2.3, is positively and statistically significantly correlated with the passage of time. We observe that performance validation has advanced from a heavy reliance on benchmark suites to the use multi-faceted strategies that include the use of benchmark suites and case studies (typically to perform micro-benchmarks) that make use of public data—which ensures the results are comparable with future sandboxes. While the applicability validation correlation is not statistically significant, we observe that argumentation was abandoned early on in favor of case studies, with some emphasis on including benchmark suites in later years. There is no apparent change in security validation over time.

We fit linear models to each validation category separately and together relative to ranked citation counts to see if validation practices are predictive of future citations. All of the models achieved an R-squared value of 0.54 which suggests that passage of time and validation practices jointly explain about half of the variance in citation count ranks. Validation practices on their own are not predictive of how highly cited a paper will become. Table 2.6 summarizes the types of claims and the validation strategies employed per type for each paper in our set.

While time plays a factor in how often a paper is cited, it is also not sufficient to predict citation counts. Consider Efstathopoulos et al. (2005) and Ringenburg and Grossman (2005), which are published in the same year and towards the beginning of the period under study. Using our criteria the Ringenburg paper is more strongly validated than the Efstathopoulos paper, yet they received 41 and 261 citations respectively at the time of this work. Zeldovich et al. (2006) is a year younger than both and was cited 418 times. This lack of a clear trend between citations and time, aside from the general trend that older papers tend to be cited more often, is repeated throughout the dataset.

2.4 Strengthening Sandboxing Results

The existing body of knowledge within the sandboxing community provides a strong basis for securing current and future software systems. However, the results in Section 2.3 highlight several gaps. In this section we discuss how structured arguments can solve the problems presented by incomparable and incomplete *ad hoc* arguments (Section 2.4.1) and possible ways to enhance sandbox and policy usability (Section 2.4.2).

Table 2.6: Claims made about sandboxes (🛡: Security, 🏎: Performance, and 🔗: Applicability) and their validation strategies (🎓: Proof, 📐: Analytical Analysis, ⏲: Benchmarks, 🏠: Case Studies, and 🚻: Argumentation). Grayed out icons mean a claim was not made or a strategy was not used. Icons made by Freepik from `www.flaticon.com`.

Category	Citation	Conference	Claims	🛡 Val.	🏎 Val.	🔗 Val.
Other (Syscall)	Provos (2003)	Usenix	🛡 🏎	🚻	⏲ 🏠	
Virtualization	Garfinkel et al. (2003a)	SOSP	🛡 🔗	🏠 🚻		⏲ 🏠 🚻
Diversity	Bhatkar et al. (2005)	Usenix	🛡 🏎 🔗	📐 🏠 🚻		⏲ 🏠 🚻
Other (Syscall)	Linn et al. (2005)	Usenix	🛡 🏎	🏠 🚻	⏲	
CFI	Abadi et al. (2005)	CCS	🛡 🏎 🔗	🎓 ⏲ 🏠 🚻	⏲	⏲ 🚻
Other (Memory)	Ringenburg and Grossman (2005)	CCS	🛡 🏎	🏠 🚻	⏲ 🏠	
MAC	Efstathopoulos et al. (2005)	SOSP	🛡 🏎	🏠 🚻	⏲ 🏠	
Web	Cox et al. (2006)	Oakland	🛡 🏎	🏠 🚻	⏲ 🏠	
SFI	McCamant and Morrisett (2006)	Usenix	🛡 🏎	🎓 🏠 🚻	⏲	
CFI, SFI	Erlingsson et al. (2006)	OSDI	🛡 🏎 🔗	🚻	⏲ 🏠	⏲ 🏠 🚻
Other (DFI)	Castro et al. (2006)	OSDI	🛡 🏎 🔗	🏠 🚻	⏲ 🏠	⏲ 🏠 🚻
Web	Reis et al. (2006)	OSDI	🛡 🏎	🏠 🚻	⏲ 🏠	
Other (InfoFlow)	Zeldovich et al. (2006)	OSDI	🛡 🏎	🏠 🚻	⏲ 🏠	
MI/AC	Li et al. (2007)	Oakland	🛡 🏎	🏠 🚻	⏲	
Web	Bandhakavi et al. (2007)	CCS	🛡 🏎 🔗	🏠 🚻	⏲	⏲ 🚻
Web	Chen et al. (2007)	CCS	🛡 🏎 🔗	🚻	⏲ 🏠	⏲ 🏠 🚻
Virtualization	Petroni and Hicks (2007)	CCS	🛡 🏎	🏠 🚻	⏲	
Virtualization	Seshadri et al. (2007)	SOSP	🛡 🔗	🚻		⏲ 🏠 🚻
Virtualization	Criswell et al. (2007)	SOSP	🛡 🏎	🎓 🏠 🚻		
Web	Wang et al. (2007)	SOSP	🛡 🏎 🔗	🏠 🚻	⏲	⏲ 🏠 🚻
Other (InfoFlow)	Krohn et al. (2007)	SOSP	🛡 🔗	🏠 🚻		⏲ 🏠 🚻
CFI	Akritidis et al. (2008)	Oakland	🛡 🏎	🏠 🚻	⏲	
Virtualization	Payne et al. (2008)	Oakland	🛡 🏎	🏠 🚻	⏲ 🏠	
MI/AC	Sun et al. (2008)	Oakland	🛡 🏎 🔗	🏠 🚻	⏲ 🏠	⏲ 🏠 🚻
Other (TaintTrack)	Chang et al. (2008)	CCS	🛡 🏎 🔗	🎓 🏠 🚻	⏲ 🏠	⏲ 🚻
Web	Oda et al. (2008)	CCS	🛡 🏎 🔗	🏠 🚻	⏲	⏲ 🚻
Other (OS)	Williams et al. (2008)	OSDI	🛡 🏎	🏠 🚻	⏲ 🏠	
SFI	Yee et al. (2009)	Oakland	🛡 🏎	🎓 🚻	⏲	
Web	Louw and Venkatakrishnan (2009)	Oakland	🛡 🏎 🔗	🚻	⏲	⏲ 🏠 🚻
Web	Parno et al. (2009)	Oakland	🛡 🏎 🔗	🏠 🚻	⏲ 🏠	⏲ 🏠 🚻
Other (Memory)	Akritidis et al. (2009)	Usenix	🛡 🏎 🔗	🚻	⏲	⏲ 🏠 🚻
Virtualization	Wang et al. (2009)	CCS	🛡 🏎	🏠 🚻	⏲ 🏠	
SFI	Castro et al. (2009)	SOSP	🛡 🏎 🔗	🏠 🚻	⏲ 🏠	⏲ 🏠 🚻
Virtualization	McCune et al. (2010a)	Oakland	🛡 🏎	🚻	⏲ 🏠	
Web	Meyerovich and Livshits (2010)	Oakland	🛡 🏎	🚻	🏠	
Other (Memory)	Akritidis (2010)	Usenix	🛡 🏎 🔗	🚻	⏲ 🏠	⏲ 🚻
SFI	Sehr et al. (2010)	Usenix	🛡 🏎	🏠 🚻	⏲	
Web	Louw et al. (2010)	Usenix	🛡 🏎 🔗	🏠 🚻	⏲	⏲ 🏠 🚻
Other (OS)	Wurster and van Oorschot (2010)	CCS	🛡 🏎 🔗	🏠 🚻	🏠	⏲ 🚻
SFI, Other (UserPolicy)	Siefers et al. (2010a)	CCS	🛡 🏎 🔗	🚻	🏠 🚻	⏲ 🚻
Web	Feldman et al. (2010)	OSDI	🛡 🏎 🔗	🚻	🏠	⏲ 🏠 🚻
MI/AC	Owen et al. (2011)	Oakland	🛡 🔗	📐 🚻		⏲ 🏠 🚻
Other (Transactions)	Jana et al. (2011)	Oakland	🛡 🏎 🔗	🏠 🚻	⏲ 🏠	⏲ 🏠 🚻
CFI	Zeng et al. (2011)	CCS	🛡 🏎	🚻	⏲	
Web	Saxena et al. (2011)	CCS	🛡 🏎	🏠 🚻	⏲ 🏠	
Web	Chen et al. (2011)	CCS	🛡 🏎	🎓 🚻	⏲ 🏠	
Virtualization	Zhang et al. (2011)	SOSP	🛡 🏎	🚻	⏲ 🏠	
SFI	Mao et al. (2011)	SOSP	🛡 🏎	🏠 🚻	⏲ 🏠	
Virtualization	Andrus et al. (2011)	SOSP	🛡 🏎 🔗	🚻	⏲ 🏠	⏲ 🏠 🚻
Diversity	Pappas et al. (2012)	Oakland	🛡 🏎	📐 🏠 🚻	⏲ 🏠	
Diversity	Hiser et al. (2012)	Oakland	🛡 🏎	🏠 🚻	⏲	
SFI	Payer et al. (2012)	Oakland	🛡 🏎	🚻	⏲ 🏠	
CFI	Kemerlis et al. (2012)	Usenix	🛡 🏎	🏠 🚻	⏲ 🏠	
Diversity	Giuffrida et al. (2012)	Usenix	🛡 🏎 🔗	📐 🚻	⏲ 🏠	⏲ 🏠 🚻
MI/AC	Xu et al. (2012)	Usenix	🛡 🏎 🔗	🏠 🚻	⏲	⏲ 🚻
Diversity	Wartell et al. (2012)	CCS	🛡 🏎 🔗	📐 ⏲ 🚻	⏲ 🏠	⏲ 🏠 🚻
Web, Other (InfoFlow)	De Groef et al. (2012)	CCS	🛡 🏎 🔗	🚻	⏲ 🏠	⏲ 🏠 🚻
Virtualization	Dunn et al. (2012)	OSDI	🛡 🏎 🔗	🏠 🚻	⏲ 🏠	⏲ 🏠 🚻
Web (MI/AC)	Giffin et al. (2012)	OSDI	🛡 🏎 🔗	🎓 🚻	⏲	⏲ 🏠 🚻
CFI	Zhang et al. (2013)	Oakland	🛡 🏎	🏠 🚻	⏲	
CFI	Zhang and Sekar (2013)	Usenix	🛡 🏎 🔗	🚻	⏲	⏲ 🚻
CFI, SFI	Niu and Tan (2013)	CCS	🛡 🏎 🔗	⏲ 🏠 🚻	⏲	⏲ 🏠 🚻
Diversity	Homescu et al. (2013)	CCS	🛡 🏎 🔗	🚻	⏲	⏲ 🏠 🚻
Other (OS)	Moshchuk et al. (2013)	CCS	🛡 🏎 🔗	🏠 🚻	🏠	⏲ 🏠 🚻
Virtualization	Nikolaev and Back (2013)	SOSP	🛡 🏎 🔗	🚻	⏲ 🏠	⏲ 🏠 🚻
CFI	Criswell et al. (2014)	Oakland	🛡 🏎	🎓 ⏲ 🚻	⏲ 🏠	
Web	Mickens (2014)	Oakland	🛡 🏎 🔗	🚻	🏠	⏲ 🏠 🚻

2.4.1 Structured Arguments

Sandboxes are often evaluated against coarse criteria such as the ability to stop exploits against certain classes of vulnerabilities, to encapsulate certain categories of operations, or to function in new environments. However, these coarse criteria typically require the sandbox to address a number of sub-criteria. For example, Zhang and Sekar (2013) provide CFI without requiring compiler support or *a priori* metadata, unlike earlier implementations. To ensure the technique is secure, they must be sure that independently transformed program modules maintain CFI when composed. Details that clarify how an individual criterion is fulfilled can easily be lost when *ad hoc* arguments are used in an effort to persuade readers that the criterion has been met; particularly in sandboxes with non-trivial design and implementation details. This can leave the reader unable to compare similar sandboxes or confused about whether or not contributions were validated.

Since many of the security criteria are repeated across most papers, the cost of developing substructure can be amortized across lots of communal use. There are many possible ways to structure arguments in support to security claims:

- Assurance cases (Kelly, 1999; Weinstock et al., 2007) provide graphical structures that explicitly tie claims together in trees that show how claims are narrowed. Knight (2015) provides a concise introduction to the topic. These structures also explicitly link leaf claims to the evidence that supports the claim. Assurance cases were created in response to several fatal accidents resulting from failures to systematically and thoroughly understand safety concerns in physical systems. Their use has spread to security and safety critical systems of nearly every variety in recent decades with case studies from aerospace (Graydon et al., 2007) and a sandbox called S^3 (Rodes et al., 2015) that was not analyzed as part of this study (Nguyen-Tuong et al., 2014). Sandboxing papers can use assurance cases to decompose claims to their most simple components, then link those components to relevant evidence in the paper (e.g. a summary of specific results, a specific section reference, etc.).
- Maass et al. (2014) use a qualitative framework to compare sandboxes based on what happens when a sandbox fails, is bypassed, or holds. Authors could structure their arguments by using the framework to describe their specific sandbox without performing explicit comparisons.
- Structured abstracts (Hartley, 2004; R. Brian Haynes et al., 1990) are used in many medical journals to summarize key results and how those results were produced. These abstracts have the benefit of being quick to read while increasing the retention of information, largely thanks to the use of structure to guide authors in precisely summarizing their work.
- Papers could provide a table summarizing their contributions and the important design or implementation details that reflect the contribution.

All of these approaches provide the reader with data missing in *ad hoc* arguments: A specific map from the claims made about a sandbox to evidence that justifies the claim has been met. They are also necessarily qualitative, but as we saw earlier, arguments are often used where more rigorous approaches are currently intractable. We believe that

adding structure to these arguments is a reasonable advancement of the state of practice in sandbox validation.

2.4.2 Sandbox and Policy Usability

Sandbox and policy usability are concerns of interest to the following stakeholders: *Practitioners* that must correctly use sandboxes to improve the security postures of their systems and *users* that must work with sandboxed applications. Some security researchers do attempt to make their sandboxes more usable by providing policy management or reducing requirements on the user, but usability is not a focus of any of the papers in our sample.

Our data shows that, with very few exceptions, sandbox researchers thoroughly evaluate the performance of their sandboxes. Why is there focus on this practical concern but not on usability? We observe that a focus on performance evaluation is partially motivated by the fact that overhead is relatively easy to quantify, but we also saw many cases where researchers were explicitly concerned with whether or not a sandbox was too resource intensive for adoption. The latter is a reasonable concern; Szekeres et al. (2013) pointed out that many mitigations for memory corruption vulnerabilities are not adopted because performance concerns outweigh protection merits.

While the idea that performance is an important adoption concern is compelling and likely reflects reality, we cannot correlate performance with the adoption of the sandboxes in our set. We cannot find a correlation because the sandboxes and their techniques in our set remain almost entirely unadopted. We only found four cases where sandboxes in our set were either directly adopted or where the techniques they evaluate are clearly implemented in a different but adopted sandbox. A lack of adoption is present even for techniques where performance and applicability have been improved over multiple decades (e.g. SFI). Three of the adopted sandboxes were created by the industry itself or by entities very closely tied to it: Google NaCl was designed with the intention of adopting it in Google Chrome in the short term (Sehr et al., 2010; Yee et al., 2009) and the paper on `systrace` was published with functioning open source implementations for most Unix-like operating systems (Provos, 2003). While the case for adoption is weaker, Cells (Andrus et al., 2011) is a more advanced design than one VMware developed in parallel (Berlind, 2012), although the sandboxes both aim to partition phones into isolated compartments using virtualization (e.g. one for work and one for personal use). More recently, Microsoft has stated that Visual Studio 2015 will ship with an exploit mitigation that we believe is equivalent to what the research community calls CFI (Hogg, 2015). A third party analysis supports this belief, however the uncovered implementation details differ from the techniques implemented in published research (Tang, 2015).

We argue that the need to evaluate the usability of our sandboxes is evidenced by the observation that performance and security evaluation are not sufficient to drive adoption. Usability is of particular concern in cases where the sandbox requires developers without security expertise (1) to re-architect applications to apply the sandbox and/or (2) to develop a security policy. In practice, it is quite common for developers without a security focus to apply sandboxes, particularly Java's. In fact, usability issues have factored into widely publicized vulnerabilities in how sandboxes were applied to Google Chrome and

Adobe Reader as well as the many vulnerable applications of the Java sandbox (Coker et al., 2015). In all of these cases applying the sandbox is a relatively manual process where it is difficult for the applier to be sure he is fully imposing the desired policy and without missing relevant attack surfaces. These usability issues have caused vulnerabilities that have been widely exploited to bypass the sandboxes. Given common patterns in usability deficiencies observed in practice, we suggest authors evaluate the following usability aspects of their sandboxes where appropriate:

- The intended users are capable of writing policies for the component(s) to be sandboxed that are neither over- or under-privileged.
- Policy enforcement mechanisms can be applied without missing attack surfaces that compromise the sandbox in the targeted component(s).
- Source code transformations (e.g. code re-writing or annotations) do not substantially burden future development or maintenance.
- The sandbox, when applied to a component, does not substantially alter a typical user's interactions with the sandboxed component.

Ideally in the research community many of these points would be evaluated during user studies with actual stakeholders. However, we believe that we can make progress on all of these points without the overhead of a full user study, particularly because we are starting from a state where no usability evaluations are performed. For example, authors can describe correct ways to determine what privileges in their policy language a component needs or even provide tools to generate policies to mitigate the risks presented by under- and over-privileged policies. Similarly, tooling can be provided to help users install policy enforcement mechanisms or check that manual applications of a mechanism are correct. Sandbox developers can transform or annotate representative open source applications and use repository mining[14] to determine how sandbox alternations are affected by code evolution present in the repository (Kagdi et al., 2007; Mauczka et al., 2010; Stuckman and Purtilo, 2014; Yan et al., 2014). Finally, a summary of how the sandbox qualitatively changes a user's experience with a sandboxed component would provide a gauge for how much the sandbox burdens end-users.

2.5 Enabling Meta-Analysis

We believe a key contribution of this work is the use of multi-disciplinary and systematic methodologies for drawing conclusions about a large body of security techniques. In this section, we discuss the generalizability of our methodology and suggest other areas to which it can be applied. Then, we discuss some challenges that we faced when doing this research and suggest changes that would address these challenges.

[14] `http://msrconf.org`

2.5.1 Generalizability of Methodology

The methodology employed in this chapter is based on two research approaches: Qualitative Content Analysis and Systematic Literature Reviews. Qualitative Content Analysis is primarily used in the humanities and social sciences. Systematic Literature Reviews were first applied to medical studies and are used primarily in empirical fields. The differences between sandboxing papers are bigger than the differences between studies of a particular cancer treatment. In addition, sandboxing papers do not fit into the "native" domains of either approach—their primary contributions are designs, techniques, and implementations.

The result of these differences is that most literature reviews and systemizations in computing are done in an ad hoc manner. Our computing research is worthy of a more rigorous approach and we think the methodology applied in this chapter can and should be applied to other topics. In fact, any topic of active research where the primary contributions is an engineered artifact, but without a clear and precise definition, would be amenable to our approach. These topics span computing research from software engineering (e.g. service oriented architecture, concurrent computation models) to systems (e.g. green computing, no instruction set computing) to human-computer interaction (e.g. GUI toolkits, warning science).

2.5.2 Meta-analysis Challenges and Suggested Solutions

In our experience, the biggest roadblock standing in the way of applying the same techniques to other segments of the research community lies in the difficulty involved in collecting analyzable metadata about papers. We experienced several fixable issues:

- The major publishers in computer science—IEEE, ACM, and Usenix—do not provide publicly available mechanisms to collect metadata and either rate limit or outright ban scraping.[15] In our case, the painstaking process of collecting and curating analyzable metadata across several sources limited our ability to explore hypotheses about our dataset's papers and their relationships to publications not in the set.
- The metadata is limited and contains little semantic content—typically the metadata includes the authors, title, data, and DOI, but little else. If abstracts and keywords were easier to harvest we could have more systematically derived topics of interest within the sandboxing community.
- Links to papers on publisher websites use internal identifiers (e.g. `http://dl.acm.org/citation.cfm?id=2498101`) instead of DOI. This makes it difficult to reference papers across publisher repositories.
- Conference websites have inconsistent layouts, which increases the difficulty of data collection.

We believe easier access to this data would have allowed us to draw more conclusions about how sandboxing papers are related and how the sandboxing landscape has evolved

[15] In at least one case ACM provided a copy of their digital library for scraping (Bergmark et al., 2001)

over time. For example, we explored the idea of using a more developed citation graph than Figure 2.2 to trace the lineage of sandboxing techniques, but found the required resource expenditures were outside of our means. This data may provide support for explanations regarding the lack of advancement in security validation practices (e.g. by showing an emphasis on a different but important dimension of advancement). These points are important to understand how we got to the current state of practice, thus improving our ability to recognize and advance means for enhancing our results.

On another data collection point, we averaged about 45 minutes per paper to code the data necessary to answer our research questions. While we do not claim that our research questions are of universal interest to the sandboxing community, we did observe that papers that answer all or most of the questions in the abstract are often clearly written throughout and easy to interpret. A small minority of sandboxing papers have far less specific abstracts. In these cases, the papers often took double the average time to comprehend and interpret. It may be useful to strive to clearly answer questions like ours in future papers to show practitioners the value sandbox researchers bring to the table. Structured abstracts, as discussed in Section 2.4.1, may be a good way to ensure these types of questions are succinctly answered.

2.6 Threats to Validity

Due to the complexity of the text and concepts we are interpreting, there is some risk that other coders would assign quotes to different codes. Different codes will change the results, but we believe this risk is mitigated through our tests of the coding frame and by our efforts to select clear quotes. Furthermore, the correlative nature of our results ensures that a few code divergences will not dramatically change the analysis's outcomes.

The primary risk is that we are missing relevant quotes that add codes to our dataset. This is typically mitigated in QCA by fully segmenting the text, but we decided against that strategy because of the very large data set we studied and irrelevance of most of the text to our goals. We did search PDFs for relevant keywords we observed were commonly linked to specific codes throughout the process (e.g. "proof", "available" to find the availability of sandbox artifacts for evaluation, "experiment" to signal a case study or benchmark, etc.) to decrease the odds of missing a code. While this does mitigate the risk, it is still likely that our results under-approximate the state of the sandboxing landscape.

2.7 Conclusion

We systematically analyzed the sandboxing landscape as it is represented by five top-tier security and systems conferences. Our analysis followed a multidisciplinary strategy that allowed us to draw conclusions backed by rigorous interpretations of qualitative data, statistics, and graph analysis. We drew several conclusions that would not have been possible to draw with a reasonable level of confidence if it were not for the use of these explicit methods:

- Sandbox security claims are often evaluated using more subjective validation strategies.
- The selected validation strategies have not grown less subjective over the period under study.
- The studied subset of the sandboxing community does not consider the usability of their mechanisms from any standpoint.
- There is an important usability trade-off presented by the flexibility of the policy or policies imposed by a sandbox.

Based on our results, we conclude that the sandbox research community will benefit from the use of structured arguments in support of security claims and the validation of sandbox and policy usability. We suggested lightweight ways to move forward in achieving these goals. Our data also shows that there is a dearth of science regarding the management of security policies for sandboxes, although we did not discuss this gap in depth.

Having drawn broad conclusions about the sandboxing landscape, we now narrow our focus to the Java sandbox. This narrowing allows us to study how issues we observed in this chapter manifest in a practical sandbox that has existed and been used for multiple decades. Additionally, Java's sandbox is known to have deployment issues as evidenced by a number of successful attacks that break out of the sandbox.

Chapter 3

Evaluating the Flexibility of the Java Sandbox[1]

It has become common for people to assume that Java is insecure after dozens of successful exploit campaigns targeted Java users in recent years, peaking in 2013. Since 2013, there has been one serious exploit campaign targeting Java, which took place in the summer of 2015. These exploit campaigns targeted vulnerabilities that exist within Java code, particularly code found in the Java Class Library. There are three broad reasons why Java has been such a popular target for attackers. First, the Java Runtime Environment (JRE) is widely installed on user endpoints. Second, the JRE can and often does execute external code, in the form of applets and Java Web Start (JWS) applications (Gong and Ellison, 2003; Gong et al., 1997). Finally, there are hundreds of known and zero-day vulnerabilities (IBM Security Systems, 2014) in Java. In the common scenario, often referred to as a "drive-by download," attackers lure users to a website that contains a hidden applet to exploit JRE vulnerabilities.

In theory, such attacks should not be so common: Java provides a sandbox to enable the safe execution of untrusted code and to isolate components from one another. In particular, the sandbox was constructed to enable the use of code from remote origins that may be malicious. The sandbox regulates the behaviors encapsulated code can exhibit by ensuring that operations that can effect the external environment (e.g. writing to a file, opening a network connection, etc.) are only executed if the imposed security policy grants the necessary permission(s). This should protect both the host application and machine from malicious behavior. In practice, these security mechanisms are buggy and leave room for Java malware to alter the sandbox's settings (Garber, 2012) to override security mechanisms. Such exploits take advantage of defects in either the JRE itself or the application's sandbox configuration to disable the security manager, the component of the sandbox responsible for enforcing the security policy (Gowdiak, 2012; Oh, 2012; Singh and Kapoor, 2013; Svoboda, 2013).

This chapter investigates this disconnect between theory and practice. We hypothesize

[1]This chapter was adapted from a paper written with help from Zach Coker, Tianyuan Ding, Claire Le Goues, and Joshua Sunshine. It is published in "Annual Computer Security Applications Conference" 2015

that the disconnect results primarily from unnecessary complexity and flexibility in the design and engineering of Java's security mechanisms. For example, applications are allowed to change the security manager at runtime, whereas static-only configuration of the manager would be more secure. The JRE also provides a number of security permissions that are so powerful that a sandbox that enforces any of them cannot be secure. We have not been able to determine why this complexity and flexibility exists but hypothesize that benign applications do not need all of this power, and that they interact with the security manager in ways that are measurably different from exploits. If true, these differences can be leveraged to improve the overall security of Java applications, and prevent future attacks.

To test these hypotheses, we conducted an empirical study to answer the question: How do benign applications interact with the Java security manager? We studied and characterized those interactions in 36 open-source Java projects that use the security manager, taken from the Qualitas Corpus (Tempero et al., 2010) and GitHub.

We discovered two types of security managers in practice. *Defenseless* managers enforce a policy that allows sandboxed code to modify sandbox settings. Such applications are inherently insecure, because externally-loaded malicious code can modify or disable the security manager. We found defenseless managers in use by applications that modified sandbox settings at runtime, typically as workarounds to using more complicated (but more correct) security mechanisms or to enforce policies or implement functionality unrelated to security. We believe that such applications use the sandbox to implement certain non-security requirements because Java does not provide better mechanisms for doing so. The sandbox is not intended to be used this way, and these use cases both reduce security in practice and limit the potential exploit mitigations that are backwards compatible with benign applications. On the other hand, applications with *self-protecting* managers do not allow sandboxed code to modify security settings. It might still be possible to exploit such applications due to defects in the JRE code that enforces security policies, but not due to poorly-deployed local security settings.

We found that it is rare for software to use the sandbox as intended—for protection from malicious external code—and cases where it is used do not use its vast flexibility and often contain configuration mistakes. For these applications, the sandbox's flexibility decreases their security without obviously benefiting the developers or application functionality. In fact, we found and reported a security vulnerability related to the sandbox in one of the applications under study. Our findings that the sandbox is rarely used and used in ways that are vulnerable suggest that the sandbox contains complexity that should either be simplified away or overcome through the use of tooling. However, there is little use in applying the sandbox further until known means of exploiting it are solved. This chapter provides a solution to stop these exploits, thus staging the groundwork for overcoming remaining complexity hampering legitimate uses of the sandbox.

We propose two runtime rules that restrict the flexibility of the sandbox and fortify Java against the two most common modern attack types without breaking backwards compatibility in practice. We evaluate our rules with respect to their ability to guard against ten applets in a popular exploit development and delivery framework, Metasploit

4.10.0[2], that successfully attack unpatched versions of Java 7. Taken together, the rules stopped all ten exploits and did not break backwards-compatibility when tested against a corpus of benign applications.

This chapter contributes the following:

1. A study of open-source applications' interactions with the security manager (Section 3.3). We identify open-source applications that enforce constraints on subcomponents via the Java sandbox, as well as unconventional behaviors that indicate usability and security problems that the Java security model can be improved to mitigate.
2. An enumeration of Java permissions that make security policies difficult to enforce (Section 3.1.2), a discussion of real-world cases where these permissions are used (Sections 3.3.3 and 3.3.4), and a sandbox-bypassing exploit for a popular open-source application made vulnerable due to their use (Section 3.3.4).
3. Two novel rules for distinguishing between benign and malicious Java programs, validated empirically (Section 3.4). These rules define explicit differences between benign and malicious programs, they are not heuristics.
4. A discussion of tactics for practically implementing the rules, with a case for direct JVM adoption (Section 3.4.1).

We begin by discussing necessary background on the Java sandbox and exploits (Section 3.1). We present the methodology and dataset for our empirical study in Section 3.2. The results of the study are discussed in Section 3.3, leading to our rules which are defined and evaluated in Section 3.4. Finally, we discuss limitations, cover related work, and conclude in Sections 3.5, 3.6, and 3.7 respectively.

3.1 Background

In this section, we describe the Java sandbox (Section 3.1.1), distinguish between defenseless and self-protecting security managers (Section 3.1.2) and provide a high-level description of how Java exploits commonly work (Section 3.1.3).

3.1.1 The Java sandbox

The Java sandbox is designed to safely execute code from untrusted sources using components summarized in Figure 3.1. When a *class loader* loads a class (e.g., from the network, filesystem, etc.), it assigns the class a *code source* that indicates the code origin, and associates it with a *protection domain*. Protection domains segment the classes into groups by *permission set*. These sets contain permissions that explicitly allow actions with security implications, such as writing to the filesystem, accessing the network, etc (Oracle, 2014c). Unlisted actions are disallowed. *Policies* written in the Java policy language (Oracle, 2014a) define permission sets and their associated code sources. By default, all classes *not*

[2]http://www.metasploit.com/

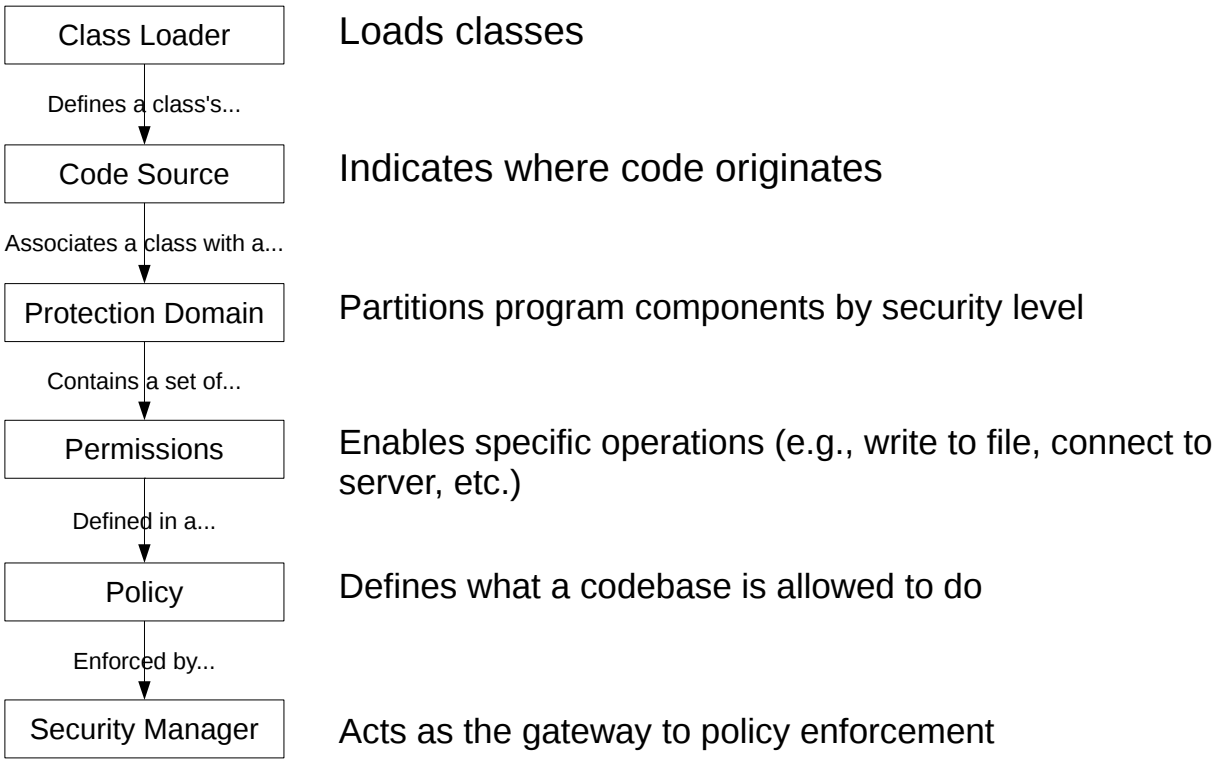


Figure 3.1: High-level summary of the Java sandbox.

loaded from the local file system are run within a restrictive sandbox that restricts their ability to interact with the host application or machine.

The sandbox is activated by setting a *security manager*, which acts as the gateway between the sandbox and the rest of the application. Whenever a sandboxed class attempts to execute a method with security implications, that method queries the security manager to determine if the operation should be permitted. To perform a permission check, the security manager walks the call stack to ensure each class in the current stack frame has the specific permission needed to perform the action.[3]

Missing and poorly scoped checks in code that *should* be protected are a common source of Java vulnerabilities, because the security-critical code must initiate the check (Cifuentes et al., 2015). Note that such vulnerabilities lie in the JRE itself (i.e., the code written by the Java developers), not in code using the sandbox to execute untrusted code.

3.1.2 Defenseless vs. self-protecting managers

Java is flexible about when the sandbox is enabled, configured, and reconfigured. The default case for web applets and applications that use Java Web Start is to set what we call a *self-protecting* security manager before loading the network application. The

[3]Stack-based access control is discussed in more detail in (Banerjee and Naumann, 2005; Besson et al., 2004; Fournet and Gordon, 2002; Wallach and Felten, 1998a)

Table 3.1: List of sandbox-defeating permissions. A security manager that enforces a policy containing any of these permission results in a defenseless sandbox. A subset of these permissions were first identified in (Gowdiak, 2012).

Permission	**Risk**
RuntimePermission("createClassLoader")	Load classes into any protection domain
RuntimePermission("accessClassInPackage.sun")	Access powerful restricted-access internal classes
RuntimePermission("setSecurityManager")	Change the application's current security manager
ReflectPermission("suppressAccessChecks")	Allow access to all class fields and methods as if they are public
FilePermission("<<ALL FILES>>", "write, execute")	Write to or execute any file
SecurityPermission("setPolicy")	Modify the application's permissions at will
SecurityPermission("setProperty.package.access")	Make privileged internal classes accessible

security manager, and thus the sandbox, is self-protecting in the sense that it does not allow the application to change sandbox settings. Self-protecting managers might still be exploited due to defects in the JRE code that enforces security policies, but not due to poorly-deployed local security settings. We contrast self-protecting managers with those we call *defenseless*, meaning that sandboxed applications are permitted to modify or disable the security manager. A defenseless manager is virtually useless in terms of improving the security of either a constrained application or its host. However, we find in Section 3.3 that developers have found interesting non-security uses for defenseless managers in otherwise benign software.

We evaluated whether each of the available Java permissions can lead to sandbox bypasses. Table 3.1 summarizes the set of permissions that distinguish between self-protecting and defenseless security managers.

3.1.3 Exploiting Java code

While the Java sandbox *should* prevent malicious applets from executing their payloads, certain defects in the JRE implementation of these security mechanisms can permit malicious code to set a security manager to `null`.[4] This disables the sandbox and enables all operations. This approach was common in drive-by downloads between 2011 and 2013 (Singh and Kapoor, 2013). Figure 3.2 shows a typical payload class whose privileges have been

[4]Many of the recent vulnerabilities would not have been introduced if the JRE were developed strictly following "The CERT Oracle Secure Coding Standard for Java." (Long et al., 2011; Svoboda, 2013; Svoboda and Toda, 2014)

```
import java.lang.reflect.Method;
import java.security.AccessController;
import java.security.PrivilegedExceptionAction;

public class Payload
        implements PrivilegedExceptionAction {
    public Payload() {
        try {
            AccessController.doPrivileged(this);
        } catch(Exception exception) { }
    }

    public void run() {
        // Disable sandbox
        System.setSecurityManager(null);
    }

    public static void outSandbox() {
        // Do malicious operations
    }
}
```

Figure 3.2: A typical sandbox-disabling Java exploit payload from `http://pastebin.com/QWU1rqjf`.

elevated by an exploit to allow it to disable the sandbox. This example payload uses `doPrivileged` to allow the unprivileged exploit class to execute the operations in the payload without causing a `SecurityException`.

There are a couple of ways to maliciously disable a security manager. **Type confusion** attacks break type safety to craft objects that can perform operations as if they have a different type. Commonly, attackers craft objects that (1) point to the `System` class to directly disable the sandbox or (2) act as if they had the same type as a privileged class loader to elevate a payload class's privileges (see CVE-2012-0507 (NIST, 2012)). In **confused deputy** attacks, exploitative code "convinces" another class to return a reference to a privileged class (Hardy, 1988) known to contain a vulnerability that can be attacked to disable the sandbox (see CVE-2012-4681 (NIST, 2013a)). The "convincing" is necessary because it is rare that a vulnerable privileged class is directly accessible to all Java applications; doing so violates the *access control* principle that is part of the Java development culture.[5]

Typically, there are less-privileged code paths to access operations that ultimately exist in privileged classes. The danger of vulnerabilities in privileged classes is therefore mitigated, because they cannot be directly exploited unless malicious code first modifies its own privileges. This redundancy is implicit in the Java security model. If any class could load more privileged classes and directly execute privileged operations, the sandbox in its current form would serve little purpose.

In practice, however, Java's security mechanisms as implemented in the JRE contain defects and vulnerabilities that reduce the benefits of this redundancy.

Modern exploits that manipulate the security manager simply disable it. This is possible largely because the Java security model grants enormous flexibility to set, weaken, strengthen, or otherwise change a security manager after its creation. Do applications need this power? Do they regularly take advantage of the ability to disable or weaken the sandbox? If not, we can stop exploits for even currently unknown vulnerabilities by eliminating these operations without breaking backwards-compatibility with benign applications. Our core thesis is that the overall security of Java applications could be improved by simplifying these security mechanisms, without loss to benign functionality.

3.2 Security manager study

In this section, we describe the dataset and methodology for our empirical study of the open-source Java application landscape. Our basic research question is: How do benign open-source Java applications interact with the security manager? The answer to this question informs which JVM-level modifications can be used to improve security while maintaining backwards compatibility. We consider only benign application code that is intended to run in production, not unit testing code, code that manages the build, etc. There are four possible answers to our question:

- *Benign applications never disable the security manager.* If true, only exploitative

[5] `https://blogs.oracle.com/jrose/entry/the_isthmus_in_the_vm`

code attempts to disable the security manager, and the ability to do so could be removed from the JVM. This would be easy to implement but would not guard against exploits that weaken the sandbox without disabling it.

- *Benign applications do not weaken a set security manager.* If true, the JVM could be modified to prevent any weakening or disabling of the sandbox. This is more powerful than simply removing the ability to disable the security manager but is significantly more difficult to implement. For example, if a permission to write to a file is replaced by a permission to write to a different file, is the sandbox weakened, strengthened, or equally secure?
- *Benign applications never modify the sandbox if a self-protecting security manager has been set.* If true, the JVM could disallow any change to a self-protecting security manager. A runtime monitor in the JVM can determine if a security manager is self-protecting (based on the permission set) when an application attempts to change the sandbox. This is much easier to implement soundly than the previously-described approach, and guards against the same number and types of exploits.
- *Benign applications do not change a set security manager.* If true, any attempted change to an already established security manager can be considered malicious. This would be the ideal result: restricting this operation is easy to implement in the JVM.

This section describes our study dataset (Section 3.2.1) and methodology (Section 3.2.2); we describe results in Section 3.3.

3.2.1 Dataset

In the absence of an existing corpus of benign applications that interact with the manager, we combined relevant subjects from the QualitasCorpus (QC) version 20130901 (Tempero et al., 2010), a collection of popular open source Java applications created for empirical studies, and GitHub. While QC contains 112 applications, we found only 24 applications interacted with the security manager. To increase the size and diversity of the dataset (beyond those that meet the QC inclusion requirements), we added 12 applications from Github that also interact with the security manager.[6] Table 3.2 lists the 36 applications that comprise the dataset. Version numbers and Git commit hashes are available in an online supplement. [7]

We identified relevant applications in the QC set by searching for the keyword `SecurityManager` in the applications' source code. We performed a similar process on the GitHub set, adding the keywords `System.setSecurityManager(` and `System.setSecurityManager(null)` to remove false positives and find applications that disable the manager, respectively. We picked the top 6 applications from the results for each keyword, removing manually-identified false positives and duplicates. We studied

[6]Applets, commonly run in a sandboxed environment, would be natural study subjects. However, we were unable to find any benign applets that interacted with the security manager, likely because of Java's strict restrictions on their behavior.

[7]`https://github.com/SecurityManagerCodeBase/ProjectsProfiles/blob/master/projectslist.xlsx`

each GitHub program at its most recent commit and each QC program at its most recent stable release as of June 2014.

3.2.2 Methodology

We performed a tool-supported manual inspection of the applications in our dataset to group them into qualitative, non-overlapping categories based on their interactions with the security manager. The first category includes applications that can or do *change a set security manager* at run time. If an application did not change the security manager, we looked to see if it set a single security manager during execution. If so, it was categorized as *sets an immutable manager*. Applications that interact with a security manager that they did not set do so to adjust to different security settings. We categorized any such application as *supports being sandboxed*. For the final category, if a program did not contain any interaction with the security manager in the main application, but did interact with it in test cases, we categorized the application as *only interacts in unit tests*; such applications use unit test interactions to test against multiple security settings.

We created static and dynamic analysis tools to assist in a manual inspection of each application's security manager interactions. We created a FindBugs (Hovemeyer and Pugh, 2004) plugin that uses a sound dataflow analysis to determine which manager definitions reach calls to `System.setSecurityManager()`. The dynamic analysis tool uses the Java Virtual Machine Tool Interface (JVMTI) (Oracle, 2014b) to set a modification watch on the `security` field of Java's `System` class, which stores the security manager object for the application.

We split the dataset between two investigators, who each analyzed applications using the following steps:

- Run `grep` on all Java source files in the application to find lines containing the keyword `SecurityManager`. Manually inspect the results in their original source code files to understand how the application interacts with the sandbox.
- Run the static analysis on retained applications. Manually inspect the returned code, focusing on initialization.
- Use the dynamic analysis, using parameters informed by the previous steps, to validate conclusions.
- Summarize operations performed on the security manager, categorize accordingly, and determine if the security manager is self-protecting or defenseless.

We undertook a pilot study where each investigator independently inspected the same six applications and compared their results. This ensured the two investigators understood the analysis steps and produced consistent results.

3.3 Study results

In this section, we describe the results of our empirical study of open-source Java programs and how they interact with the security manager.

Table 3.2: Security manager interactions dataset.

App name	Description	KLOC	Repo
Apache Ant	Java Project Builder	265	QC
Apache Batik	SVG Image Toolkit	366	QC
Apache Derby	Relational Database	1202	QC
Eclipse	IDE	7460	QC
FreeMind	Mind-Mapping Tool	86	QC
Galleon	Media Server	83	QC
Apache Hadoop	Distrib. Comp. Frwk.	1144	QC
Hibernate	Obj.-Rel. Mapper	376	QC
JBoss	App. Middleware	968	QC
JRuby	Ruby Interpreter	372	QC
Apache Lucene	Search Software	726	QC
Apache MyFaces	Server Software	328	QC
NekoHTML	HTML Parser	13	QC
Netbeans	IDE	8490	QC
OpenJMS	Messaging Service	112	QC
Quartz	Job Scheduler	66	QC
QuickServer	TCP Server Frwk.	64	QC
Spring	Web Dev. Library	828	QC
Apache Struts	Web Dev. Library	277	QC
Apache Tomcat	Web Server	493	QC
Vuze	File Sharing App.	895	QC
Weka	Machine Learning Algs.	531	QC
Apache Xalan	XML Trans. Library	204	QC
Apache Xerces	XML Parsing Library	238	QC
AspectJ	Java Extension	701	GH
driveddoc	Application Connector	7	GH
Gjman	Development Toolkit	¡1	GH
IntelliJ IDEA	IDE	4094	GH
oxygen-libcore	Android Dev. Lib.	1134	GH
refact4j	Meta-model Frwk.	21	GH
Security-Manager	Alt. Security Manager	4	GH
Spring-Modules	Spring Extension	212	GH
System Rules	JUnit Extension	2	GH
TimeLag	Sound Application	1	GH
TracEE	JavaEE Support Tool	18	GH
Visor	Closure Library	1	GH

3.3.1 Summary of benign behaviors

Recall that in Section 3.2, we refined the high-level research question—how do benign applications interact with the security manager?—into four possibilities, and that the possible mitigations required in each case varied. Revisiting those possibilities with respect to our dataset, as summarized in Table 3.3, we found:

- Benign applications *do* sometimes disable the security manager. We found that such applications typically use a defenseless sandbox for non-security purposes. We discuss some of these applications in more detail in Section 3.3.3.
- Several benign applications *do* provide methods for the user to dynamically change the security policy or the manager in ways that can reduce sandbox security.
- Benign applications do *not* change the security manager if a self-protecting security manager has been set.
- Benign applications do sometimes change a set security manager. We observed multiple applications that changed a set security manager.

In terms of the four possible mitigation strategies, only the third—a runtime monitor that blocks modifications to a self-protecting security manager—can improve security without breaking benign behavior. Fortunately, this technique does not require complex, context-sensitive information about whether a change to a policy weakens the sandbox or not.

3.3.2 Applications by category

Table 3.3 summarizes our dataset into the categories described in Section 3.2. The applications in categories 1, 3, and 4 are consistent with any of the potential JVM modifications because they do not interact with the security manager in complex ways (i.e., if all applications fell into these categories, the Java security model could be dramatically simplified by eliminating most of its flexibility without breaking existing applications). We will not discuss applications in categories 3 or 4 further, because they did not result in useful insights about common benign behaviors. Most applications in the *Sets an immutable manager* category use the sandbox correctly. We discuss a few particularly interesting examples below. There are eight applications in the *Changes set manager* category, which is the most interesting in terms of possible modifications to the Java security model. They make the most use of Java's flexible security mechanisms. We therefore focus on these applications in our discussion.

We discuss applications that use the sandbox for non-security purposes in Section 3.3.3 and applications that use the sandbox for its intended security purposes in Section 3.3.4.

3.3.3 Non-security uses of the sandbox

Most of the applications that interact with the sandbox in non-security ways did so to enforce architectural constraints when interacting with other applications; the rest forcibly disabled the sandbox to reduce development complexity. This misappropriation of Java's

Table 3.3: Classification of application interactions with the security manager.

Type of Interaction	QC	GitHub	Total
1. Sets an immutable manager	6	1	7
2. Changes set manager	5	3	8
3. Supports being sandboxed	10	3	13
4. Interacts only in unit tests	3	5	8

```
System.setSecurityManager(new AntSecurityManager(originalSM,
  Thread.currentThread()));
// ...execute Ant...
finally {
  // ...
  if (System.getSecurityManager() instanceof AntSecurityManager) {
      System.setSecurityManager(originalSM);
  }
```

Figure 3.3: Snippet of Eclipse code that uses a security manager to prevent Ant from terminating the JVM.

security features increases the difficulty of mitigating attacks against them by increasing the odds of backwards compatibility issues. These applications included applications in both categories 1 and 2, and all require defenseless security managers.

Enforcing architectural constraints

Java applications often call `System.exit()` when an unrecoverable error occurs. When such an application is used as a library, `System.exit()` closes the calling application as well, because both are running in the same JVM. To prevent this without modifying the library application, a calling application needs to enforce the architectural constraint that called library code cannot terminate the JVM.

We found three applications in our dataset that enforce this constraint by setting a security manager that prevents `System.exit()` calls: Eclipse, GJMan, and AspectJ.[8] For example, Eclipse uses Ant as a library, and Ant calls `System.exit()` to terminate a build script in the event of an unrecoverable error. However, when Eclipse uses Ant as a library, it reports an error to the user and continues to execute. Figure 3.3 shows how Eclipse uses a security manager to enforce this constraint; Eclipse restores the original manager after Ant closes.

This technique does enforce the desired constraint, and appears to be the best solution available in Java at the moment. However, it is problematic for applications using the sandbox for security purposes. The technique requires the application to dynamically

[8]GJMan contains a code comment referencing a blog post that we believe is the origin of this solution: `http://www.jroller.com/ethdsy/entry/disabling_system_exit`

change the security manager, which in turn requires a defenseless manager. As a result, the calling applications *themselves* cannot be effectively sandboxed, as might be desirable e.g., when run from Java Web Start. The host machine thus cannot be protected from the application itself, or the library code that the application calls.

Web applications outside the sandbox

We found web applications that insist on being run unsandboxed. By default, Java executes such applications inside a self-protecting sandbox with a restrictive policy that excludes operations like accessing local files, retrieving resources from third party servers, or changing the security manager.

Applications in our set that require these permissions opted to run outside of the sandbox. We found two applications that do this: Eclipse and Timelag. Both applications attempted to set the security manager to `null` at the beginning of execution. A restrictive sandbox catches this as a security violation and terminates the application; to run it, the user must ensure that such a sandbox is not set. The rationale for disabling the manager in Eclipse is explained in a code comment that reads, "The launcher to start eclipse using webstart. To use this launcher, the client must accept to give all security permissions." Timelag performs the same operation, but without associated explanatory comments that we could find, thus we can only infer the developer's motivation.

The developers of Eclipse and Timelag could have either: 1) painstakingly constructed versions of the application that run reasonably using only the permissions available within the sandbox (e.g. by detecting the sandbox and avoiding or disabling privileged operations) or 2) gotten the applications digitally signed by a recognized certificate authority and configured to run with all permissions. These developers likely found these alternatives overly burdensome. The examples from our study suggest that developers are sometimes willing to exchange security guarantees in the interest of avoiding such labor-intensive options.

3.3.4 Using the security manager for security

Other applications interact with the manager in security-oriented ways. Batik, Eclipse, and Spring-modules allow the user to set and change an existing manager; Ant, Freemind, and Netbeans explicitly set then change the manager.

Batik SVG Toolkit allows users to constrain applications by providing a method to turn the sandbox on or off. This trivially requires a defenseless sandbox. The Batik download page provides several examples of library use, one of which (the "rasterizer" demo) enables and disables the sandbox. However, there seems to be no reason to do so in this case other than to demonstrate the functionality; we were unable to discern the rationale from the examples or documentation.

Ant, Freemind, and Netbeans explicitly set then change the manager, requiring the ability to reconfigure, disable, or weaken the sandbox at runtime. Ant allows users to create scripts that execute Java classes during a build under a user-specified permissions

```
<permissions>
  <grant class="java.security.AllPermission"/>
  <revoke class="java.util.PropertyPermission"/>
</permissions>
```

Figure 3.4: An Apache example of an Ant build script element to grant all but one permission. This results in a defenseless security manager; thus revoking one permission does not lead to application security.

set. Figure 3.4 shows an example permission set from the Ant Permissions website.[9] The `grant` element provides the application all permissions, while the `revoke` element restricts the application from using property permissions. This example policy leads to a defenseless security manager, thus malicious code can easily disable the sandbox and perform all actions, including those requiring `PropertyPermissions`. Although this policy is only an example, its existence suggests possible confusion on the part of either its author or its consumers about appropriate security policies for untrusted code.

Ant saves the current manager and replaces it with a custom manager before executing constrained external code. The custom manager is not initially defenseless when configured with a strict security policy, but contains a private switch to make it so for the purposes of restoring the original manager. Ant therefore catches applications that perform actions restricted by the user while typically protecting sandbox settings. However, it is not clear this implementation is free of vulnerabilities. Netbeans similarly sets a security manager around separate applications.

Both of these cases require a defenseless security manager, otherwise the application would not be able to change the current security manager. Similar to the case in Section 3.3.3, Java provides an "orthodox" mechanism to achieve this goal while aligning with intended sandbox usage: a custom class loader that loads untrusted classes into a constrained protection domain. This approach is more clearly correct and enables a self-protecting sandbox.

An attempt to solve a similar problem in Freemind 0.9.0 illustrates the dangers of a defenseless manager. Freemind is a mind mapping tool that allows users to execute Groovy scripts on such maps (Groovy is a scripting language that is built on top of the JRE). A Java application that executes such a script typically allows it to execute in the same JVM as the application itself. As a result, a specially-crafted mind map, if not properly sandboxed, could exploit users that run its scripts.

Freemind implements an architecture that is intended to allow the sandbox to enforce a stricter policy on the Groovy scripts than on the rest of Freemind. The design centers around a custom security manager that is set as the system manager in the usual manner. This custom manager contains a private field holding a proxy manager for script execution. In this design, all checks to the security manager are ultimately deferred to the proxy manager in this field. When this field is set to `null`, the sandbox is effectively disabled

[9] https://ant.apache.org/manual/Types/permissions.html

```
/** By default, everything is allowed. But you
 * can install a different security controller
 * once, until you install it again. Thus, the
 * code executed in between is securely
 * controlled by that different security manager.
 * Moreover, only by double registering the
 * manager is removed. So, no  malicious code
 * can remove the active security manager.
 * @author foltin */
public void setFinalSecurityManager(
  SecurityManager pFinalSecurityManager) {
  if(pFinalSecurityManager == mFinalSecurityManager){
    mFinalSecurityManager = null;
    return;
   }
   if(mFinalSecurityManager != null) {
     throw new SecurityException("There is a SecurityManager
       installed already.");
   }
   mFinalSecurityManager = pFinalSecurityManager;
 }
```

Figure 3.5: Initialization of Freemind's security manager, including a custom proxy. In the code comment, "double registering" refers to repeating the sequence of setting the security manager. The implementation is similar to using a secret knock on a door as a form of authentication. This demonstrates two problems with the sandbox as used by developers: (1) using Java policies as a blacklist is dangerous and (2) modifying the manager at run time requires a work-around (ineffective or incomplete, in this case) to defend against malicious users.

```
def sm = System.getSecurityManager()
def final_sm = sm.getClass().getDeclaredField("
   mFinalSecurityManager")
final_sm.setAccessible(true)
final_sm.set(sm, null)
new File("hacked.txt").withWriter { out -> out.writeLine("HACKED!"
   ) }
```

Figure 3.6: Exploit that breaks out of the scripting sandbox in Freemind to execute arbitrary code. The sandbox grants the `ReflectPermission("suppressAccessChecks")` permission, thus the exploit can make a security critical private field accessible to disable the security manager. Had this permission not been granted, the exploit would have to use a different tactic.

even though the system's manager is still set to the custom manager. Given a properly configured sandbox, scripts cannot access this field.

Figure 3.5 shows how Freemind sets the proxy security manager field. Once a manager is set, if `setFinalSecurityManager` is called again with a different security manager, a `SecurityException` is thrown, but calling the method with a reference to the set manager disables the sandbox. The comment implies that this sequence of operations was implemented to prevent malicious applications from changing the settings of the sandbox.

Freemind sets a proxy security manager to stop unsigned scripts from creating network sockets, accessing the file-system, or executing programs before initiating execution of a Groovy script. The manager grants all other permissions by overriding permission checks with implementations that do nothing, thus any script can turn off the sandbox.

We demonstrated that the custom security manager is easily removed using reflection to show that the problem is more complex than simply fixing permission checks related to setting the security manager. Figure 3.6 shows a Groovy exploit to turn off the manager. The script gets a reference to the system's manager and its class. The class has the same type as the custom security manager, thus the exploit gets a reference to the proxy manager field. The exploit makes the field public using a privileged operation the sandbox policy should not have granted, thus enabling the exploit to reflectively `null` it, disabling the sandbox to allow "forbidden" operations. We notified Freemind developers of this vulnerability in August of 2014 and offered our advice in achieving their desired outcome. They acknowledged the report, but have not fixed the issue.

All of these applications ran afoul of the Java sandbox's flexibility even though they attempted to use it for its intended purpose. Given their current designs, they must all be run with defenseless managers (most can be redesigned to remove this requirement), and those that manipulate the set security policy dynamically do so problematically. While Java does provide the building blocks for constraining a subset of an application with a policy that is stricter than what is imposed on the rest of the application, it is clear that it is too easy to get this wrong: We've seen no case where this goal was achieved in a way that is known to be free of vulnerabilities. These case studies support our general claims that

the Java security mechanisms are overly complex, and that this complexity contributes to security vulnerabilities in practice.

3.4 Fortifying the sandbox

Based on our study of how open-source Java programs interact with the security manager, we propose two changes to the current Java security model to stop exploits from disabling *self-protecting* managers. These rules can be applied to all Java applications. The rules reduce the flexibility and thus complexity of the Java security model without breaking backwards compatibility in practice:

Privilege escalation rule. If a self-protecting security manager is set for the application, a class may not directly load a more privileged class. This rule is violated when the protection domain of a loaded class implies a permission that is not implied in the protection domain that loaded it.

Security manager rule. The manager cannot be changed if a *self-protecting* security manager has been set by the application. This is violated when code causes a change in the sandbox's configuration, the goal of many exploits.

In this section, we evaluate the protection merits and backwards compatibility of these rules through an implementation of runtime monitors that enforce them. This evaluation was done in collaboration with a large aerospace company. Section 3.4.1 discusses how we implemented our runtime monitors, Sections 3.4.2 and 3.4.3 explain the methodology behind and results of experiments that evaluate the rules' ability to stop exploits while maintaining backwards compatibility.

3.4.1 Implementation using JVMTI

JVMTI is a native interface that enables the creation of dynamic analysis tools, called agents, such as profilers, debuggers, or thread analyzers. JVMTI agents can intercept and respond to events such as class or thread creation, field access or modification, breakpoints, etc. We utilize the ability to monitor class creation and field accesses to implement and evaluate the efficacy of our rules.

Enforcing the privilege escalation rule

To enforce the Privilege Escalation rule, our agent stops a program when a class is loaded to check for privilege escalation. The existence of restricted-access packages complicates this implementation slightly. *Restricted-access packages* are technically public but are intended only for internal JRE use. Benign applications can (and often do) use JDK classes to access these restricted implementations (e.g., much of the functionality in `java.lang.reflect` is backed by the restricted-access `sun` package). We therefore allow the JRE itself to load restricted-access packages at runtime, but prevent such loading from the application classes. Because exploit payloads are not implemented in restricted-access JRE packages, the Privilege Escalation rule can permit this standard behavior while preventing attacks.

Note that there are two ways that application code might directly access such packages in the current security model: (1) exploit a vulnerability in a class that *can* access them or (2) receive permission from the security manager via an `accessClassInPackage("sun")` permission in the policy. The first behavior is undesirable, and is thus rightfully prevented by the enforcement of this rule. The second behavior would require a defenseless manager.

Enforcing the security manager rule

We enforce the Security Manager rule by monitoring every read from and write to the field in the `System` class that stores the security manager (the `security` field). The agent stores a shadow copy of the most recently-set security manager. Whenever the field is written, the agent checks its shadow copy of the manager. If the shadow copy is `null`, the manager is being set for the first time. If the new manager is self-protecting, the agent updates the shadow copy. If not, the agent stops performing stringent checks because the rule does not apply in the presence of a defenseless manager.

The agent checks modifications to the security manager and validates it when it is referenced. The latter is necessary to catch type confusion attacks, which change the manager without triggering a JVMTI modification event. The tool detects unauthorized changes every time the manager is used by comparing the current manager with the shadow copy for changes. Type confusion attacks that masquerade as a privileged class loader will not be detected by our agent, and may still be dangerous when exploited in collaboration with other JRE vulnerabilities.

Performance Challenges and JVM Integration

JVMTI requires turning off the just-in-time compiler (JIT) to instrument class field writes, which is required to enforce the security manager rule. This slows down program execution enough that our monitors are not suitable for adoption with current JVMTI implementations. We speculate that it is possible to work around these performance issues, but likely at the cost of substantially increased implementation complexity. Watches are currently implemented using a simple but slow hook in the bytecode interpreter. Instead, the JIT could insert additional instructions in generated code to raise an event when the field is accessed by JITed code. However, this may interfere with code optimization passes that would otherwise be performed. Furthermore, it is not possible to simply instrument an application's bytecode to implement our rules because our instrumentation would run at the same privilege level as the application, but the rules require privileged access to Java security settings (e.g. the policies assigned to specific protection domains). Given these barriers, a more correct and general approach is to embed our rules directly in the JRE. We have reached out to OpenJDK's `security-dev` mailing list about doing so, but it is not clear the approach has been seriously considered.

Table 3.4: Effectiveness test results. The exploits are taken from the subset of Metasploit 4.10.0 that apply in modern environments and follow a drive-by-download paradigm. Taken together, the proposed security model restrictions stop all tested exploits.

	Monitor	
CVE-ID	**Privilege Escalation**	**Both**
2011-3544	Attack Succeeded	Attack Blocked
2012-0507	Attack Blocked	Attack Blocked
2012-4681	Attack Succeeded	Attack Blocked
2012-5076	Attack Succeeded	Attack Blocked
2013-0422	Attack Blocked	Attack Blocked
2013-0431	Attack Blocked	Attack Blocked
2013-1488	Attack Succeeded	Attack Blocked
2013-2423	Attack Succeeded	Attack Blocked
2013-2460	Attack Blocked	Attack Blocked
2013-2465	Attack Succeeded	Attack Blocked

3.4.2 Effectiveness at fortifying the sandbox

We evaluated our rules' ability to block sandbox-disabling exploits on ten Java 7 exploits for the browser from Metasploit 4.10.0, an exploit development and delivery framework. We ran the exploits on 64-bit Windows 7 against the initial release of version 7 of the JRE. The ten exploits in question include both type confusion and confused deputy attacks. Metasploit contains many Java exploits outside of the subset we used, but the excluded exploits either only work against long obsolete versions of the JRE or are not well positioned to be used in drive-by downloads. Our results thus show whether our rules can stop the vast majority of current exploits.

We ran the exploits (1) without the agent, (2) with the agent but only enforcing the Privilege Escalation rule, and (3) while enforcing both rules. We tested the Privilege Escalation rule separately because while the Security Manager rule stops all the exploits on its own, the Privilege Escalation rule stops exploits earlier, has significantly less overhead, and can detect attacks that are not explicitly targeting the manager. Table 3.4 summarizes our results. All ten of the exploits succeeded without the agent. The Privilege Escalation rule stops four of them. All ten were stopped when both rules were enforced.

Together, the rules are capable of stopping current exploit tactics while narrowing available future tactics by blocking privilege escalation exploit routes.

3.4.3 Validating Backwards-Compatibility

By construction, the rules do not restrict benign behavior in the applications we studied in Sections 3.2 and 3.3. To mitigate the threat of overfitting and increase the generalizability of our results, we also executed the monitors on the applications in Table 3.5. This set is composed of benign JWS applications that, like applets, are automatically sandboxed.

Table 3.5: Backwards compatibility dataset. The commit column shows the date of commit we investigated, which was also the most recent commit at the time of the experiment.

Name	Description	KLOC	Workload	Commit
ArgoUML	UML Tool	389	1244 test cases	1/11/15
Costello	GUI Tester	closed source	9 provided examples	5/09/12
CrossFTP	FTP Client	closed source	GUI fuzzing, sample workload	1/18/15
OpenStreetMap	Map Editor	343	406 test cases	1/18/15
JabRef	Reference Manager	148	3 provided examples	3/11/14
mucommander	File Manager	106	27 test cases	1/23/14

This expands the scope of our results beyond the desktop applications studied above and evaluates our proposed modifications in context (JWS programs are typically run with restrictive security policies). The set also includes closed-source applications, providing evidence that our results generalize beyond open source.

For each program, we confirmed that the agent does not negatively affect benign workloads. ArgoUML, JavaOpenStreetMap, and mucommand contained unit tests that we ran in the presence of our monitors. Costello, and JabRef did not provide tests, but do provide example workloads that we used in their place. CrossFTP contained neither tests nor sample workloads, thus we fuzzed the GUI for 30 minutes using a custom fuzzer and uploaded a file to a remote FTP server as a sample workload.[10]

In each case, we confirmed that the tests, sample workloads, or fuzzed executions worked without a security manager. To sandbox the applications, we developed security policies using a custom security manager that does not throw exceptions and that prints out checked permissions as each program executes. Finally, we ran each case a third time using our policy and the standard Java security manager with our monitors attached and enforcing the rules. The rules did not break any unit tests, sample workloads, or fuzzed executions.

Finally, to validate our rules on representative desktop applications, we confirmed the agent does not break programs in the DaCapo Benchmarks v9.12-bach set (Blackburn et al., 2006). DaCapo systematically exercises each application using a range of inputs to achieve adequate coverage. For all but one case, we set a security manager that granted all permissions and attached our monitors to application execution; we let Batik set its own security manager because it exits if it cannot do so. Our rules did not break any DaCapo applications.

3.5 Limitations and validity

Limitations. Neither of the rules we propose in Section 3.4 will stop all Java exploits. While the rules catch all of the exploits in our set, some Java vulnerabilities can be exploited

[10]GUI fuzzing source code can be found at `https://goo.gl/ccTLVR`.

to cause significant damage without disabling the security manager. For example, our rules will not detect type confusion exploits that mimic privileged classes to perform their operations directly. However, our rules substantially improve Java sandbox security, and future work will be able to build upon these results to create mitigation techniques for additional types of exploits.

Internal validity. Our study results are dependent on accurately studying the source code of applications and their comments. In most cases, security manager interactions are easily understood, but there are a few particularly complex interactions that may be misdiagnosed. Furthermore, we did not review all application code, thus we may have taken a comment or some source code out of context in larger applications. Finally, using two different reviewers may lead to variations in the interpretations of some of the data.

We mitigated these threats by using a checklist, FindBugs plugin, and JVMTI agent to provide reviewers with consistent processes for reviewing code and validating their results. Furthermore, we inspected entire source files that contained security manager operations. We tested our tools and processes in a pilot study to find and mitigate sources of inconsistencies.

External validity. The study only includes open-source applications. It is possible that closed-source applications interact with the security manager in ways that we did not see in the open-source community. However, we inspected a few small closed-source applications with our aerospace collaborators. We did not find any code that suggested this is the case. This result is further supported by the closed-source programs included in the dataset in Section 3.4.3.

Reliability. While the majority of the study is easily replicable, GitHub search results are constantly changing. Using GitHub to generate a new dataset would likely result in a different dataset. Furthermore, over the course of the study, one application either became a private repository or was removed from GitHub (Visor).

3.6 Related work

As far as we are aware, no study has examined Java applications' use of the sandbox. However, several recent studies have examined the use of security libraries that can be overly complex or misused, discovering rampant misuse and serious vulnerabilities. Georgiev *et al.* uncovered vulnerabilities in dozens of security critical applications caused by SSL library protocol violations (Georgiev et al., 2012). These applications misconfigured high-level libraries such that the high-level libraries misused low-level SSL libraries, which in turn failed silently. Somorovsky *et al.* demonstrate vulnerabilities in 11 security frameworks such that Security Assertion Markup Language (SAML) assertions are not checked properly in the face of certain API mis-orderings (Somorovsky et al., 2012). Li *et al.* examined browser-based password managers and found that many of their features relied on an incorrect version of the same-origin policy, which could allow attackers to steal user credentials (Li et al., 2014b).

Our rules increase the security of the sandbox by effectively removing unnecessary features. Prior work has taken a different approach, proposing to re-implement the Java

sandbox or add to it to increase security. Cappos *et al.* created a new sandbox structure involving a security-isolated kernel separating sandboxed applications from the main system (Cappos et al., 2010b). They validated this structure by translating past Java CVEs into exploits for the new kernel. Provos *et al.* describe a method of separating privileges to reduce privilege escalation (Provos et al., 2003). Their approach is partially implemented in the Java security model. Li and Srisa-an extended the Java sandbox by providing extra protection for JNI calls (Li and Srisa-an, 2011). Their implementation, Quarantine, separates JNI accessible objects to a heap which contains extra protection mechanisms. The performance of their mechanism is also measured using DaCapo. Siefers *et al.* created a tool, Robusta, which separates JNI code into another sandbox (Siefers et al., 2010b). Sun and Tan extend the Robusta technique to be JVM independent (Sun and Tan, 2012).

Java applets are the most common ways to transmit Java exploits. Detectors have been created to identify drive-by downloads in JavaScript (Cova et al., 2010), and in Adobe Flash (Ford et al., 2009). Helmer *et al.* used machine learning to identify malicious applets (Helmer et al., 2001). Their approach monitored system call traces to identify malicious behavior after execution. However, this approach is entirely reactive. Our approach terminates exploits when they attempt to break out of the sandbox, before they perform their payloads. Schlumberger *et al.* used machine learning and static analysis to identify common exploit features in malicious applets (Schlumberger et al., 2012). Blasing *et al.* detect malicious Android applications using static and dynamic analyses of sandboxed executions (Blasing et al., 2010). Unlike these automated approaches, our rules show that a better understanding of benign sandbox interactions can inform unique mitigation strategies.

3.7 Conclusion

Li Gong, the primary designer of the Java security architecture, admitted in a ten year retrospective on Java Security that he did not know how or how extensively the "fine grained access control mechanism" (i.e. the Java sandbox) is used (Gong, 2009). Our study shows how and how extensively the sandbox is used in open source software, identifies unnecessary functionality that enables sandbox-escaping exploits, and discovers developer-facing complexity hampering use of the sandbox.

Our empirical study of open-source applications supports the hypothesis that the Java security model provides more flexibility than developers use (or likely need) in practice. The study also strongly suggests that the model's complexity leads to unnecessary vulnerabilities and bad security practices. We further validated the findings of our study by defining two rules, which together successfully defeated Metasploit's applet exploits without breaking backwards compatibility.

We take several general lessons from these findings. First, Java should provide simpler alternative mechanisms for various common goals, such as constraining access to global resources or adapting to multiple security contexts. We found that developers sometimes use the sandbox to prevent third party components from calling `System.exit()`, a specific instance of a more general development problem: frameworks often need to enforce

constraints on plugins (e.g., to ensure non-interference). We also observed that developers who attempted to make non-trivial use of the sandbox often do so incorrectly, even though the functionality in question could theoretically be implemented correctly within the current model (albeit with increased complexity). One promising approach is to allow programmers to temporarily strengthen security policies (e.g. by adding a permission).

We observed that many developers struggle to understand and use the security manager for any purpose. The complexity involved in applying the sandbox is perhaps why there were only 36 applications in our sample. Some developers seemed to misunderstand the interaction between policy files and the security manager that enforces them. Others appear confused about how permissions work, not realizing that restricting just one permission but allowing all others results in a *defenseless* sandbox. Our concerns here are shared by the IntelliJ developers, who include static analysis checks to warn developers that a security expert should check all security manager interactions (IntelliJ, 2014). In general, sandbox-defeating permissions should be packaged and segregated to prevent accidentally defenseless sandboxes. Finally, some developers appear to believe the sandbox functions as a blacklist when, in reality, it is a whitelist. When considered as a whole, our results and observations suggest that the model itself should be simplified, and that more resources—tool support, improved documentation, or better error messages—should be dedicated to helping developers correctly use the sandbox.

In the next chapter we overcome many of these developer-facing challenges through the use of tools. The tools help developers recover a security policy from Java bytecode, refine the policy, ensure the policy contains no dangerous permissions, and automatically apply it. This tooling was heavily motivated by the lessons learned in this chapter. Indeed, it is unlikely the tooling would even be considered if we had not fortified the Java sandbox first. At first glance, many people assume the tooling is useless due to prior security issues with the sandbox, but these issues are solved by the project discussed above.

Chapter 4

MAJIC: Machine-Assisted Java Isolation and Containment[1]

In the previous chapter we empirically analyzed how the Java sandbox is used in open source applications. The results enabled us to fortify the sandbox to stop many classes of sandbox-escaping exploits. We observed a number of cases where developers struggled to create sufficient security policies and failed to sandbox subsets of their applications. In every case where developers were attempting to sandbox application subsets they were acting to encapsulate third party components.

As discussed in Chapter 1, it is rare to build a modern application without making use of externally developed components. Given the risks discussed, it is desirable to limit the behaviors of these components to just those operations we understand and need. To achieve this ideal, we must define a security policy that fully captures the operations a set of code is allowed to perform. How do we define such a policy and ensure it completely describes the guarded computations without granting unnecessary privileges? This is a challenge we address in this chapter.

The Java sandbox infrastructure provides no tooling or easily consumed data for deriving a security policy from a set of code. The building blocks are present to express and enforce a policy, but it is left to developers to use these blocks effectively. Ideally we would redesign all or parts of the sandbox API to solve these issues. However, the required changes are invasive and would have to overcome political and technical inertia at great cost. These changes would impact developers years later if at all. Instead, we provide tooling to cope with the existing means. We design, implement, and evaluate a tool to achieve machine-assisted Java isolation and containment (MAJIC).[2] Our tool, MAJIC, helps developers perform every step involved in applying the sandbox to Java applications without requiring alterations to the JRE (some of the related work in Section 4.7 requires JVM modifications).

MAJIC uses program analysis and fine-grained policy merging to create security poli-

[1]This chapter was written in collaboration with Jonathan Aldrich, William Scherlis, and Joshua Sunshine.

[2]MAJIC's "J" is pronounced the same as the "J" in Java, thus MAJIC is pronounced the same way as "magic".

cies for subsets of Java applications. Subsets are defined by users as sets of Java archives (JARs), classes, or a mix of both. While we cannot always fully recover policies by analyzing program code, our tool provides code completion, syntax highlighting, correctness checking, subset refactoring, and traceability between permissions and code to help developers refine their policy. Once the policy is finished, MAJIC transforms the application's bytecode to enforce the policy using the standard Java sandbox.

Chapter 3 identified several points of friction in applying the sandbox and identified unnecessary design and implementation complexity that enables current sandbox-bypass techniques (Cifuentes et al., 2015). The sandbox as applied by MAJIC is properly configured without the need for developers to grapple with these complex features. In addition, MAJIC warns developers when common mistakes are detected, such as using one of the potentially dangerous permissions discussed in the previous chapter.

We worked with two large companies to identify real applications using libraries and frameworks practioners wanted to sandbox, and we sandboxed them using MAJIC. These engagements allowed us to vet the efficacy of MAJIC's operations and its applicability to problems the industry currently faces. We measured the overhead of our transformations by sandboxing an existing benchmark framework. This case study used a nearly worst-case scenario for our tools: the majority of the method calls cross the boundary between unsandboxed and sandboxed code. In this case, the transformations add 6.6% overhead for code that straddles this boundary on top of the 6.9% the Java sandbox introduces. It is not possible to assess an average or best case due to the fact that our transformations add overhead when unsandboxed code calls sandboxed code. The number of calls that cross this boundary are dependent on the application and the subset that is sandboxed within it.

This chapter contributes the following:

- An exposition of the complexity involved in manually sandboxing entire Java applications or their subsets (Section 4.1).
- Algorithms for statically and dynamically recovering required security permissions from Java bytecode and merging them into usable policies (Section 4.2).
- Tooling to help developers refine recovered policies and impose them on Java applications (Sections 4.3 and 4.4)
- An evaluation of MAJIC's efficacy, applicability, and the overhead it introduces through a series of case studies using real applications from industrial collaborators in Section 4.5

We discuss our tool's limitations in Section 4.6, related work in Section 4.7, and conclude in Section 4.8.

4.1 Background

In the previous chapter, Section 3.1 provided a high level summary of the Java sandbox limited, to content necessary to understand sandbox-escaping exploits. This section describes the Java sandbox in more detail (Section 4.1.1) including how it is typically used and the challenges involved in applying it manually to applications or their subsets (Section 4.1.2).

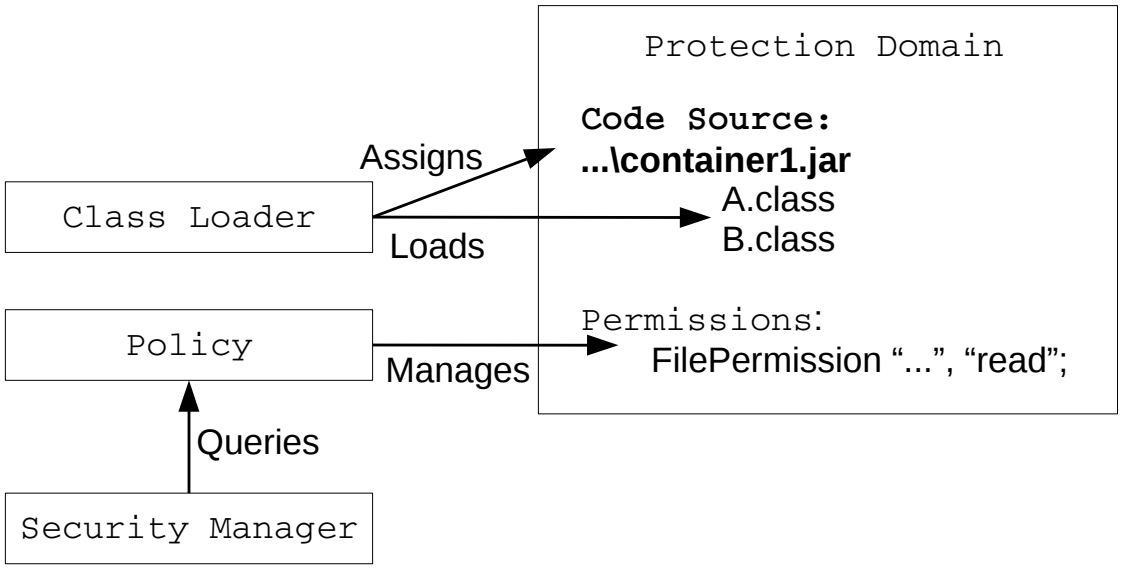


Figure 4.1: Summary of developer-facing Java sandbox components.

4.1.1 The Java Sandbox

The Java sandbox imposes security policies on sets of classes, thus allowing developers to limit the behaviors that entire applications or their subsets can manifest at run time. The operations of application-accessible sandbox components are summarized in Figure 4.1.

Classes are placed in a sandbox when they are loaded by a *class loader*. When the class loader is preparing a class for execution, it assigns the class a *code source* to identify where the class originates. The code source is a URL that refers to either a directory or a JAR file. Code sources perform the important function of identifying what *protection domain* a class belongs to (i.e. all classes with the same source are in the same domain). A protection domain contains a set of *permissions*. Permissions grant classes in a protection domain the ability to perform sensitive operations such as writing to a file or connecting to a remote server. Typically, permission sets are specified and assigned to one or more code sources (and thus protection domains) in a file written in the Java policy language (Oracle, 2015c). These sandbox pieces provide all of the data required to enforce a security policy.

To enforce a policy, a method that performs a sensitive operation queries the *security manager* to ensure that every class in the current call stack has the permission required for the operation (Wallach and Felten, 1998b). Methods in the Java class library (JCL) perform most of these queries, but an application designed to use the sandbox may query the manager as well. Ultimately, a *policy* object that was instantiated when the JVM was initialized is queried to get the permissions for each protection domain represented by classes in the call stack. If the permission check fails a `SecurityException` is thrown, otherwise the security manager returns with no side-effects.

This design is flexible and extensible. A wide variety of sandbox components can be extended to enforce nearly any security policy in many scenarios. For example, it is easy to create new permissions to enforce tailor-made security policies, and the security manager

can be extended to alter when and how policies are enforced.

Java's sandbox is typically applied to an application automatically. For example, a security manager is set before any external code is loaded by a JVM initializing applet or Java Web Start (JWS) environments. Applications that run in these environments may be developed with no knowledge of how the sandbox is configured or instantiated beyond understanding the default limitations imposed on them (e.g. no filesystem access). In fact, benign applets have no control over the sandbox's configuration or instantiation, and JWS applications may only alter the configuration if they are digitally signed and the user allows elevated access.

In Chapter 3 we found that, at least in the open-source community, it is rare for applications to use the sandbox directly and common for those that do to use the sandbox incorrectly. Why are so few developers leveraging these powerful and flexible sandbox components to improve their application's security posture, and why do they often fail when they try? The next section explains how the complexity of the sandbox foils its use even in seemingly simple cases.

4.1.2 Manually Sandboxing Components

Generally speaking, the sandbox should be used to encapsulate components that may be vulnerable or malicious. It is likely that both risks are present where third-party components are in use. Therefore, given the popularity of component repositories such as Maven Central, almost any modern Java application would likely benefit from use of the sandbox. A developer sandboxing an entire application at once faces adversity from the first step. This developer must successfully create a policy that allows the application to run while avoiding a policy that is over-privileged. The latter is of particular concern because the JRE ships with several powerful permissions, any one of which (when granted) can enable sandboxed code to disable the sandbox (see Section 3.1.2 in the previous chapter).

While tooling could solve most related challenges, there are few mechanisms to help developers create a security policy given an application to sandbox, and there are even fewer mechanisms for helping them understand the risks presented by the permissions they are granting. While there is documentation that discusses the risk of using various JDK permissions (Oracle, 2014d), this documentation assumes understanding of JRE internals and does not highlight the most dangerous permissions that should essentially never be granted. Thus, developers are left to sort through security policy issues almost entirely by hand.

A busy developer that must create a security policy has two options: (1) infer permissions through manual inspection or (2) infer permissions through dynamic analysis (Inoue and Forrest, 2005).

In the first case, she must read the documentation for existing permissions, then inspect the application's source code to infer which permissions may be required. These permissions are added to the policy. When the policy seems reasonably complete she can check her work by running various executions of the application to see if a `SecurityException` occurs. This exception indicates the policy was violated and is therefore missing a necessary permission. This approach is fraught with problems: She may infer permissions that are

not actually required (potentially allowing unwanted operations), she may miss permissions which will result in execution failures, sandbox exceptions may be silently caught, and the approach does not scale to large applications she cannot cost-effectively inspect.

In the second case, she must inspect permissions used at run time – this can be done by creating a security manager that prints out each permission that is checked without throwing exceptions, configuring the JRE to use it, and running many executions of the application to collect required permissions. This strategy will not lead to over-privileged policies, but even with a comprehensive testing strategy there is no guarantee the policy contains the permissions required for all desired executions of the program.

A developer sandboxing an entire application is essentially finished at this point. However, a developer sandboxing a subset of an application must still filter discovered permissions for the subset and assemble sandbox components to impose the policy on just the subset. For the latter developer, creating the policy is likely the easy step. Even when given a security policy, a developer is unlikely to successfully impose the policy on a subset of a large, complex application.

There are two possible cases for imposing a policy on an application subset. Either all of the classes to sandbox are in identifiable JARs that are free of classes that should be unconstrained, or not. In the former case the developer can use the policy language to assign the policy to just the code sources that refer to targeted JARs. However, this condition is unlikely to be true unless the application was intentionally designed for it to be true. Must an unsandboxed class extend a sandboxed class? Is there a custom class loader in the sandbox that must load code into an even stricter sandbox? Do custom exceptions need to cross sandbox boundaries? If the answer is yes to these or similar questions, sandboxing the selected components is difficult, if not impossible, to resolve using the standard sandbox components. While not every application will have these concerns, it is our experience that they are common in large, complex applications. Complex applications are also more likely to use several third party components and would thus benefit even more from sandboxing than a simple, relatively easy to sandbox application.

Solving these challenges typically requires developers to create custom class loaders per sandbox, unless they are willing to alter the JVM or directly change the JCL. Either change would make their application substantially more difficult to deploy and maintain, particularly as few knowledgeable users are willing to have their entire environment changed for one application. To avoid JRE changes, the loaders must define protection domains based on the type of the class loader instead of using coarse grained code sources. This ensures that inclusion in a protection domain is at the granularity of a class instead of a JAR or directory. However, the standard component to manage policies does not use this construct, thus it must be extended to recognize finer grained policy assignment.

To sandbox a library under these conditions, the library's classes must be loaded by a custom class loader into the intended protection domain. These classes must be moved outside of the class path or they will be loaded outside of the sandbox by the application's class loader. However, this move will cause type checking issues for application classes that use library classes because the latter are no longer available statically. This problem is solved by refactoring the application classes to reflectively use the library code.

Given all of these challenges, of course developers fail to use the sandbox correctly!

Fortunately, many of these steps can be performed by tools. The remainder of this chapter discusses how to mechanically recover policies, manually refine them, and use tooling to impose them securely.

4.2 Recovering a Security Policy

To use the Java sandbox, we first need a security policy to define the behaviors that code is allowed to exhibit. Security policies are not explicit in Java code in a format that can be easily consumed, hence the previously discussed processes for inferring permissions. However, instead of constructing a policy by hand we can use program analysis to recover required permissions from Java bytecode. Java permissions are classes that extend the JCL's `Permission` class. They are typically constructed with parameters to define the entity the permission grants access to and the specific actions that will be allowed on the entity. For example, a `FilePermission` takes in a parameter to define what file the permission grants access to, and which actions such as reading, writing, and deleting can be performed on it. Sometimes, however, the entity or actions are fixed by the permission type. For example, the entity for a `SecurityPermission` is always the JVM's security subsystem, thus we must only specify actions such as the ability to set a security policy.

Permission types can be recovered using static analysis because any method that must guard a sensitive operation must also query the security manager to ensure permission is granted before performing the operation. However, some initialization parameters are unknown until run time in cases where the entity or action is dependent on events generated by the user or external configuration. Thus, static analysis can recover every type of permission a body of Java code needs unless permissions are reflectively loaded (we have never seen this case) or code is obfuscated, but we can not always recover every initialization parameter. While not a complete policy, this is still a useful start. Knowing the required permission types gives the user a basis for discovering unknown parameters and for knowing how many parameters need to be recovered. Furthermore, static analysis can be combined with dynamic analysis to recover many, if not all, of the missing parameters.

Our static analysis is closely related to the analysis presented by Centonze et al. (2007), but we reduce precision to achieve acceptable performance at scale. Centonze combines the static analysis algorithm with a dynamic analysis that automatically runs each method that could cause a permission check. This collects permissions that are definitely required at runtime while rejecting false positives, which are more common than in MAJIC's case due to the greater precision. Unlike Centonze's solution, MAJIC does not automatically exercise the application's code and instead requires the user to exercise the application manually or, preferably, using a testing suite. This grants MAJIC's user greater control over the executions that will be analyzed, which we have found to be useful in case studies that make extensive use of configurable plugin systems.

In this section, we discuss how to statically and dynamically recover permissions from Java code in Sections 4.2.1 and 4.2.2 respectively. In Section 4.2.3 we discuss how permissions recovered using the different techniques are merged using permission types, initialization parameters, and stack traces to form a policy.

4.2.1 Static Permission Recovery

Every permission check starts with a call to one of the security manager's many methods, but ends at one of two versions of a method called `checkPermission`. These methods are in the security manager, accept the generic `Permission` type as a parameter, and are ultimately responsible for using internal sandbox components to determine whether or not a permission is granted. Thus, to statically recover permissions we use a control flow analysis to find calls to any method in the security manager whose name is `checkPermission` originating from a class that is to be sandboxed. Permission checks are often directly initiated by classes in the JCL. Thus, classes in the class library must be analyzed if they are used by a sandboxed class or its dependencies.

Furthermore, the permission check must appear in the analyzed execution trace between the sandboxed method and any later call to `doPrivileged`. The sandbox ignores any method calls in a stack trace that happened before a call to `doPrivileged` (i.e. the method signals to the sandbox code that walks the stack that methods called before the signal are not relevant). Thus, a permission is not relevant if `doPrivileged` is called after the sandboxed method but before the permission check. Hardcoded permission parameters can be recovered using a dataflow analysis after relevant permission checks have been identified.

An interprocedural dataflow analysis inspects the full path between a sandboxed method and its relevant permission check to identify objects constructed from subclasses of `Permission`. The permission types and their constant constructor parameters are stored in a simulated stack. At the program point where `checkPermission` is called, the analysis determines which of the stored permissions is being checked by inspecting the permission argument on the simulated stack. Used permissions and the stack trace formed by the methods the analysis followed are stored to form the security policy later. The traces are used to help the user understand where recovered permissions originate in the code and, as discussed later, to merge permissions.

In practice, MAJIC performs the control- and dataflow analyses and context sensitive constant propagation at the same time, on bytecode, and using the ASM library (Bruneton et al., 2002). Performing the analysis on bytecode is convenient because the user may not have the source code available locally for their default JRE or even for the component they want to sandbox. Combining the steps also reduces overhead because work would otherwise be duplicated.

All classes in the sandbox and their non-JCL dependencies are analyzed independently, thus permissions used by sandboxed code that is called reflectively will be found. Reflection in the JCL is generally not a problem because permission checks tend to happen before these types of complications arise. However, if a permission is missed by this analysis it can be picked up by the dynamic analysis later.

In general, reflection is not a challenging technical issue when using the sandbox or MAJIC. The challenge is that users of the sandbox and MAJIC must be aware of classes that (1) should be sandboxed and (2) are available for unsandboxed classes to dynamically load and use. We do not help the user ensure they have accounted for all potentially loaded classes that are also candidates for sandboxing. The architecture of the sandbox ensures that classes dynamically loaded by a sandboxed class will also be sandboxed (this

is discussed further in Section 4.4.2).

4.2.2 Dynamic Permission Recovery

To recover permissions at run time we extend the standard library's security manager to override `checkPermission`. MAJIC's implementation of this method records the type of the permission that is being checked, its parameters, and the current stack trace. This data is sent to MAJIC where stack traces for new permissions are checked to determine whether they contain a call to a method in a sandboxed class. Irrelevant permissions are discarded. As depicted in Figure 4.2, MAJIC helps users launch their application under the monitoring of the extended security manager. This strategy assumes that applications will not replace our security manager with their own. Our experience and the analysis in Chapter 3 suggest this is a reasonable assumption because most applications do not replace an already configured manager.

4.2.3 Merging Permissions

Static and dynamic analysis recover a large number of possibly duplicated permissions. Duplicate permissions have the same type, appear in the same execution traces, and either have identical initialization parameters or share one initialization parameter with one or both missing the other.

MAJIC compares the stack traces for permissions to determine if they appear in the same execution. Stack traces are sets of traces that include a class name, source file name, method name, and line number for each call between a permission check and either the first method in the application (dynamic analysis) or a sandboxed method (static analysis). MAJIC compares stack traces to ensure they contain identical traces in the same order starting from the permission check and ending at the last method call in the shorter trace.

Duplicate permissions satisfy conditions that are formalized below, where A and B may be duplicate permissions. These conditions formalize the conditions that duplicates have the same type, come from the same execution trace, and either share both parameters or have one parameter in common with one permission missing the mismatched parameter. These conditions are sufficient because they only label permissions duplicates if either the permissions are identical in every way or they are different only because they are the same permission recovered starting from different points in the same execution trace.

For example, if A and B are duplicates aside from (1) A missing a parameter B has and (2) A missing a few methods B has early in the stack trace, the missed parameter is the result of A being recovered later in the execution trace and therefore missing the initialization of the parameter. In this case, A will have a stack trace that is a proper subset of the B's stack trace with identical ordering between elements in the traces. This happens, for example, when the dynamic analysis recovers a permission starting from the main method but the static analysis recovers it starting from a sandboxed method.

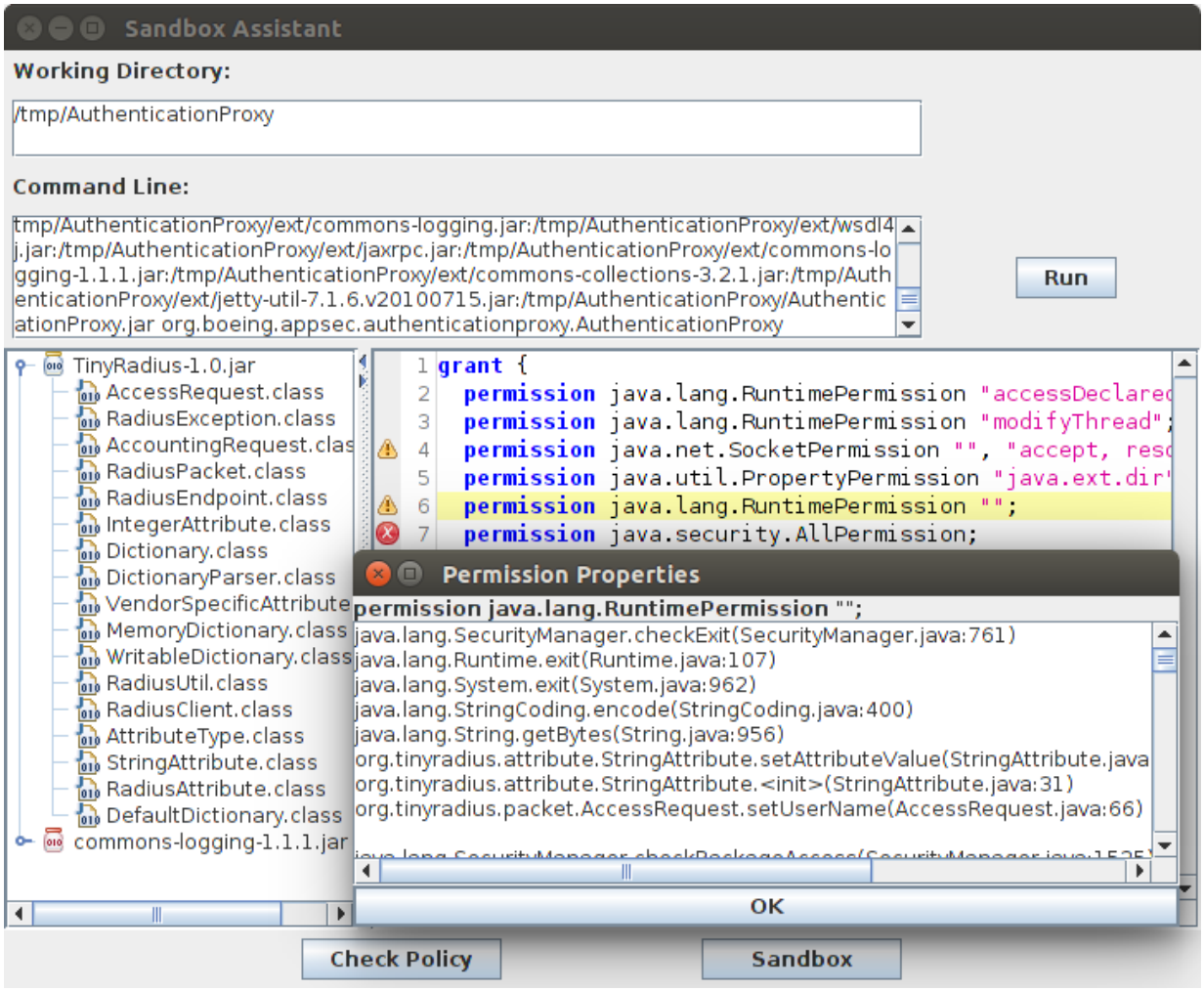


Figure 4.2: The sandbox assistant helps users create, refine, and impose a security policy. Dynamic analysis controls are in the top third of the window and policy refinement controls cover the rest.

$$A = B \iff type_A = type_B \wedge (stackTrace_A \subset stackTrace_B \vee stackTrace_B \subset stackTrace_A) \wedge paramsMatch$$

$$stackTrace_A \subset stackTrace_B \iff trace1_A = trace1_B \wedge trace1method_A = checkPermission \wedge trace2_A = trace2_B ... \wedge traceN_A = traceN_B \wedge N = length(stackTrace_A)$$

$$paramsMatch \iff (entitiesExistAndMatch \wedge actionsExistAndMatch) \vee (entity_A = entity_B \wedge action_A = \emptyset \vee action_B = \emptyset) \vee (entity_A = \emptyset \vee entity_B = \emptyset \wedge action_A = action_B)$$

$$entitiesExistAndMatch \iff (entity_A = entity_B) \wedge (entity_A \neq \emptyset)$$

$$actionsExistAndMatch \iff (action_A = action_B) \wedge (action_A \neq \emptyset)$$

When the condition holds, duplicate permissions are merged, favoring the more specific permission if they are not strictly duplicates (i.e. they do not have identical types, stack traces, and identified parameters). For example, if two permissions are equal aside from the fact that one is missing an entity, the permission missing the entity is discarded because it is a duplicate of a permission that is more specific. The more specific permission has both parameters and can be used by the sandbox without additional effort.

The merging process produces a policy in the standard Java policy language that the user may refine before imposing it on the application. This policy can be imposed on an entire Java application or it can be applied to an application subset by MAJIC. To use the policy for subsets in the ideal case discussed in Section 4.1 the user must define code sources in the policy.

4.3 Refining the Security Policy

Section 4.2 described how to recover security policies given a Java application or its subset. This policy may be sufficient for the desired executions already, but typically the policy needs manual refinement.

Relatively simple applications can potentially execute code paths that require undesirable permissions. For example, some operations require the `String` class to encode a string using the `StringCode` class. The latter calls the `exit` method in the `System` class if a required encoding is unavailable, which requires a `RuntimePermission` to grant the code the right to shut down the virtual machine. Not only is the particular error condition rare, it introduces a relatively powerful permission the developer may reasonably decide to not grant. As a result, the developer must review the generated policy to see if there are any permissions worth removing. However, it is more common that there are permissions present that have missing parameters. This often results from permissions recovered using static analysis from code paths, such as the `StringCode` example, that are so rare that even a comprehensive testing suite run with the dynamic analysis won't help recover the parameters.

MAJIC provides an editor, depicted in Figure 4.2, to help the developer refine the policy. The user may exclude classes from the subset of code to be sandboxed, in which case all of the permissions present solely because of the excluded item disappear from the policy (items can be re-included to bring permissions back). The user may edit the current policy in a text editor that provides auto-completion with pop-up documentation for all JDK permissions and syntax highlighting for the standard policy language. The editor allows the developer to check the policy, which will mark each permission that is missing a parameter or has invalid syntax with a warning icon. Permissions that grant the sandboxed code the ability to disable the sandbox are marked with an error icon.

This editor collapses some permissions that are not merged as duplicates (see Section 4.2.3). Permissions that look identical may appear in the policy even dozens of times if they are not collapsed. These permissions are not merged because their stack traces are different and each set of traces may provide useful insights, but merging the permissions would lose all but one of the stack traces. Instead, the editor shows this case as one permission but the developer can right click a permission to view all of the stack traces where the permission appears. This traceability provides vital clues in cases where parameters are missing. The traces tell the developer exactly which code causes a permission to be required, which may be all of the information a knowledgeable person needs to infer missing parameters. Otherwise, the developer can use the trace to learn exactly where to look in the applications source- or bytecode to discover the parameter(s).

A developer who has finished refining the policy may then use MAJIC to impose it. At this stage, MAJIC ignores any permission that is missing parameters or whose syntax is invalid within the standard Java policy language.

4.4 Imposing the Security Policy

In Sections 4.2 and 4.3 we describe how to develop a security policy for Java code. A policy does nothing unless it is imposed on code to stop unwanted operations. Imposing an existing security policy on an entire Java application requires no special effort. This can be done by specifying standard virtual machine switches on the command line. However, the process to impose a policy on a subset of a Java application is currently difficult, error

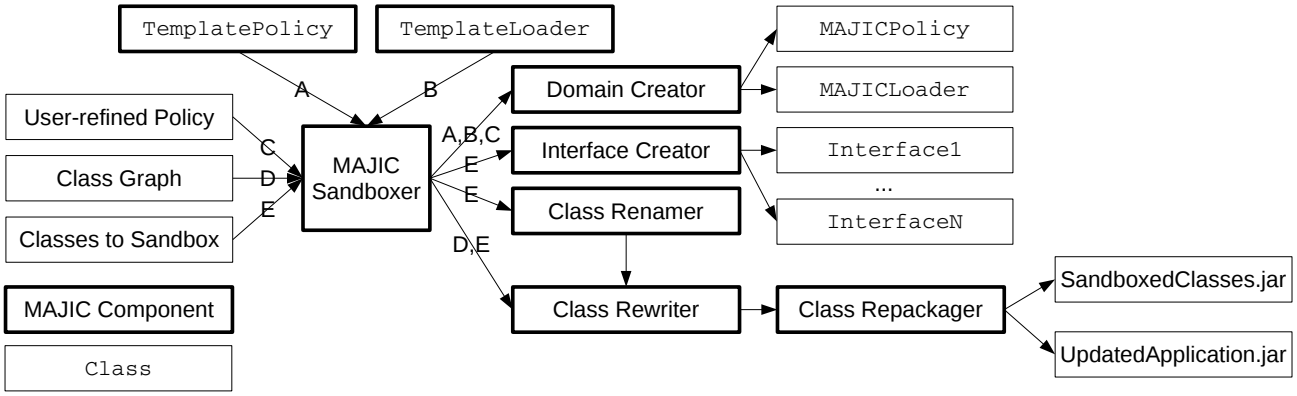


Figure 4.3: MAJIC's sandboxer carries out all of the operations required to sandbox a subset of a Java application, aside from those that would also be required to sandbox an entire application. Labeled arrows show how inputs are mapped from the sandboxer, which orchestates the process, to subcomponents that perform specific sandboxing operations. For example, the dependency graphs of all classes in the application (D) and the set of classes to sandbox (E) are fed into the class rewriter. These inputs cause the rewriter to transform all sandboxed classes and the classes that depend on them.

prone, and entirely manual.

In this section, we overview a process for sandboxing fine-grained subsets of applications. Our design does not require changes to the JRE and automates nearly all of the steps that are unique to sandboxing subsets of applications. We go on to describe how the various steps of the sandboxing process are implemented in MAJIC.

4.4.1 Design Overview

As summarized in Figure 4.3, we break the sandboxing process into five reusable bytecode transformations managed by an orchestrator called the *sandboxer*. The sandboxer automatically carries out the manual process described in Section 4.1.2 for sandboxing an application subcomponent.

The sandboxer requires three inputs from the user: (1) the recovered and refined policy (Sections 4.2 and 4.3 respectively), (2) the application's class path,[3] and (3) the subcomponent to sandbox. The process proceeds as follows:

1. A *domain creator* writes a custom policy class (Section 4.4.2) to initialize each protection domain with its permission set and creates a class loader (Section 4.4.3) for each protection domain.
2. The `main` methods are updated to set the JVM-wide policy object to the one generated in step 1.
3. An *interface generator* produces an interface containing all non-static externally accessible fields and methods for each sandboxed class. These are used in place of

[3]MAJIC generates dependency graphs for classes in the class path.

sandboxed types outside of the sandbox to satisfy the type checker.

4. A *class renamer* changes the package name for each sandboxed class to ensure package names are unique to each sandbox and will therefore not interfere with class loading.[4] The class renaming is not strictly necessary if we are careful to remove the original copies of the classes after copying them to the sandboxed JAR. Still, renaming the sandboxed classes eases debugging (e.g. by making it easier to spot sandboxed class references in bytecode).
5. A *class rewriter* transforms sandboxed classes to implement the interfaces generated in step 3. Additionally, externally accessible methods are transformed to use the interfaces as types for parameters and return values whose types are sandboxed (Section 4.4.4).
6. Unsandboxed classes that use sandboxed classes are re-written to load sandboxed classes using the correct class loader generated in step 1. Virtual and interface calls to sandboxed methods are re-written to use the generated interfaces and static method calls are rewritten to use reflection (Section 4.4.4).
7. Sandboxed classes are packaged into their own JAR file and any unsandboxed classes are updated in their respective containers. Digital signatures are stripped from JAR files that are updated with transformed classes.

A report is generated during the process that summarizes each executed transformation. A sample report appears in Appendix A. The report includes a task list that details remaining steps. These steps are either already well supported by existing tools (e.g. signing JARs) or would require complicated tooling to perform tasks the user can quickly and easily perform manually (e.g. modifying scripts that can be written in one of many different scripting languages). These steps typically include:

- Move generated sandbox components into the application's class path and update class path definitions in launcher scripts or manifest files
- Move JAR files that contain sandboxed classes to a user-specified location outside of the class path
- Add virtual machine command line switches to launcher scripts to set: a policy, a security manager,[5] and the `majic.sandboxed.loc` property to the location of sandboxed class JARs
- Re-sign JAR files that were signed before contained classes were updated

Finally, because a security manager is set, the JVM will attempt to enforce a security policy on classes the user did not specify as sandboxed. MAJIC generates a standard Java policy file with `AllPermission` for these classes. This policy may be customized by the

[4]The `URLClassLoader` expects to find classes with the same package name in the same JAR file, which would not be the case if one of those classes is sandboxed without renaming. To avoid this challenge we would have to create our own fully featured class loader, which is more complicated and risky than simply renaming classes.

[5]The required switches are identical to those used to sandbox an entire application. MAJIC generates these switches and places them into the report, thus the user must simply copy and paste them.

user to constrain the entire application. Figure 4.4 shows the structure of an example application before and after sandboxing.

It is important to note that much of the complexity described in this section cannot be avoided without alterations to the JVM. Unless an application is carefully designed to be sandboxed, the standard sandboxing interface is too course grained to sandbox a typical Java application without the complexity and bookkeeping MAJIC automates.

4.4.2 Policy Class Generation

At any given time there can only be one `Policy` object managing the policies for an application. This object loads permissions from an implementation-specific policy store (policy files by default), returns permission sets for a given code source or protection domain, and allows other sandbox components to check if a permission is implied by a particular domain. Typically a domain is defined by a course-grained code source. To achieve the required granularity without JRE changes, we must extend the policy class.

MAJIC contains an extended version of the policy class that accepts the current policy object as a parameter at construction to save the default behavior. The methods to get permission sets and check if permissions are granted are overridden to check the identity of the class loader for the class whose policy is in question. Assuming the class loader is a known sandbox loader, the permission sets for that loader are used instead of using the default behavior. Our extended version must initialize permission sets that belong to our protection domains and use the new sandbox loaders to identify those domains.

Additionally, MAJIC's policy class is updated to ensure sandboxed class loaders are also used to identify the domain in which they are loaded. This secures the use of reflection because any loaded and reflected external code will be sandboxed if we sandbox the loader. Furthermore, our policy object does not simply check the current loader that loaded a class, it checks every class loader in the loader chain starting from the current loader and ending at the bootstrap loader. A class is assigned to the domain for the first sandbox loader that is found. Due to the search order, the assigning loader is the closest in the loader chain to the one that loaded the class. Thus, if there are nested sandboxes the class is guaranteed to appear in the innermost sandbox represented in the loader chain. If no sandbox loader is found, the policy defers the request to the original policy object. Both updates ensure class loaders cannot be used to bypass the sandbox.

Finally, the main methods of sandboxed applications are updated to instantiate a MAJIC policy object, pass the current policy object to the constructor, and set the new policy object as the sole policy for the application.

4.4.3 Class Repackaging and Loading

Sandboxed classes are moved out of the class path, otherwise the application class loader can load them outside of the sandbox at any time (Oracle, 2015b). After the move, the application must load the classes using a class loader whose identity will be used to identify each class's protection domain. The standard sandbox supports the use of multiple sandboxes enforcing different security polices, where each sandbox has its own

protection domain. MAJIC creates one class loader per domain. These loaders subclass `URLClassLoader` and have names that uniquely identify their domains, but they otherwise do not extend the superclass.[6]

Some class loaders delegate the loading of classes in a specific package to the first loader that loaded a class in that package, thus the sandboxed classes must be renamed. MAJIC appends the name of the class loader that will load a sandboxed class to the end of the class's package name.

4.4.4 Re-writing Method Calls

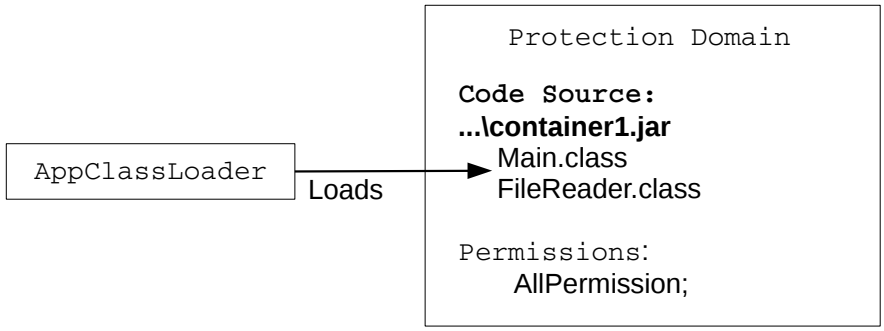
Previous sections laid out the infrastructure required to place classes into sandboxes. We now discuss how unsandboxed classes interact with sandboxed code. For these interactions we must modify the bytecode of unsandboxed classes. Our modifications ensure that calls to sandboxed methods use the sandboxed version of the method-owning class instead of the original version. We make these changes on bytecode for two reasons: (1) we want MAJIC to fulfill its role even in cases where source code is not available and (2) the required transformations introduce extensive use of reflection that would make modified source code difficult and risky to maintain.

Sandboxed classes need to be transformed in a few simple ways. References to sandboxed types need to be remapped from their original names to their sandboxed names, the classes need to implement interfaces generated based on their externally accessible methods and fields, and any externally accessible method must be transformed to use the generated interfaces as return and parameter types where sandboxed types appear. The externally accessible methods are not further transformed beyond casting parameters of sandboxed types to their class version from the interface version at the beginning of the method and casting the return type from the class type to the interface type before returning. These transformations reduce the overall number of transformations required for sandboxed code while ensuring it can be used reflectively from unsandboxed code. An alternative approach would require MAJIC to transform all sandboxed classes the same way unsandboxed classes are transformed (described below). This would grossly and unnecessarily complicate the tools.

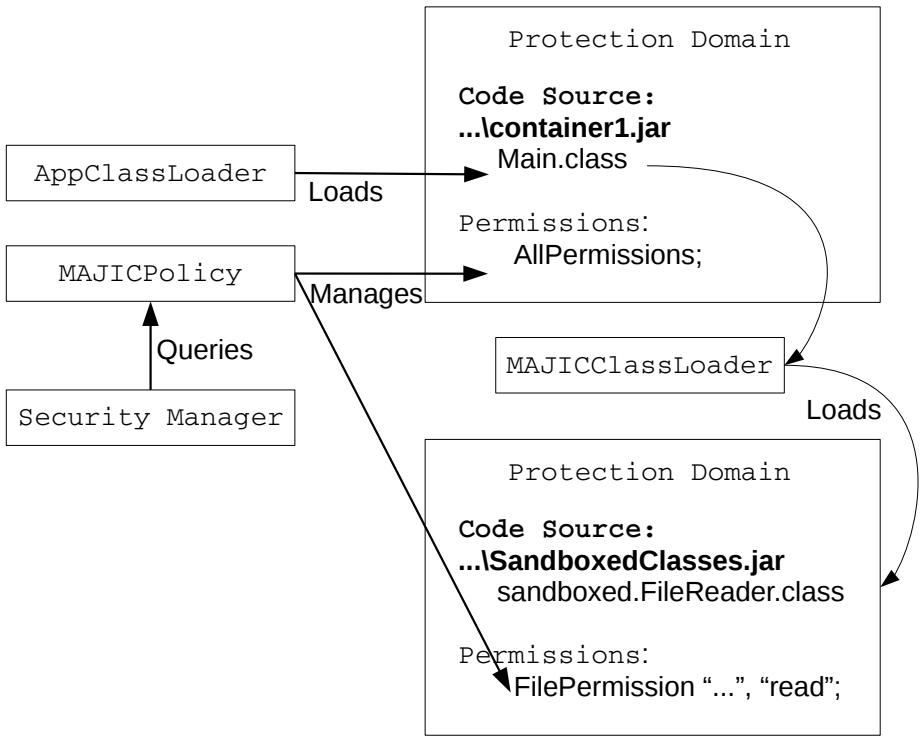
Outside of the sandbox there are broadly two types of method calls that must be transformed: static calls and non-static calls. To handle all of these cases, two helper methods are added to every unsandboxed class using sandboxed code. The first method accepts a JAR and class file name and uses the appropriate class loader to load the named class from the JAR. MAJIC knows which loader is appropriate because it knows which classes are being sandboxed and to which domain(s) they belong. The second method is used to reflectively lookup and return a reference to a constructor with a given signature for a class.

In cases where a sandboxed class is instantiated from unsandboxed code the bytecode that creates the new object and calls the constructor is replaced with calls to our helper

[6]We use a well established class loader with no modifications because class loading is complicated and may introduce vulnerabilities. Furthermore, existing class loaders contain optimizations we do not want to lose.



(a) Unsandboxed



(b) Sandboxed

Figure 4.4: In (a) the application is not sandboxed and every class exists in the same protection domain. This is the typical case for a Java application that runs on the desktop. In (b) the `FileReader` class has been sandboxed by using a custom classer loader with our custom policy to achieve least privilege.

methods. The helper methods load the class into the sandbox and fetch the constructor that was originally called. Additional code is inserted to reflectively call the constructor, cast the new object to an interface type generated for the now sandboxed class, and store it on the stack as before. Calls made to this object are changed to interface calls.

There are two potential complications when instantiating the sandboxed version of the class. First, the constructor's parameters may be placed on the stack before or after creating the new object or a mix of both. This can make it difficult to ensure the stack is not corrupted while transforming class instantiation. Second, primitive types cannot be passed as parameters to reflective method calls. These problems are solved using the same process. All constructor parameters on the stack before or after a new object is created are saved using local variables and restored on the stack immediately before the constructor is called. Primitive types are boxed as they are restored.

Static calls are relatively easy to replace using the existing helper methods. All static method calls are replaced with code to load the sandboxed version of the method-owning class, reflectively fetch and call the method, and place any return value on the stack.

Having fixed the method calls, sandboxed methods may still throw an exception that is itself sandboxed. An unsandboxed method may attempt to either catch a sandboxed exception or specify that it throws the exception, which will cause a type checking error. The interface solution used in other similar cases cannot work because interfaces must subclass only `Object` but exceptions must be of type `Throwable`. Thus, MAJIC converts a sandboxed exception's type to a `MajicException` that provides generic messages for unsandboxed methods that specify they throw a sandboxed exception and catch clauses outside of the exception's sandbox. This preserves the semantics of catch clauses, but loses the semantics of the exception itself. We did not see cases where sandboxed exceptions have semantics beyond specific error messages, but a future implementation could account for cases with richer semantics by calling the sandboxed exception's methods from the `MajicException`.

4.5 Evaluation

We evaluate MAJIC's ability to assist in complex cases where real developers want to use the Java sandbox. In particular, we used the findings in Chapter 3 to select use cases (e.g. sandboxing plugins) where open source applications attempted to use the sandbox for security purposes but failed. We then worked with industrial collaborators to find applications that could benefit from sandboxing in similar ways. We chose this strategy to: 1. Validate that the challenging use cases exist outside of the individual cases in the open source community, 2. Create opportunities to test our tools with interested and consistently available users from our target audience, and 3. Avoid architectural issues caused by failed sandboxing attempts in existing applications.

To effectively assist, MAJIC must recover most of an applied security policy statically and dynamically and leave transformed code runnable, secure, and performant. If it does not meet these requirements it is potentially better to instead pursue invasive JVM and JCL changes that avoid critical technical limitations. We evaluate these aspects of MAJIC

Table 4.1: The number of permissions in the final policy and counts for parameters recovered statically, dynamically, and manually for case studies unrelated to performance. The union of the three sets of recovered parameters covers all parameters required for the policy. Manually recovered parameters were added by the user in cases where the program analyses could not recover the data.

Case	**# Perms**	**# Stat**	**# Dyn**	**# Man**
TinyRadius	6	6	1	1
Jetty	28	26	22	0
IoT	15	12	2	7
FreeMind	8	8	0	1

through a series of case studies. We sandboxed an authentication library in an application developed by a large aerospace collaborator (Section 4.5.1). At their request, we equipped and lightly trained one of their developers who went on to sandbox a different library using MAJIC (Section 4.5.2). Finally, we worked with a large engineering and electronics company to sandbox all plugins in an Internet of Things framework (Section 4.5.3).

We measured the degree to which policies are recovered statically, dynamically, and manually in these three studies to evaluate MAJIC's ability to recover policies. First, we ran the static analysis to recover all permission types and counted the number of permissions that had at least one parameter recovered. Afterwards, we ran the dynamic analysis and counted how many of the permissions had previously missing parameters recovered. Finally, we counted all permissions that were still missing a parameter as cases that need to be completed manually. The counts for these cases are summarized in Table 4.1. It was rare to remove any recovered permissions, but in cases where they were removed it was because the static analysis recovered permissions for operations the code could perform that are not actually required by the particular application under study (e.g. due to library functions that are not used). The dynamic analysis did not recover new permissions, but was often useful for filling in missing permission parameters.

To ensure transformed programs remain runnable, MAJIC uses the bytecode verifier in ASM after transformations are complete to statically ensure that classes in the sandboxed application will pass the JVM's bytecode verifier. We also ran each case to ensure desired executions still function correctly. To evaluate the security of MAJIC's transformations, we built a Java agent to dump the names of classes and their loaders as a program executes. We used this agent with each sandboxed application to ensure classes appear in the expected protection domains. Finally, we sandboxed a benchmark plugin in DaCapo to measure the overhead of MAJIC's transformations (Section 4.5.5).

4.5.1 Sandboxing a Third-Party Library

Our aerospace collaborator has carried out a series of projects to replace the use of passwords with two-factor authentication mechanisms such as smart cards and one-time password tokens. One of these projects targeted thick client applications that communicate

with non-web servers. We'll refer to this project as Auth+. The Auth+ team did the difficult and security critical work of creating a facade library to interface with the enterprise's various two-factor authentication systems. Targeted applications use Auth+ to authenticate client users to the application's server before the server fulfills requests. One of the supported authentication mechanisms requires the RADIUS protocol, which Auth+ supports via a third-party library called Tiny Radius.[7]

While Tiny Radius is open source, it makes an excellent sandboxing target because it contains enough code (just under 5000 LOC) that it could reasonably hide a backdoor, it was developed entirely by a third party, and it has access to authentication credentials that, if stolen, could cause grave damage to the enterprise.

Tiny Radius manifests several properties that make it relatively easy to sandbox with MAJIC. The library is mostly self-contained with few entry points and little need for application code to extend library classes. Furthermore, the entry points to the library are always used in the same way, thus variances in user inputs are not consequential to the resulting security policy. However, the library does contain a custom exception that must be caught outside of the sandbox. We were able to recover and impose a security policy containing six permissions in less than 5 minutes. Due to the ease and speed with which MAJIC can sandbox this case, we used Auth+ to evaluate the efficacy of policy recovery and MAJIC's transformations from the standpoint of security.

We replaced the version of Tiny Radius in Auth+ with a malicious version. Our version exfiltrates any credentials it receives to a malicious actor via the network. We tried two cases: A. where the trojan was crafted to hide from policy recovery and B. where the trojan was not crafted to hide. In case A, permissions required for the trojan did not appear in the policy, thus when the trojan executed in our experiment a `SecurityException` was thrown and the trojan was not allowed to execute. In case B, the permissions required by the trojan appear in the policy. We provided this policy to an Auth+ developer and asked them if the policy met their expectations. The developer spotted a permission they did not expect while reviewing the policy, which turned out to be a permission for the trojan. However, had the rouge permissions gone unnoticed the trojan would not have been impeded by the sandbox.

Auth+ supports applications with high availability requirements but an exception when malicious code executes crashes the program. Thus, a trojan could cause grave damage simply by trying to execute without success. We provided an option to wrap unsandboxed code that calls sandboxed code in try/catch blocks to catch exceptions that result from policy violations and output a warning. With this option, the compromised version of Auth+ continues to work securely for authentication mechanisms that do not rely on Tiny Radius, but those that do will fail at the point that the malicious code executes without harming non-malicious executions and without compromising the credentials.

[7] `http://tinyradius.sourceforge.net/`

4.5.2 Users Sandboxing a Third Party Library

The previous exercise inspired a developer named Allen at the aerospace company to try using MAJIC on his own. We include details of Allen's experience as anecdotal evidence in support of MAJIC's efficacy. Allen had five years of development experience in Java and worked as an Application Security Engineer. We gave him a copy of MAJIC, spent 30 minutes walking him through an 18 slide presentation summarizing the intent behind MAJIC, how it works at a high level, and its basic operation. We concluded this session with a quick demo of the previous exercise. This introduction was enough to enable him to be successful.

Without further support from us, Allen sandboxed five Jetty 7[8] embedded web server components constituting 76 KLOC in a legacy application that uses the library to embed a web service. This exercise took him 45 minutes, most of which was spent investigating the application to determine the proper subset to sandbox. The latter was complicated by the fact that the application extends some of the classes in Jetty, and the subclasses had to be placed in the Jetty sandbox to ensure the proper function of the application. We obtained a copy of the sandboxed application to evaluate using our Java agent.

Allen offered the following remarks highlighting the value of policy recovery, "MAJIC's ability to read through the code and automatically populate which permissions you need to allow and put that in policy file I feel is something I couldn't live without even if it didn't do other tasks." Regarding the entire process, "With a little bit of navigation around the GUI tools and some understanding of the Java project structure I was able to create a Java Security policy to sandbox a piece of our Java project that is out of date and therefore has some dangerous security vulnerabilities in it. I would recommend MAJIC and use it again as it takes a very complex problem and greatly simplifies it."

4.5.3 Sandboxing All Framework Plugins

We collaborated with a large engineering and electronics firm that is building an Internet-of-Things framework (called IoT after this point). We selected this case to evaluate MAJIC's ability to sandbox dynamically discovered and reflectively loaded framework plugins, where the plugins are developed by third parties. IoT allows users to plugin in services that use data from various sensors and context from the user to produce a useful output. For example, one service uses `food2fork.com` to look up popular recipes by ingredient or meal for a user that states she is hungry. Yet another service learns the user's preferred ambient temperatures at different times of the day to ensure the room the user is in is at an appropriate temperature. These services are developed by third parties and potentially gain access to sensitive data about the user, thus they are attractive to sandbox.

IoT uses the Spring Framework[9] to load service beans for a Spring context. Spring contexts are essentially modules constructed by instantiating and configuring classes based on a developer-defined configuration. Thus, to sandbox the plugins we sandboxed the JARs containing Spring's bean and context classes, which constitute 63 KLOC. However,

[8] `http://www.eclipse.org/jetty/`
[9] `https://spring.io/`

this is not entirely sufficient because Spring loads beans for a context using the thread's context class loader by default instead of using the caller's loader (in this case, the sandbox loader). The thread's loader is often the application's default class loader, thus classes are loaded outside of the sandbox by default. To ensure Spring loads classes into the sandbox, we modified MAJIC to set a sandbox loader for Spring contexts to use if the application did not set a specific loader. In the latter case, the developer must sandbox the specified loader.

In this case tools cannot define a definitive policy. The framework has a plugin API, but plugins also use external APIs. Thus, recovering a policy solely from the framework's API code is not sufficient to define the range of acceptable behaviors, nor is it sufficient to simply analyze a set of plugins. We used both options to develop half of the policy in this cause, but we still needed to define a standard policy to use the same way a standard policy is used for all Java applets. We used the policy refinement mechanisms discussed in Section 4.3 to develop the other half of the policy.

In particular, we communicated with the collaborator to determine what plugins must be allowed to do at a high level to meet the framework's functional requirements. For example, we learned plugins must not send data over unencrypted channels, thus the sandbox's policy grants the ability to send data over the network only on the port used for HTTPS. IoT also contains a specific data directory for plugins to use, and the sandbox does not allow them to access the filesystem outside of that directory. In this case, MAJIC left the framework runnable, sandboxed, and performant.

4.5.4 Sandboxing a Vulnerable Component

In Section 3.3.4, the previous chapter discussed vulnerable uses of the Java sandbox that were supposed to secure the use of application subcomponents. In one particular case, an open source mind mapping tool called FreeMind attempted to use the sandbox to encapsulate user-created Groovy scripts embedded in mind maps. We disclosed a vulnerability in their use of the sandbox and provided a sandbox-bypassing exploit. In this section we discuss how we sandboxed Groovy in FreeMind using MAJIC.

FreeMind's feature set is augmented using a custom plugin system, and the Groovy script feature is implemented as a plugin. Unlike in the previous section, we cannot simply sandbox a specific class loader to sandbox the execution of Groovy scripts both because we don't want to sandbox all plugins and because the plugin system uses a class loader from the JCL that cannot be targeted without affecting other programs. Instead, we must sandbox Groovy and the subset of the scripting plugin that initiates the execution of scripts.

The scripting plugin contains the core of its implentation in a class called `ScriptEngine`. This class contains an internal class that extends `GroovyShell` to execute scripts. Static analysis recovered all of the permissions required to fulfill FreeMind's requirements by analyzing this internal class and Groovy, with the need to manually recover only one parameter. After manually altering FreeMind to remove the vulnerable sandbox implementation, we used MAJIC to successfully fulfill FreeMind's encapsulation requirements. This stopped a variation of our exploit as depicted in Figure 4.5. We had to modify the exploit because

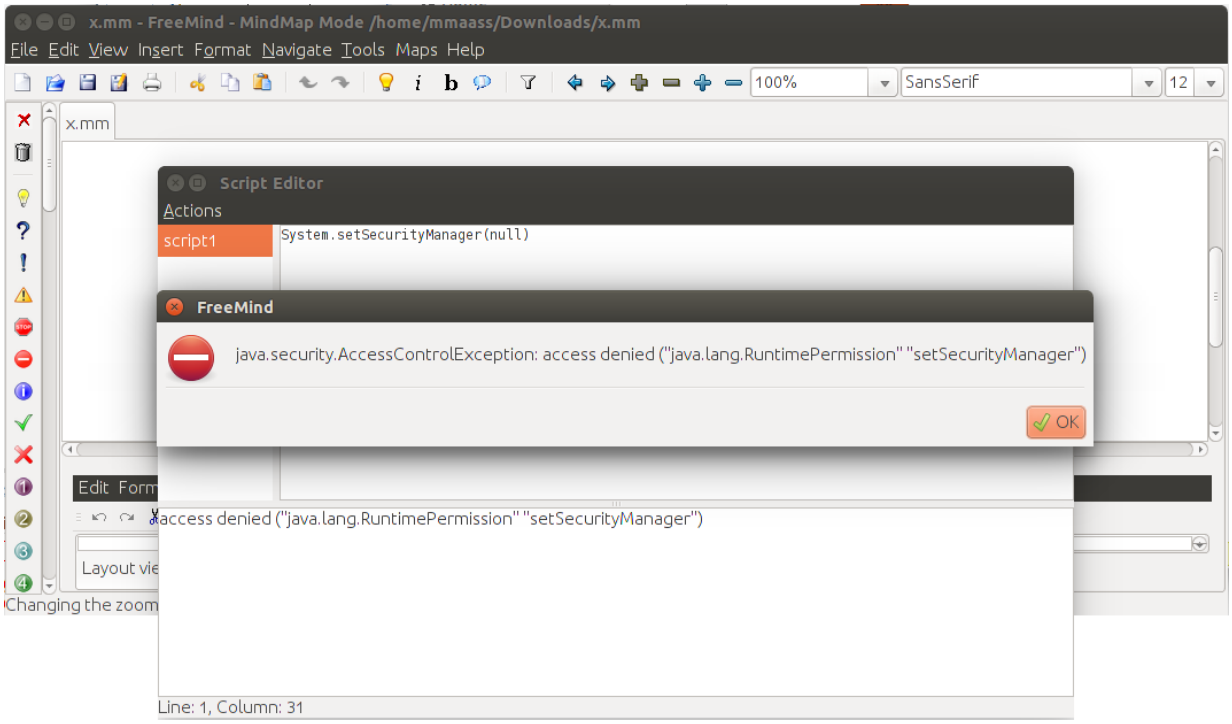


Figure 4.5: FreeMind incorrectly used the sandbox to encapsulate user-created Groovy scripts. We removed the vulnerable implementation and used MAJIC to securely sandbox FreeMind's script plugin. Unlike the original implementation, our sandbox cannot be directly disabled.

our original exploit was tailored to a custom and vulnerable security manager we removed.

4.5.5 Targeting Plugins and Performance

MAJIC's transformations add overhead by (1) converting virtual method invocations into slower interface invocations (StackOverflow, 2015), (2) requiring additional type casts to convert between interface and class types for sandboxed classes used across sandbox boundaries, and (3) complicating the instantiation of sandboxed classes in unsandboxed code. As a result, overhead will primarily appear at the border between unsandboxed and sandboxed code. To measure this overhead, we sandboxed a library in a benchmark in DaCapo 9.12-bach (Blackburn et al., 2006) where the benchmarked application's code makes many calls into the library.

We used HPROF (Oracle, 2015a) to measure the number of calls that could potentially cross from sandboxed to unsandboxed code under different sandboxing scenarios for each DaCapo benchmark separately and for DaCapo as a whole. We chose to target Xalan, which transforms XML to HTML, because the majority of the methods called during Xalan's execution appear in classes from a separately packaged serialization library. Thus, by sandboxing this library we achieve nearly worst-case overhead, which would not be the case for any other scenario we explored.

Table 4.2: The running times and overhead for the Xalan benchmark run in DaCapo unaltered, uniformly sandboxed, and with just a serialization library in Xalan sandboxed. The Xalan figures include the overhead from the Java sandbox in addition to MAJIC's transformations.

Case	Time (ms)	Overhead
Not Sandboxed	4623	-
All Sandboxed	4943	6.9%
Library Sandboxed	5245	13.5%

The three cases in Table 4.2 allowed us to measure how much overhead the Java sandbox in general adds to isolate the overhead added by our transformations. We found that simply setting a security manager that enforces a policy with the `AllPermisssion` adds 6.9% overhead and our transformations bring Xalan's overall overhead to 13.5%.

Our measurements took place on a ThinkPad T440s Ultrabook with 12 GB of RAM and a quad core 64-bit Intel Core i7-4600 processor at 3.3 GHz with a 4 MB cache. OpenJDK 7u79 was used to run Java applications.

4.6 Limitations

MAJIC automates much of the sandboxing process to the point that cases essentially impossible to correctly sandbox by hand can now be handled correctly and quickly. However, there are several difficult cases that still remain. Due to the complex architectures implemented in many modern Java applications, it is possible for developers to specify subsets to sandbox that cause class linking and loading issues at run time. For example, many large applications make use of custom class loaders that interfere with MAJIC's sandbox class loader placement for some subset definitions. This is manually fixed quickly once the nature of the conflict is understood, but it would be better to handle them automatically. Additionally, many applications couple classes spread across several containers through complicated inheritance relationships, making it less obvious which classes must be in the sandbox.

Many instances of these issues can be solved using a class loader monitor, such as the one we used during our evaluation, that shows precisely which loaders are loading which classes and from where. However, applications themselves sometimes get in the way of debugging. The JVM's class loading and linking exceptions have messages that are difficult to understand. Applications rightfully mask or wrap these exceptions to provide more friendly error messages, but this often hides what actually went wrong along with the information required to begin debugging the problem.

Finally, while our design supports the creation of protection domains for distinct application subsets, we did not write all of the code required to achieve this because the feature is not necessary to evaluate the approach. Our prototype allows the user to pick and sandbox only one subset. Managing multiple protection domains would require MAJIC

to provide the user with a fine-grained way to manage domain assignment for sandboxed classes. Furthermore, dependencies would need to be carefully managed to ensure type checking issues do not result from the user's desired subsets, which is a more complicated problem when dealing with a potentially arbitrary number of domains. It is also potentially possible to extend this design to support nested sandboxes (e.g. by allowing a loader in one sandbox to loaded classes into an internal sandbox), which is not a case supported by the standard Java sandbox interface.

4.7 Related Work

While little tool-supported sandboxing research has been done, others have worked in this domain. Inoue and Forrest (2005) developed a tool for dynamically discovering the permissions that a Java application needs, but it does not target subsets of applications or support policy merging as MAJIC does. Schreuders et al. (2013b) developed the functionality-based access control model for applying hierarchical parametrized policies to applications built in any programming language based on their functional features. Application features are determined for policy assignment using statically discoverable data, such as imported libraries. Well adopted system-level Sandboxes such as Systrace (Provos, 2003), SELinux (Loscocco and Smalley, 2001), and AppArmor (Cowan et al., 2000) ship with dynamic analysis tools to help develop starter policies.

Much of the work targeting the Java sandbox has increased the scope of the sandbox without making it easier to use. Bartoletti et al. (2009) extended the Java security model to support history-based access control with security policies enforced by an execution monitor woven into bytecode. Like MAJIC, this allows user to sandbox subsets of applications without JVM changes, but it reflects a large departure from the standard Java security model. Autrel et al. (2013) extended Java policies to support organization-based access controls. Quarantine was created to isolate and protect Java objects accessible via the JNI (Li and Srisa-an, 2011). Siefers et al. (2010b) extended the Java sandbox to isolate native code via software-based fault isolation and to ensure policies enforced by a security manager also apply to native code. Sun and Tan (2012) extended Siefer's mechanism to make it compatible with additional JVMs. Cappos et al. (2010b) developed a mechanism to sandbox the standard library in Python and analyzed the effect the technique would have on Java's sandbox. Unlike MAJIC, these extensions require significant JVM changes or replace much of the Java security model.

While Android sandboxes applications and does use Java, it does not include the standard Java sandbox. Android combines various mechanisms in its application framework and the operating system to create a sandbox that is ultimately less flexible and extensible than standard Java's. FireDroid improved flexibility by adding a policy-enforcement mechanism for system calls (Russello et al., 2013). Aurasium (Xu et al., 2012) and AppGuard (Backes et al., 2014) transform Android applications to include a sandbox reminiscent of standard Java's, and share our goal of requiring no system changes. These solutions are at the granularity of an application, but Compac sandboxes third-party application components (Wang et al., 2014). XiOS brings a similar style of fine-grained access

control to Apple's iOS, again without requiring invasive system changes (Bucicoiu et al., 2015).

4.8 Conclusion

We discussed the design and implementation of MAJIC, a tool that assists developers in applying the Java sandbox to applications in whole or in part. Our tool contributes the ability to quickly and accurately perform two operations that were previously tedious, manual, and error prone: recovering policies from arbitrary Java code and sandboxing subsets of Java applications with different policies. Where MAJIC cannot automate the sandboxing process it helps developers make correct decisions or checks their work. Applications sandboxed with MAJIC run 6.9% slower overall due to use of the sandbox. In one nearly worst-case experiment we measured 6.6% overhead at borders between sandboxed and unsandboxed code on unmodified JVMs.

We suggest the following guidelines to reduce the burdens involved in sandboxing application subsets:

- Consider desired sandbox boundaries from the beginning of an application's design.
- Limit the use of reflection to cases that do not cross sandbox boundaries.
- Group classes in packages and JARs based on the protection domains they should appear in.
- Limit coupling between classes that should appear in different domains, particularly the use of inheritance.

However, compliance will be difficult due to the large number of existing frameworks and libraries developed without knowledge of these guidelines. Furthermore, while these guidelines make tooling easier to use, they do not necessarily simplify manual sandboxing. Even if the guidelines are followed ideally and manual sandboxing is possible, MAJIC is still required to help recover and refine a policy.

Changing the JDK, particularly the policy language, to allow code sources to specify individual classes could reduce or eliminate the burdens of partitioning classes, minimizing coupling, and dealing with class loader conflicts. However, these changes are onerous enough politically and technologically that we chose to face complexity in our transformations rather than incur the costs of overcoming existing inertia.

In this and the previous chapter we eliminated some unnecessary complexity in the Java sandbox and overcame necessary complexity through the use of tooling. Many sandboxes contain complexity points that lead to vulnerabilities. In the next chapter we once again broaden our scope to explore sandbox failures (a side-effect of complexity) and contain those failures using an architecture that leverages cloud computing environments.

Chapter 5

In-Nimbo Sandboxing[1][2]

The sandboxes discussed to this point have all been *in-situ*, existing within the system being defended. We saw in the case of Java that when the sandbox fails the host machine is compromised. In this chapter we explore similar failures in other sandboxes and devise an architecture for containing unwanted outcomes resulting from the failures. While the architecture generalizes, we use sandboxes for rich file formats to focus our efforts and evaluation.

Rich file formats such as Adobe's PDF and Microsoft's DOCX and PPT illustrate the need for sandboxes. PDF is essentially the Internet's paper. PDFs are used to present scientific literature, engineering and software datasheets, operating manuals for most products, legal and tax documents, and essentially anything else where layout and presentation preservation are important. Similar cases can be made for Microsoft's Word and PowerPoint formats. However, none of the listed formats represent static passive content. PDF in particular allows the embedding of fully capable programming languages, movies, audio, 3D models, and interactive forms. The viewers for these formats have been written in weakly typed languages (primarily C/C++) for decades to accommodate such rich behavior without compromising performance. This results in applications that are intractable to verify for security attributes. Combining rich feature sets with unverifiable code has led to viewers rife with vulnerabilities.

Given how precarious this situation is, it is little surprise that malicious actors have and continue to leverage these formats to compromise unsuspecting users. The most common attack sequence, known as spear phishing, involves an actor sending a carefully crafted email to a targeted recipient with a subtly corrupted file attached. For example, a government official may be emailed a reminder about an upcoming and relevant conference with a flyer attached with more information. When opened, the flyer exploits the viewer and compromises the recipient's machine. Vendors have responded by leveraging the power of sandboxes. Adobe sandboxed the parser and renderer for PDF using a version of Google Chrome's sandbox and Microsoft introduced Protected Mode to achieve a similar effect for the Microsoft Office suite. However, these sandboxes have been successfully attacked.

[1] *Nimbo* is Latin for "cloud."

[2] This chapter was adapted from a paper written with help from William Scherlis and Jonathan Aldrich. It is published in "Symposium and Bootcamp on the Science of Security" 2014

In the case of PDF, one common suggestion from the security community is to deal with this issue by using alternative PDF viewers (Landesman, 2010; Patrik, 2009; Scherschel, 2012), which may not suffer from the same vulnerabilities. But in fact many of these share code, because Adobe licenses its PDF parser and renderer (Systems, 2012). The advice nonetheless has some merit because even if an alternative viewer contains the exact same vulnerability as Reader, an exploit PDF that targets Reader won't necessarily work "out of the box" on an alternative. Many organizations, however, have grown dependent on PDF's more exotic features, such as fly-through 3D models, forms that can be filled out and then digitally signed by a user using hardware tokens, or embedded multimedia in a wide range of formats. Alternative readers have historically not supported the full range of PDF features, and often still do not, because exotic features are often expensive to implement and may be used in small niches.

Let us consider, therefore, an example organization with needs that require use of Adobe Reader, which has a significant, complex attack surface to defend. To gauge the attack surface, we examined the vulnerability streams (essentially RSS feeds of vulnerability reports) provided by NIST's National Vulnerability Database. We only used streams that contained a full years worth of data as of 2013. We found relevant vulnerabilities by searching entries that contain a cpe-lang:fact-ref field with a name attribute containing the text adobe:reader, adobe:acrobat, or adobe:acrobat_reader. We found great diversity in where the vulnerabilities reside in the code, as shown in Figure 5.1. This distribution was determined by coding all entries by hand that were identified as belonging to Reader and where the coded Common Weakness Enumeration (CWE) explicitly or implicitly identified a component. Thus, the results likely understate the diversity of vulnerabilities in Reader. How can the organization that must use Adobe Reader deal with this extensive attack surface?

Others have attempted to detect malicious PDFs (Cert-IST, 2010; Dixon, 2012; Esparza, 2012; Fratantonio et al., 2011; Laskov and Srndic, 2011; Maiorca et al., 2012; Nedim Srndic and Pavel Laskov, 2013; Smutz and Stavrou, 2012; Tzermias et al., 2011) with modest success even in the best cases. A systematic analysis (Maiorca et al., 2013) substantiates the inadequacy of many techniques for detecting malicious PDFs. We believe that even if detection methods advance, they will never be adequate against advanced attackers. The sheer complexity involved in detecting every possible attack in a format as massive as PDF is prohibitive. Thus, even given the successful attacks against deployed sandboxes we maintain that using sandboxes to encapsulate risky components is a reasonable engineering decision in these circumstances.

Contribution. This chapter acts as an additional case study in the value of sandboxing and acknowledges that sandboxes are imperfect and sometimes fail. We present and evaluate an application-focused sandboxing technique that is intended to address both sides of the risk calculation – mitigating the consequences of traditional sandbox failures while also increasing the effort required by an attacker attempting to compromise the target system. Our technique is reminiscent of software as a service, thus allowing us to evaluate the security benefits of those and similar architectures. We present the technique, describe a prototype we developed to support a field trial deployment, and assess the technique according to a set of defined criteria. These steps act to validate the following hypotheses

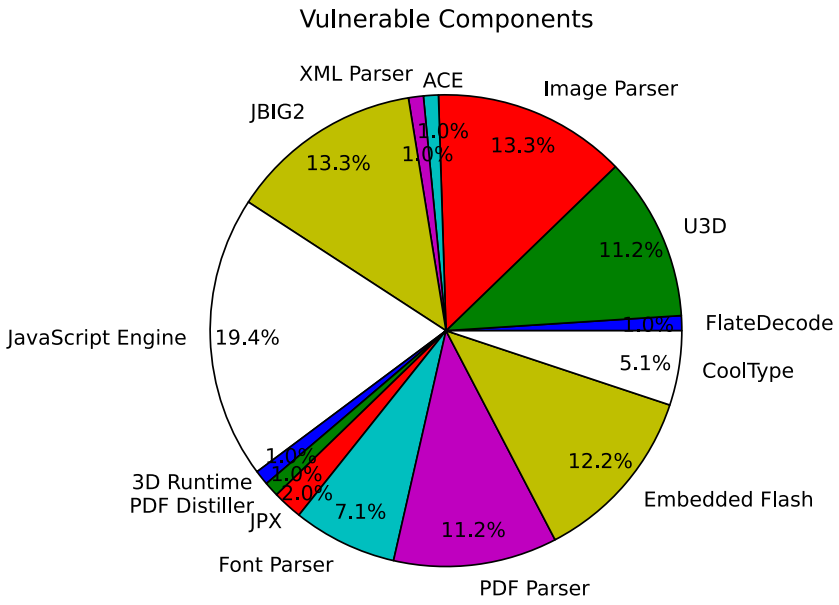


Figure 5.1: The distribution of vulnerabilities in Reader amongst identifiable components. This distribution is intended to show that problems in Reader are so diverse there is no clear place to concentrate defensive efforts other than sandboxing.

about the technique:

- Attack Surface Design Flexibility: In-nimbo sandboxing provides flexibility in attack surface design. We focus on tailoring the sandbox to the application, which doesn't allow for a "one size fits all" implementation. Our technique allows architects to more easily design and implement an attack surface they can confidently defend when compared to other techniques. This is because the technique is less constrained by structures within an existing client system.
- Attack Surface Extent: Our technique results in encapsulated components with smaller, more defensible attack surfaces compared to the cases where the component is encapsulated using other techniques. Along with the previous criterion, this should have the effect of diminishing the "likelihood" part of the risk product.
- Consequence of Attack Success: Remote encapsulation reduces the consequences of attack success. Our technique reduces the magnitude of the damage resulting from an attack on the encapsulated component when compared to the same attack on the component when it is encapsulated using other techniques. That is, we suggest the approach diminishes the extent of consequence in the risk product.

We also apply the following criteria:

- Performance: We focus on latency and ignore resource consumption. The technique slightly increases the user-perceived latency of an encapsulated component compared to the original version of the component. This is based on data from our field-trial deployment, and in this regard we do benefit from the fact that the encapsulated component is a large, complex, and relatively slow vendor application.
- Usability: We focus on changes to the user experience as well as ease of deployment. The technique does not substantially alter the user's qualitative experience with the encapsulated component after sandbox initialization. Deployment is straightforward, as described below.

As alluded to above, we evaluate these points based on data from a field trial with disinterested users at a large aerospace company. Performance measurements were taken from this deployment. We should note that we were not able to control all features of the deployment, and there was some architectural mismatch with respect to communications channels and cloud architecture. Additionally, our system is an operationalized prototype with some inconvenient design choices. The data we obtained nonetheless suggests that the approach offers real benefits – even in the presence of these limitations.

The next section briefly discusses the state of sandboxes in practice to illustrate the need for a new technique. Our technique is further motivated in Section 5.2. We discuss an in-nimbo sandbox prototype we built and deployed for Adobe Reader and its performance in section 5.3. Section 5.4 compares the in-nimbo sandbox with applying a local sandbox to Reader. We look at thought experiments for applying in-nimbo sandboxing in Section 5.5 before discussing possible extensions in Section 5.6.

5.1 Sandboxes in Practice

In practice, sandboxes tend to combine a number of distinct encapsulation techniques to achieve their goal. For example:

- chroot sandboxes (commonly referred to as chroot jails) redirect the root of the filesystem for a specific application (Friedl, 2012). This redirection has the effect of preventing the application from accessing files that are not below what it sees as the filesystem root.
- Google NaCl applies SFI and runtime isolation to prevent memory corruption exploits in native code and to constrain what the native code can do at runtime respectively (Yee et al., 2009).
- Microsoft Internet Explorer's Protected Mode works by constraining the execution of risky components, such as the HTML renderer, using rules encoded as integrity levels (MSDN, 2011).
- TxBox intercepts all of the system calls made by an application and ensures the calls do not violate a security policy (Jana et al., 2011).
- TRuE intercepts systems calls, employs SFI, and redirects system resources (Payer et al., 2012).

The combination of techniques applied by each sandbox varies both the types of policies that can be enforced by the sandbox (Clarkson and Schneider, 2010; Hamlen et al., 2006) and how usable the sandbox is for the architect applying it. A chroot jail cannot prevent the execution of arbitrary code via a buffer overflow, but chroot jails can make sensitive files unavailable to the arbitrary code and are quick and easy to deploy when compared to applying Google NaCl to a component. Of course, these two sandboxes are not intended to encapsulate the same types of applications (locally installed desktop applications versus remote web application components), but the comparison does illustrate that the techniques applied by a sandbox have an impact on both how widely a particular sandbox can be applied and how it can fail to protect the underlying system.

Software complexity adds an additional wrinkle. Consider Adobe Reader, which has a robust PDF parser, a PDF renderer, a JavaScript engine, a Digital Rights Management engine, and other complex components critical to the application's function. It may be extremely difficult to find a sandbox with the right combination of techniques to effectively encapsulate Reader without introducing significant complexity in applying the sandbox itself. Even when breaking up these components for sandboxing purposes, the architect must apply a sandbox where the combination of sandboxing techniques is powerful enough to mitigate the threats faced by the component while also applying the techniques in a manner that is acceptable given the unique characteristics and requirements of each sandboxed computation. Additionally, the sandboxed components must be composed in a way where the composition is still secure (Mantel, 2002; Sewell and Vitek, 1999). If the combination is not secure, it may be possible for an attacker to bypass the sandbox, e.g. by hopping to an unsandboxed component. The complexity may make the sandbox itself a target, creating an avenue for the attacker to compromise the sandbox directly.

5.2 In-Nimbo Sandboxing

This section motivates the need for in-nimbo sandboxing by looking closer at general weaknesses in traditional sandboxes and discusses characteristics of a more suitable environment for executing potentially vulnerable or malicious computations. We then discuss a general model for in-nimbo sandboxing that approximates our ideal environment.

5.2.1 Why In-Nimbo Sandboxing?

Most mainstream sandboxing techniques are in-situ, meaning they impose security policies using only Trusted Computing Bases (TCBs) within the system being defended. In-situ sandboxes are typically retrofitted onto existing software architectures (McQuarrie et al., 2010a,b,c; Office Support, 2012; Schuh, 2012; Stender, 2010; Uhley and Gwalani, 2012) and may be scoped to protect only certain components: those that are believed to be both high-risk and easily isolatable (Project, 2012; Sabanal and Yason, 2011). Existing in-situ sandboxing approaches decrease the risk that a vulnerability will be successfully exploited, because they force the attacker to chain multiple vulnerabilities together (Buchanan et al., 2012; Obes and Schuh, 2012) or bypass the sandbox. Unfortunately, in practice these techniques still leave a significant attack surface, leading to a number of attacks that succeed in defeating the sandbox. For example, a weakness in Adobe Reader X's sandbox has been leveraged to bypass Data Execution Prevention and Address Space Layout Randomization (ASLR) due to an oversight in the design of the sandbox (Delugre, 2012). Experience suggests that, while in-situ sandboxing techniques can increase the cost of a successful attack, this cost is likely to be accepted by attackers when economic incentives align in favor of perpetrating the attack.[3] The inherent limitation of in-situ techniques is that once the sandbox has been defeated, the attacker is also "in-situ" in the high-value target environment, where he can immediately proceed to achieve his goals.

In order to avoid the inherent limitations of in-situ sandboxing approaches, we propose that improved security may be obtained by isolating vulnerable or malicious computations to ephemeral computing environments away from the defended system. Our key insight is that if a vulnerable computation is compromised, the attacker is left in a low-value environment. To achieve his goals, he must still escape the environment, and must do so before the ephemeral environment disappears. The defender controls the means by which the attacker may escape, shaping the overall attack surface to make it more defensible, thereby significantly increasing the cost of attacks compared to in-situ approaches while simultaneously reducing the consequences of successful attacks.

We use the term ephemeral computing environment to refer to an ideal environment whose existence is short, isolated (i.e. low coupling with the defended environment), and non-persistent, thus making it fitting for executing even malicious computations. A number of environments may approach the ideal of an ephemeral computing environment, for example, Polaris starts Windows XP applications using an application-specific user account

[3]Google Chrome went unexploited at CanSecWest's Pwn2Own contest for three years. Then in 2012, Google put up bounties of $60,000, $40,000, and $20,000 in cash for successful exploits against Chrome. Chrome was successfully exploited three times (Goodin, 2012).

that cannot access most of the system's resources (Stiegler et al., 2006). Occasionally deleting the application's account would further limit the scope of a breach. Terra comes even closer by running application specific virtual machines on separate, tamper resistant hardware (Garfinkel et al., 2003b). Terra requires custom hardware, complicated virtual machine and attestation architectures, and doesn't outsource risk to third parties. In this chapter we focus on cloud environments. Cloud computing closely approximates ephemeral environments, as a virtual computing resource in the cloud is isolated from other resources through the combination of virtualization and the use of separate infrastructure for storage, processing, and communication. It may exist just long enough to perform a computation before all results are discarded at the cloud. We call this approach in-nimbo sandboxing.

Executing computations in the cloud gives defenders the ability to customize the computing environment in which the computation takes place, making it more difficult to attack. Since cloud environments are ephemeral, it also becomes more difficult for attackers to achieve persistence in their attacks. Even if persistence is achieved, the cloud computing environment will be minimized with only the data and programs necessary to carry out the required computation, and so will likely be of low value to the attacker. There may still be some value to the attacker in the compromised cloud machines, but this is now the cloud provider's problem, which he is paid to manage. This ability to outsource risk to the provider is a significant benefit of in-nimbo sandboxing from the point of view of the client. In order to escape to the high-value client environment, the attacker must compromise the channel between the client and the cloud. However, the defender has the flexibility to shape the channel's attack surface to make it more defensible.

To make the idea of in-nimbo sandboxing clear, consider Adobe Reader X. Delegating untrusted computations to the cloud is quite attractive for this application, as Reader in general has been a target for attackers over several years. As described more in Section 5.3, we have built and experimentally deployed in an industrial field trial an in-nimbo sandbox for Reader that sends PDFs opened by the user to a virtual machine running in a cloud. An agent on the virtual machine opens the PDF in Reader. Users interact with the instance of Reader that is displaying their PDF in the cloud via the Remote Desktop Protocol (RDP). When the user is done with the document, it is saved back to the user's machine and the virtual machine running in the cloud is torn down. This example illustrates how a defender can significantly reshape and minimize the attack surface.

5.2.2 General Model

There are good reasons to reduce the Trusted Computing Base (TCB) required to execute applications and entire operating systems (McCune et al., 2008a,b; McCune et al., 2010b; Singaravelu et al., 2006). The idea is to completely isolate unrelated computations from each other, and to use a TCB that is small enough to be verified, thus reducing and localizing the attack surface to a small, thoroughly vetted subset of the system's code.

In-nimbo sandboxing addresses this challenge by providing designers, even when working in legacy environments, with the flexibility to design TCB(s) to suit their specific context, thus channeling the attacker to a designer-chosen attack surface. This is significantly more flexible as it allows designers to achieve the benefits of a minimal TCB in

current commodity hardware/software systems, largely unconstrained by the particular engineering decisions of those systems. When applying in-situ sandboxes, an architect is limited to applying the sandboxing techniques that are supported by the instruction set, operating system, application type, etc., of the system she is trying to defend. These challenges can be particularly difficult to address when vendor software must be sandboxed. However, in-nimbo sandboxes can limit the majority of the attack surface to the communication channel between the client and the cloud. The designer can design a communication channel they are adequately prepared to defend.

In general, in-nimbo sandboxes contain the following:

- A specialized ***transduction mechanism*** in the computing environment we are trying to protect (the principal computing environment) that intercepts invocations of untrusted computations and transfers them to the high value TCB Architecture (see below) on the same system. The transduction mechanism also receives results from the high value TCB architecture and manages their integration into the rest of the system.
- A ***high value TCB architecture*** that sends the untrusted computation and any necessary state to an ephemeral computing asset, separate from the principal computing asset. The high value ***TCB architecture*** receives the results of the computation and state from the cloud, verifies both, and transfers them back to the transduction mechanism. We use the term TCB architecture to reflect the fact that our TCB(s) may be nested in or otherwise cooperate with another TCB (e.g., another sandbox). The nested TCBs can thus compensate for each other's faults and oversights and add redundancy. An attacker must counter each TCB in the architecture to compromise the overall system. In the case of the high value TCB architecture, this could allow an attacker to compromise the system we are trying to defend.
- The cloud executes untrusted computations in a ***low value TCB architecture*** and sends results and state back to the high value TCB architecture.

The components and their data flows are depicted in Figure 5.2. By picking these components and the contents of the data flows, the defenders effectively channel an antagonist to an attack surface the defenders are confident they can protect. An attacker must bypass or break every TCB in both TCB architectures or bypass the transduction mechanism to successfully compromise the high value environment.

5.2.3 Complementary Prior Work

Malkhi *et al.* developed a technique in 1998 for executing Java applets that is remarkably similar to in-nimbo sandboxing in both intent and high level architecture (Malkhi et al., 1998) (we were not aware of this work until well after publishing our original paper on in-nimbo sandboxing). When a user's browser requests an applet, the applet is instead transferred to a remote dedicated machine for execution. The downloaded code is transformed before execution to use the initiating browser as a graphical terminal for interacting with the applet as if it is running locally. A web proxy is used as the transduction mechanism and an applet that acts as the viewer in the user's browser is the high value TCB.

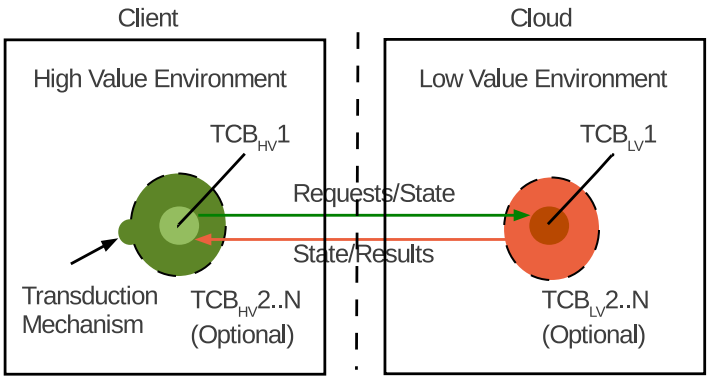


Figure 5.2: A general model of an in-nimbo sandboxing showing the transduction mechanism and the TCB architecture model. The circles with dashed borders represent one or more TCBs that contain the primary TCBs in each environment. The primary TCB in the high value environment ($TCB_{HV}1$) is responsible for sending requests and state to the low value environment's primary TCB ($TCB_{LV}1$). $TCB_{LV}1$ performs any necessary computations and returns state and results to the $TCB_{HV}1$, which must verify the results. The transduction mechanism moves computations and results into and out of the high value TCB architecture respectively.

This viewer uses a novel combination of Remote Method Invocation (RMI) and Java Abstract Window Toolkit (AWT) stubs to allow for remote interaction. As a result, Malkhi's sandbox is tied to the Java Virtual Machine, and cannot be used to sandbox non-Java technologies. In-nimbo sandboxing generalizes these basic principles to other technologies and modernizes their implementation. While Malkhi makes many of the same arguments for remotely sandboxing untrusted code that we do, we use a structured evaluation that goes well beyond the *ad hoc* arguments Malkhi presents.

Martignoni *et al.* have applied cloud computing to sandbox computations within the cloud in an approach that is complementary to ours (Martignoni et al., 2012). Their trust model reverses ours: whereas our model uses a public, low-trust cloud to carry out risky computations on behalf of a trusted client, they use a private, trusted cloud to carry out sensitive computations that the client is is not trusted to perform. They utilize Trusted Platform Modules to attest to the cloud that their client-end terminal is unmodified. They must also isolate the terminal from any malicious computations on the client. Our technique assumes security precautions on the side of the public cloud provider—an assumption we feel is realistic, as cloud providers already have a need to assume this risk.

The scenarios supported by the two techniques are complementary, allowing the application of the two techniques to different components of the same system. For example, Martignoni's sandbox may be used for performing particularly sensitive operations such as online banking, while our technique is useful in the same system for executing untrusted computations from the Internet. These choices reflect the varying trust relationships that

are often present in software systems.

Nakamoto *et al.* published a technique they call Desktop Demilitarized Zones (Nakamoto et al., 2011) that was developed in parallel with the project discussed in this chapter. They use virtual machines running in network demilitarized zones (DMZ) to sandbox applications that interact with non-enterprise assets. For example, if a user clicks a link to an external web page, they are silently redirected to a browser running in a virtual machine in the DMZ instead of using their local browser. The machines in the DMZ are refreshed hourly to ensure compromised machines do not stay compromised for long. Due to their presence in the DMZ, these machines have access to the Internet but not directly to the internal network. Users interact with the sandboxed application via the RDP. There is little security analysis provided for their specific approach. The technical details suggest the approach is superficially similar to in-nimbo sandboxing, but in-nimbo sandboxing further specializes machines to enable more confident security analysis. In our correspondence with the authors of this paper, they pointed out that their technique can be observed in a commercial product now sold by Invincea,[4] although this and other similar products were not developed by the authors.

5.3 Case Study

In-nimbo sandboxing can be applied to sandboxing entire applications or just selected components/computations. In this section we discuss the design of an in-nimbo sandbox for Adobe Reader that we prototyped and experimentally deployed at a large aerospace company. We then discuss the basis for comparing the sandbox with an in-situ sandbox for Adobe Reader. In the last section, we discuss other uses for in-nimbo sandboxes.

5.3.1 Design

To demonstrate the ability of an in-nimbo sandbox to support rich features, we set out with the goal of designing and building a sandbox for Reader that can support a user clicking links in a PDF, filling out and saving forms, interacting with multiple PDFs in the same session, printing PDFs, copying and pasting data to and from a PDF, and interacting with an embedded 3D model. We neglect features such as signing PDF documents with a smart card and several other non-trivial features Adobe advertises (though these features are likely supported via the RDP implementation we used). As an additional convenience, we decided that any Reader settings changed by the user should persist across their sandbox sessions. These design choices ensure the user's interactions with in-nimbo Reader are substantially similar to their typical interactions with in-situ Reader. In fact, aside from a few missing but less commonly used features, the user's interaction only differs in the appearance of the window chroming.

Figure 5.3 shows a high-level structural model of our prototype in-nimbo sandbox for Adobe Reader. The transduction mechanism consists of a file association between the PDF file extension and nimbo_client.py. This small script and its infrastructure is the primary

[4]https://www.invincea.com/

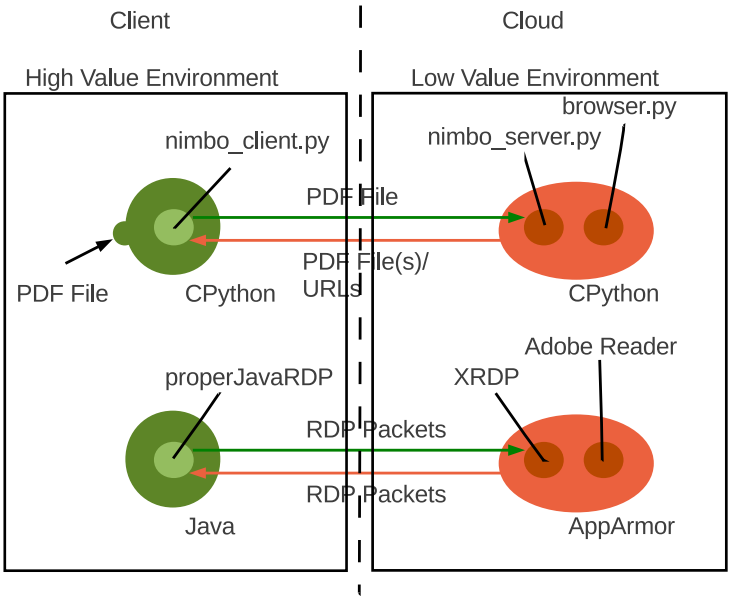


Figure 5.3: An in-nimbo sandbox for Adobe Reader.

TCB in the high value environment (24 lines of Python + Java RDP). When a PDF file is opened nimbo_client.py transfers the file to nimbo_server.py running in a cloud virtual machine instance. If a sandboxed session does not yet exist, a new session is created. Otherwise, the new PDF is opened as a new tab in the already open instance of Reader.

The user interacts with PDFs opened in the sandboxed version of Reader over an encrypted RDP connection. When Reader, the RDP session, or the RDP client is closed, all of the PDFs in the sandbox are sent back to nimbo_client.py. The PDFs must be returned to the high value client environment because the user may have performed an operation in the sandbox that changed the PDF, such as filling out and saving a form. After the PDFs are returned, nimbo_server.py restores the home directory of the in-nimbo user account that runs Reader to its original state. The cloud environment can always start virtual machines from a clean template, but alternatively resetting the home directory can enable a virtual machine to be re-used, e.g., due to high congestion. The account that runs Reader has limited access to the rest of the system.

When a user clicks a web link in a PDF, that link is sent to the nimbo_client.py and opened on the workstation. If the user does not want links to be opened on their workstation due to the risk of loading a potentially malicious site, they could instead have links opened in an in-nimbo version of their favorite browser. In this way, in-nimbo sandboxes can be composed. Sandbox composition is useful in this case because it prevents any one sandbox from becoming too complicated, and, in particular, having an overly rich attack surface.

5.3.2 Performance

Our prototype sandbox's performance evaluation is limited by two factors: transfer rates and inefficiencies in the cloud virtual machine set-up for the field trial. But even with inefficiencies in the virtual machine set-up, user perception of performance is comparable with Reader's performance locally. Transfer rates dominate the time between when a user opens a PDF and when the user can interact with the PDF, but this rate is typically limited by the connection's upload rate at the workstation. As a result, the upload time statistics presented in this section are not intrinsic to in-nimbo sandboxing and may vary with different connections. The statistics were gathered in our field trial using an in-nimbo sandbox that was deployed as the company would typically deploy cloud applications. The measurements as a whole provide evidence that the approach performs acceptably for typical users.

The Python scripts use the same efficient file transfer schemes used by FTP clients. The server prototype in the cloud implements several optimizations, including queuing virtual machines with Reader already running and waiting for a user to connect. However, the infrastructure software running on the cloud virtual machines is not optimized for this use case. For example, establishing a connection with RDP would be faster if the connection was initialized ahead of time (i.e. subsequent connections via RDP to the virtual machine are much faster than the first connection). This is a side-effect of our choice to use Linux, X server, and xrdp. The issue does not exist on Windows with the RDP server Microsoft provides. It is also possible that xrdp can be modified to remove the issue. Table 5.1 summarizes results from our industrial collaborator who used an internal cloud and a Microsoft Windows client.

We measured performance with our collaborator by opening a 1 megabyte (MB) PDF ten times. We decided to use a 1 MB PDF after inspecting a corpus of about 200 PDFs characteristic of the use cases of the users and averaging the sizes of its contents. The corpus was collected by an engineer over three years and included the types of PDFs an average engineer encounters throughout their day: brochures, technical reports, manuals, etc. Most PDFs in the corpus were on the order of a few hundred kilobytes, but a small number of the PDFs were tens of MB in size.

For our measurements the high-value environment was 1,800 miles away from the datacenter hosting our cloud. While we parallelize many of the operations required to get to the point where the user can interact with the PDF, the largest unit of work that cannot be comfortably parallelized is transferring the PDF itself. In our tests, the user could interact with our 1 MB PDF within 2.1 seconds, which is comparable to the 1.5 seconds it takes to interact with a PDF run in Reader locally instead of in-nimbo. The Reader start-up difference is due to the fact that the virtual machine is much lighter than the workstation. The virtual machine doesn't need to run anti-virus, firewalls, intrusion prevention systems, productivity software, and other applications that slow down the workstation.

Though our sandbox increases the startup time, sometimes by several seconds in the case of large PDFs due to the transfer time, we observed no performance issues on standard broadband connections in the United States when interacting with the PDF. The sandbox also performed well when running malicious PDFs that were collected when they were used

PDF Size	1 MB
Average upload time	2.1 +/- 0.3 seconds*
Average Adobe Reader start time in-nimbo	0.5 seconds
Average time to establish RDP channel	1.5 seconds
Average time until user can interact	2.1 seconds
Distance from client to cloud	1,800 miles
Average Adobe Reader start time in-situ	1.5 seconds

Table 5.1: Performance metrics for an in-nimbo sandbox using a cloud internal to an enterprise and a Microsoft Windows client. Figures based on 10 runs. The upload time is the primary bottleneck.
*Confidence level: 95%

in attempted attacks targeted at our collaborator's employees. The malware did not escape the sandbox, nor did it persist across sessions. These results suggest that this technique is currently best applied to longer running, interactive computations unless the running time for the entire initialization process is negligible. The aerospace company initially rejected a proposal to transition the sandbox into production for day-to-day use by high-value targets within the company (e.g. senior executives). However, this decision was made due to a lack of resources and now, three years later, they are once again considering a production deployment.

5.3.3 Limitations

The sandbox prototype does not support the most recent features of PDF because we used the Linux version of Reader, which is stuck at version 9. This limitation is an accidental consequence of expedient implementation choices and is not intrinsic to in-nimbo sandboxing. It is possible, for example, to instead run Windows with the latest version of Reader in the cloud virtual machine, but this set-up would not substantially influence the performance results given the dominance of the transfer rate. (Furthermore, it is possible to run newer Windows versions of Reader in Linux. Adobe Reader X currently has a Gold rating in the WINE AppDB for the latest version of WINE (HQ, 2014). The AppDB claims that a Gold rating means the application works flawlessly after applying special, application specific configuration to WINE.)

Our malware test of the sandbox is limited by the fact that we didn't have access to malicious PDFs directly targeting Linux or malware that would attempt to break out of our sandbox.

5.4 In-Nimbo Adobe Reader vs. In-Situ Adobe Reader X

In this section we make a structured comparison between our in-nimbo sandbox for Reader with an in-situ Adobe Reader. First, we summarize the framework used for the comparison, and then we apply the framework. The purpose of the framework is to support a systematic exploration of our hypothesis that in-nimbo sandboxing leads to attack surfaces that (1) are smaller and more defensible and (2) offer reduced consequences when successful attacks do occur. The framework is necessarily multi-factorial and qualitative because quantification of attack surfaces and the potential extent of consequences remains elusive.

5.4.1 Structuring the Comparison

To compare sandboxes we consider what happens when the sandbox holds, is bypassed, or fails. A sandbox is *bypassed* when an attacker can accomplish his goals by jumping from a sandboxed component to an unsandboxed component. A sandbox *fails* when an attacker can accomplish his goals from the encapsulated component by directly attacking the sandbox. The key distinction between a sandbox bypass and a failure is that any malicious actions in the case of a bypass occur in a component that may have never been constrained to prevent any resulting damage. In a failure scenario, the malicious actions appear to originate from the sandbox itself or the encapsulated component, which creates more detectable noise than the case of a bypass. A bypass can occur when an insufficient security policy is imposed, but a failure requires a software vulnerability. These dimensions help us reason about the consequences of a sandbox break, thus allowing us to argue where in the *consequences* spectrum a particular sandbox falls within a standard risk matrix. To place the sandbox in a risk matrix's *likelihood* spectrum (i.e. the probability of a successful attack given the sandbox), we consider how "verifiable" the sandbox is. Our risk matrix only contains categories (e.g. low, medium, or high) that are meaningful to the comparison at hand. Finally, we rank the outcomes that were enumerated within the argument by their potential hazards, which helps highlight the difference between risk categories.

5.4.2 Comparing In-Nimbo Adobe Reader to In-Situ Adobe Reader

Adobe Reader X's sandbox applies features of Microsoft Windows to limit the damage that an exploit can do. The Reader application is separated into a low privilege (sandboxed) principal responsible for parsing and rendering PDFs and a user principal responsible for implementing higher privilege services such as writing to the file system. The sandboxed principal is constrained using limited security identifiers, restricted job objects, and a low integrity level. The higher privilege principal is a more stringently vetted proxy to privileged operations. The sandboxed principal can interact with the user principal over a shared-memory channel. The user principal enforces a whitelist-based security policy on any interactions from the sandboxed principal that the system administrator can enhance. Under ideal circumstances the sandboxed principal is still capable of reading secured objects

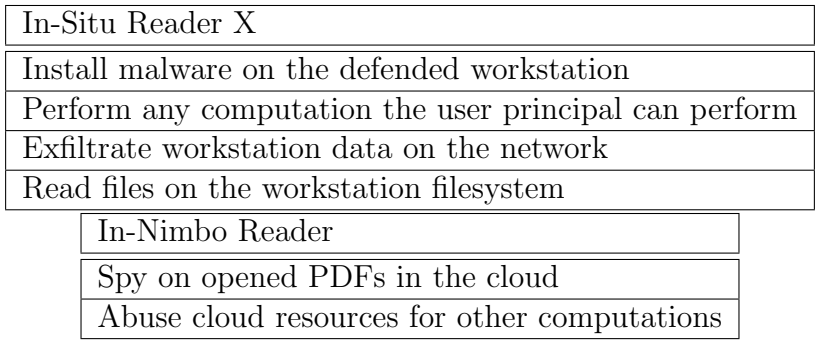


Figure 5.4: Likely sandbox "consequence outcomes" ranked from most damaging at the top to least damaging at the bottom. Each sandbox's outcomes are independent of the other sandbox's.

(e.g., files and registry entries),[5] accessing the network, and reading and writing to the clipboard without help from the user principle.

The consequences of an attack on Reader, even when the sandbox holds, are high. In its default state, a PDF-based exploit could still exfiltrate targeted files over the network without any resistance from the sandbox. If the sandbox is successfully bypassed, the attacker can leverage information leakages to also bypass mitigations such as ASLR as mentioned in Section 5.2.1. Such bypasses, which are publicly documented, are likely to be serious enough to allow a successful attack to install malware on the targeted machine. If other bypass techniques exist, they could allow an attacker to perform any computations the user principal can perform. These outcomes are ranked from most to least damaging in Figure 5.4. Overall, the consequences generally fall into one of the following categories:

- The integrity of the PDF and viewer are compromised
- The confidentiality of the PDF is compromised
- The availability of reader is compromised
- The security (confidentiality, integrity, availability) of the cloud infrastructure is compromised
- The security of the high value environment is compromised

The Reader sandbox is moderately verifiable. It is written in tens of thousands of lines of C that are heavily based on the open-source sandbox created for Google Chrome. The design and source code were manually evaluated by experts from several independent organizations who also implemented a testing regime. According to Adobe, source code analysis increased their confidence in the sandbox as its code was written. The operating system features on which it depends are complex; however, they were implemented by a large software vendor that is known to make use of an extensive Secure Development Lifecycle for developing software (Michael Howard and Steve Lipner, 2006).

[5]Windows Integrity Levels can prevent read up, which stops a process from reading objects with a higher integrity level than the process. Adobe did not exercise this capability when sandboxing Adobe Reader X.

We split the analysis as it pertains to the in-nimbo case based on the attacker's goal. In the first case, the attacker is either not aware of the in-nimbo sandbox or for other reasons does not change their goal from the in-situ case (i.e. they still aim to successfully attacker just Reader). In the second case, we consider an attacker that is aware of the in-nimbo sandbox and wants to compromise the workstation. When the attacker's goal does not change the consequences of a successful attack are lower in the in-nimbo case. When the attacker wants to compromise the workstation, the consequences of a successful attack are the same as in the in-situ case, but the odds of success (likelihood component) are lower.

Attacker Goal: Compromise Reader

Figure 5.5 summarizes our qualitatively defined risk for Reader X's sandbox against Reader running in our in-nimbo sandbox when the attacker's goals have not changed. The in-nimbo sandbox has a lower consequence (from the standpoint of the user) because exploits that are successful against Reader may only abuse the cloud instance Reader runs in. The operator of the cloud instance may re-image the instance and take action to prevent further abuse. However, to abuse the cloud instance the attacker would have to both successfully exploit Reader and bypass or break additional sandboxing techniques we apply in the cloud. The exploit must either not crash Reader, or its payload must survive the filesystem restoration and account log-off that would occur if Reader crashed due to the exploit (see 5.3 for details). We consider this analysis to be reasonable evidence to support our hypothesis that the in-nimbo sandbox can lead to reduced consequences in the event of successful attacks.

Attacker Goal: Compromise the Workstation

Assuming the attacker is aware of the in-nimbo sandbox and wants to compromise the workstation, the consequences of attack success remain the same as the in-situ case. The in-nimbo case is still better off, however, because the likelihood of success is diminished due to the smaller and flexible attack surface presented by the in-nimbo sandbox.

The attacker could potentially *bypass* the sandbox by tricking our mechanism into opening the PDF in a different locally installed application capable of rendering PDFs. For example, the attacker may disguise the PDF file as an HTML file, causing it to be opened in the browser if the transduction mechanism is only based on file associations. The browser might have an add-on installed that inspects the document, determines it is a PDF regardless of extension, and then renders the PDF. While this attack would eliminate the benefit of the sandbox, it is not likely to be successful if the user verifies there is not a local PDF viewer installed/enabled (an important part of configuration). The transduction mechanism can also simply use a richer means of determining whether or not a file is a PDF.

The sandbox could *fail* in a way that compromises the user by either an exploitable vulnerability in our 273 line Python-based TCB (and its infrastructure), the Java RDP client we use, or a kernel mode vulnerability exploitable from either. Such a failure would require first successfully compromising the cloud instance as discussed earlier and then

finding an interesting vulnerability in our small, typesafe components. In other words, a failure of the sandbox requires that the TCBs in both the client and the cloud fail.

In short, the in-nimbo sandbox is easier to verify and requires more work for an attacker to achieve enough access to compromise the client. Adobe Reader X's sandbox is harder to verify and allows the workstation it is running on to be compromised even if the sandbox holds. Due to the well known characteristics of each sandbox, we consider this evaluation to be reasonable evidence of the validity of our hypotheses that in-nimbo sandboxing leads to smaller, more defensible attack surfaces. While we only evaluated one sandbox in addition to the in-nimbo sandbox and for only one application, the results of the evaluation are largely influenced by issues that are fundamental to in-situ sandboxing when compared to in-nimbo sandboxing.

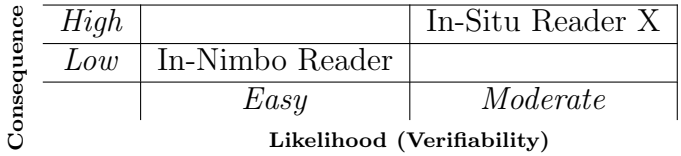


Figure 5.5: A grid summarizing our evaluation of Reader X and our In-Nimbo sandbox for Reader.

Concluding the Comparison

A final potential point of concern is that the cloud instance's hypervisor could be compromised, thus compromising other virtual machines managed by that hypervisor or even the entire cloud. We do not consider this issue in our analysis because the sandbox is a use of the cloud, not an implementation of a cloud. One of the key selling points behind using a cloud environment is that the provider manages the infrastructure; they take on the risk and management expenses. The ability to outsource risk that cannot be eliminated to a party that is willing to assume the risk is a key advantage of this approach. Additionally, the technique does not add a new threat to clouds in the sense that anyone can rent access to any public cloud for a small sum of money and attempt to compromise the hypervisor. Finally, we are primarily sandboxing computations because we don't trust them. In the case of the Reader sandbox, a compromise could cause sensitive PDFs to be stolen, which would still be better than the compromise of an entire workstation full of sensitive information.

In this section, we have used a structured approach to compare in-nimbo Reader to the standard version. Overall, the comparison is in the in-nimbo sandbox's favor, but for different reasons depending on the attacker's goals. If the attacker treats in-nimbo Reader the same way as the traditional install of Reader, the consequences of a successful attack are lower because the attacker has only compromised a customized, temporary, and low-value environment. If the attacker is aware of the in-nimbo sandbox and wants to compromise the workstation, the consequences are the same as in the in-situ case but the likelihood of success if lower because the full in-nimbo sandbox must be successfully attacked as well as

Reader. The in-nimbo sandbox has a smaller attack surface that was intentionally designed to increase our confidence in our ability to repel attacks.

5.5 In-Nimbo Thought Experiments

In this section we consider the potential of applying the in-nimbo sandboxing technique for a subset of HTML5 and for protecting proprietary data. There are other examples (not elaborated here) that would be similar to our sandbox for Reader, such as an in-nimbo sandbox for Microsoft Word or Outlook.

5.5.1 Selective Sandboxing

HTML5 specifies a canvas element (W3C, 2012) that provides scripts with a raster-based surface for dynamically generating graphics. By default, browsers must serialize the image to a PNG for presentation to the user (other raster formats may be specified). For an animated surface, the serialized image represents the currently visible frame. Unfortunately, early implementations of such rich browser features have continually been prone to vulnerabilities. CVE-2010-3019 documents a heap-based buffer overflow in Opera 10's implementation of HTML5 canvas transformations (NIST, 2010b). In 2013 a stack-based overflow was discovered in Mozilla Firefox's HTML5 canvas implementation(NIST, 2013b). As a result, a risk-focused user may desire that HTML5 canvas computations be sandboxed.

Figure 5.6 shows the model of an HTML5 canvas in-nimbo sandbox. In this case, the transduction mechanism is a proxy running between a browser and the Internet. The transduction mechanism intercepts and inspects HTML pages for canvas declarations. When a canvas declaration is detected, the proxy collects all referenced JavaScript code and sends the page and scripts to the high value TCB architecture (the client). The client sends the collected HTML and JavaScript to the cloud instance, which utilizes an instrumented browser to render any drawing on the canvas. The canvas is replaced with the image file the browser generates (per the HTML5 standard) when the rendering script finishes. When a loop in the rendering script is detected (i.e. an animation is present), the canvas declaration is replaced with an image tag pointing to a 3 second animated GIF composed of all of the frames the script rendered in that time period. All JavaScript that drew on the canvas is removed, and the cloud returns the re-written JavaScript, HTML, and PNG to the client. The client verifies the image file and checks that no unexpected code/markup changes have been made before sending the results back to the proxy. The proxy passes the modified results to the browser.

This sandbox would effectively confine exploits on HTML5 canvas implementations to a low value computing environment. Furthermore, it would reduce the verification problem from verifying the full HTML5 canvas implementation and its dependencies to that of verifying raster image formats supported by the canvas tag and ensuring that no code has been added to the intercepted files (i.e., code has only been removed and/or canvas tags have been replaced with image tags). While the sandbox does not support animated canvases longer than 3 seconds or whose visual representation is dependent on real-time

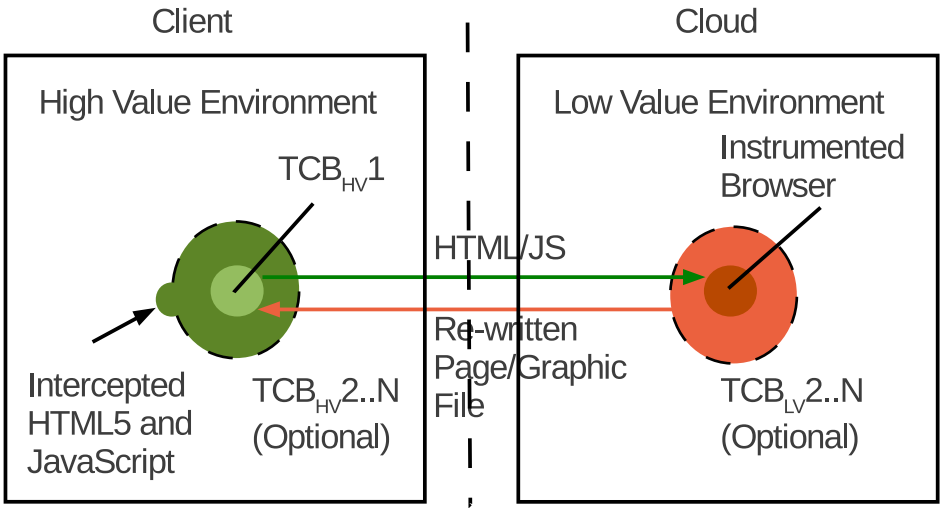


Figure 5.6: The model for an in-nimbo sandbox for HTML5 canvases.

user input, a user who cannot accept such limitations can use a browser that is fully sandboxed in-nimbo such as Reader was in the previous section. It is also possible that an alternative implementation of this sandbox could support longer animations and user input.

5.5.2 Protecting Proprietary Algorithms

Modern audio players allow users to manage libraries of tens of thousands of songs and automatically perform experience-enhancing operations such as fetching album covers from the Internet. More advanced players also attempt to identify songs and automatically add or fix any missing or corrupt metadata, a process known as auto-tagging. However, the act of robustly parsing tags to identify an album and artist to fetch a cover is potentially error prone and complicated. CVE-2011-2949 and CVE-2010-2937 document samples of ID3 parsing vulnerabilities in two popular media players (NIST, 2010a, 2011). Furthermore, an audio player vendor may consider all steps of the waveform analysis they use to identify untagged audio files to be proprietary. To address these concerns, the vendor may wish to design their application to make use of an in-nimbo sandbox to perform these operations.

Figure 5.7 shows the model of a possible in-nimbo sandbox for fetching album covers and performing auto-tagging. The transduction mechanism is the audio player itself. When the player detects an audio file that is new it sends the file to the high value TCB architecture (the client). The client sends the audio file to the cloud instance, which performs the task of automatically adding any missing tags to the audio file and fetching the correct album cover. The cloud sends the tagged audio file and album cover to the client, where it will be verified that only the audio files tags have changed, that they comply with the correct tagging standard, and that the album cover is of the expected format and well formed. The client will then send the verified audio file and album cover to the audio player, which

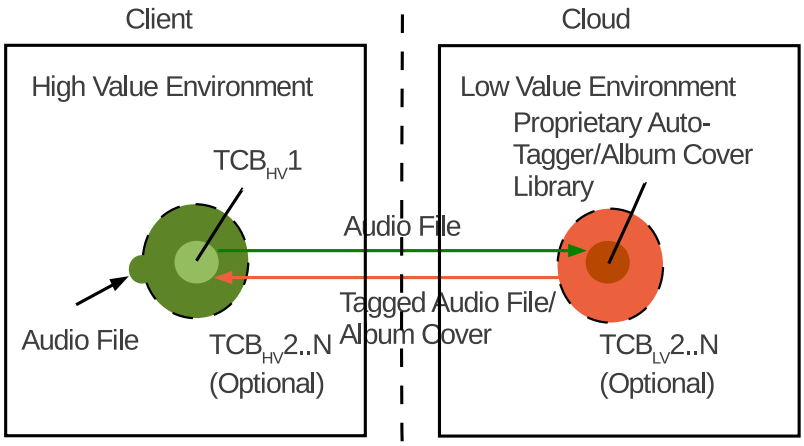


Figure 5.7: The model for an in-nimbo sandbox for an auto-tagging audio player.

will place both into their correct places in the library.

Assuming that all managed audio files make their way through the in-nimbo sandbox at least once, this sandbox effectively mitigates the risk of robustly parsing tags while also not exposing the inner-workings of the proprietary waveform algorithm. Potential limitations are curtailed by intentionally designing the application to utilize an in-nimbo sandbox.

5.6 Discussion

This chapter argued that we can improve system security (confidentiality, integrity, availability) by moving untrusted computations away from environments we want to defend. We first introduced one approach for achieving that idea, a category of sandboxing techniques we refer to as in-nimbo sandboxing. In-nimbo sandboxes leverage cloud computing environments to perform potentially vulnerable or malicious computations away from the environment that is being defended. Cloud computing environments have the benefit of being approximately ephemeral, thus malicious outcomes do not persist across sandboxing sessions. We believe this class of sandboxing techniques is valuable in a number of cases where classic, in-situ sandboxes do not yet adequately isolate a computation.

We argued that in-situ sandboxing does not adequately reduce risk for Adobe Reader, thus motivating us to build an in-nimbo sandbox for Reader. We then discussed the design of an in-nimbo sandbox for Reader and presented a structural argument based on five evaluation criteria that suggests that it is more secure and that, with respect to performance, has a user experience latency subjectively similar to that of Reader when run locally. After arguing that our sandbox is more secure than an in-situ sandbox for Reader, we illustrated how in-nimbo sandboxes might be built for other applications that represent different in-nimbo use cases.

Our argument for why this sandbox is better is structured but necessarily qualitative. We believe that many security dimensions cannot now be feasibly quantified. We nonethe-

less suggest that structured criteria-based reasoning building on familiar security-focused risk calculus can lead to solid conclusions. Indeed, we feel the approach is an important intermediate step towards the ideal of quantified and proof-based approaches.

Chapter 6

Future Work

Many of the previous chapters discuss relevant future work in context. This chapter discusses broader future work ideas that are not as closely connected to previously discussed topics. In particular, this chapter discusses policy usability in non-Java sandboxes, architectural constraints for sandboxing, and decision and engineering support for policy derivement and enforcement.

6.1 Enhancing Policy Usability

With few notable exceptions, the sandboxes easily observed in practice support either security policies the user cannot change or trivial policy languages for user-defined policies. Perhaps unsurprisingly, policy usability is a point of comparison between those sandboxes that do support rich user-defined policies. For example, AppArmor and SELinux are often compared in terms of the usability of their policy languages.[1] [2] Rich user-defined policies enable users to bound the behaviors an encapsulated component can exhibit at a fine granularity, while customizing the bound for the operational environment. However, these benefits are predicated on the user's ability to successfully write reasonable policies in the provided language. Given the problems resulting from complex security policy languages discussed in previous chapters, this space seems rife for additional improvement. Indeed, AppArmor is often cited as being easier to write policies for than SELinux, but even AppArmor users do not escape insecure policies.[3] We see two avenues for improving policy usability, which we will discuss in turn: (1) Demonstrate that the policy languages themselves are usable and (2) automate most, if not all, of the policy derivation process with quality guarantees for the produced policy.

The programming language community has made some progress on the first avenue. Programming language usability has recently been cited as a major concern and is emerging as a primary area of study (Ko et al., 2011; Mayer et al., 2012; Stylos and Myers, 2008). This

[1] `http://security.stackexchange.com/questions/29378/comparsion-between-apparmor-and-selinux`

[2] `https://www.suse.com/support/security/apparmor/features/selinux_comparison.html`

[3] `http://blog.azimuthsecurity.com/2012/09/poking-holes-in-apparmor-profiles.html`

is important because we often provide developers with tools that are confusing, incomplete, or buggy and expect the developers to use them to create software that is usable, complete, and free of bugs. The evidence in previous chapters suggests this same concern translates to sandboxes. The techniques being developed to evaluate the usability of fully-featured programming languages may also translate to policy languages, which are often relatively small domain specific languages (Albuquerque et al., 2015). However, in the case of policy languages we have one primary concern: Are the targeted users capable of writing secure policies for real-world components? However, we may also ask a different question: Is it reasonable to ask users to write security policies from scratch, or should we instead ask them to refine policies generated by tools?

This thesis has made progress on the automation question, which addresses the second avenue listed above. Chapter 4 discussed an approach for automatically deriving a starter policy for Java applications. This approach took advantage of choices made by the designers of the Java sandbox to create scalable program analyses. However, the general idea translates to essentially any sandbox that requires a user-defined security policy: Determine which of the program's operations have an effect on the external environment. In native programs, it is often the case that these operations are represented by system calls and that by statically and dynamically analyzing these calls, we can generate policies in a number of different policy languages. In fact, both AppArmor and SELinux ship with dynamic analysis tools to perform this type of operation, but they do not include static analysis tools capable of providing guarantees about the extent of the policy (i.e. the policy's coverage of the operations the program *could* perform). Static analysis will grow more appropriate for this type of application as binary analysis techniques continue to improve in terms of scalability and precision, thereby eliminating the need for source code (Brumley et al., 2011; Kinder and Veith, 2008; Song et al., 2008).

6.2 Architectural Constraints to Support Sandboxing

Chapter 1 discussed Firefox as a case study in how sandboxing can be hampered by high coupling. Chapter 4 concluded with advice for organizing Java applications to make them easier to sandbox. Program structure is also of concern in mainstream sandboxes like Microsoft Office's, Internet Explorer's, and Adobe Reader's (see Chapters 1 and 5) where the applications had to be re-architected to enable the sandboxing of parsers and renderers. Other sandboxes avoid this problem by simply sandboxing the entire application, which does not provide the level of granularity required to protect those applications from their own components.

Some research has been done to automatically partition applications based on privilege requirements (Akhawe et al., 2012; Bapat et al., 2007; Brumley and Song, 2004; Wu et al., 2013). However, of these approaches, only Akhawe *et al.*'s approach can be applied before the implementation phase of a development project. The National Research Council's report on software producibility for defense (NRC, 2010) notes, "Architecture represents the earliest and often most important design decisions–those that are the hardest to change and the most critical to get right." The examples above, Firefox in particular, provide

powerful support for this insight.

We need tools to help us treat sandboxes as an architectural primitive, thereby allowing us to carefully design the interconnections between our components to enable sandboxing. Furthermore, we need programming languages that allow us to enforce these architectural constraints by, perhaps, annotating a module's intended sandbox or intended security policy while implementing the module. This would enable program analysis to identify harmful coupling and potential policy violations that could compromise the developer's ability to securely encapsulate the various system components. It would also ensure the design intent is explicit from the standpoint of internal attack surfaces, discussed more below.

6.3 Decision and Engineering Support: Preparing the Battlefield with Sandboxes

The previous sections discussed fairly low-level open challenges that deal with the details of applying particular sandboxes. This section envisions sandboxes as tools for "preparing the battlefield." Historically, military leaders prepared battlefields by ordering the: digging of trenches, construction of fortifications, removal of cover the enemy can use, strategic positioning of troops, and other actions aimed at increasing the odds of success in a hostile engagement with an enemy force. In software security, we too can prepare the battlefield to our advantage, but we need help making the complex decisions required to adequately reduce risk with limited resources.

What components in a software system should be sandboxed? To begin to optimally answer this question, we need a reasonable understanding of the most likely points of attack and their likely weaknesses. Some work has been done in this space, for example, in measuring attack surfaces (Manadhata and Wing, 2011; Theisen, 2015; Younis et al., 2014), in synthesizing security device placement (Rahman and Al-Shaer, 2013), and in mapping out potential attack paths through networks (Lippmann and Ingols, 2005; Shandilya et al., 2014). However, these approaches are less useful in this particular context if they cannot measure large systems that have grown organically over decades, which leads to questions about their ability to scale in practice. Furthermore, they only consider the external attack surface and not internal attack surfaces.

Imagine we have architected an ideal network where the only possible paths of attack are known to us and carefully picked by us to limit attackers to just those surfaces we are ready to defend. These surfaces are inevitably software and, as we have seen throughout this thesis, software artifacts are rarely uniformly trusted. What is missing are tools and metrics to help us identify trust gradients within these applications and a means to cleanly separate them for sandboxing with targeted security policies. This would allow us to spend our resources on the least trusted subsets of software systems. However, the status quo often leaves developers with little option but to guess which subsets of their programs are the least trustworthy[4] and perhaps with few options to separate these subsets from the

[4]A more disciplined developer might ask questions similar to the following: Which components are

rest.

When we have defined a subset to target, how should we target it? What is the nature of the risk? What sandboxes are capable of encapsulating the subset in question? Do any of these sandboxes support a security policy capable of mitigating the threat? If we could confidently answer these questions, we may be able to apply sandboxes both effectively and with greater precision.

supplied by vendors with a poor security reputation? Which components are the most poorly analyzed or least analyzable?

Chapter 7

Conclusion

Researchers have spent the last few decades building and evaluating new sandboxing techniques and sandboxes. We view sandboxes as a security primitive for building secure and trustworthy software systems. Thus, while we believe it is important to continue to construct sandboxes, it is equally important to understand the sandboxing landscape in the bigger picture. This thesis steps back and presents the bigger picture, finding several areas for improvement and contributing some of the most critically needed advances.

In Chapter 2 we systematically analyzed ten years (2004-2014) of sandboxing literature from five top tier venues. We drew a number of conclusions:

- Sandbox design often requires trade-offs between: A. the amount of effort required from the user or B. the number of requirements imposed on the component(s) to be sandboxed. When the first option is favored, users are often required to re-architect systems or develop security policies that achieve least privilege almost entirely by hand. This decreases the usability of the sandbox and increases the odds of failure, but these sandboxes are often more broadly applicable while stopping a wider range of attacks. When the second option is favored, sandboxes become limited in the range of components they can encapsulate because they can only target those components that meet certain requirements (e.g. built in certain programming languages, built against certain libraries, containing specific metadata, etc.). However, sandboxes that favor component requirements often require little if any effort from the person applying the sandbox.
- Sandbox usability is often ignored entirely. While we found some literature that discusses sandbox usability in venues outside of our scope, the research at the venues where sandboxing papers are primarily published did not discuss the topic during the time span we studied. Sandbox usability is a serious consideration where developers or system administrators lacking security expertise play a role in applying a sandbox. Sandboxes that are too complex to understand or use correctly add an additional security layer an attacker must defeat, but can result in failures when the sandbox is used incorrectly.
- Sandbox validation has not improved in the time span studied. Researchers often make strong security claims about their sandboxes, but these claims are often eval-

uated using relatively subjective validation strategies that favor *ad hoc* arguments. This is concerning given our view of sandboxing as a security primitive because components we rely on heavily would almost certainly benefit from having claims backed up by objective science.

Given these findings, we narrowed our scope to the Java sandbox to determine whether or not usability and validation issues manifest in failures in practice. More specifically, we investigated design and implementation complexity that leads to vulnerabilities. These vulnerabilities are caused by the sandbox creators. We also investigated complexity leading to usability issues, which can lead to vulnerabilities caused by sandbox users. We differentiated between complexity that is unnecessary because it does not benefit users, which is complexity that should be eliminated, and necessary complexity that we cannot get rid of and should instead ameliorate.

In Chapter 3, we used a spate of exploits targeting Java from the first half of this decade to focus our efforts in search of unnecessary complexity in Java's sandbox. We started with the observation that Java exploits in this time period share a key operation: They turn the sandbox off at run time. Is it necessary to allow applications to turn the sandbox off at run time?

More broadly, what operations do exploits require that benign applications do not? We found through an empirical study of open source applications that there are distinct differences between how exploits and benign applications interact with the sandbox. We used these differences to define backwards compatible rules that, when enforced by run time monitors in the JVM, stop all known sandbox-escaping Java exploits. Essentially, these rules turn off unnecessary complexity in the JVM that allowed exploits to thrive.

While we were able to remove some unnecessary complexity (e.g. complexity caused by features that are not used by benign applications) in the JVM, we observed significant but likely necessary complexity that hampers benign use of the sandbox. Developers often misunderstand the security model and therefore make mistakes in their manual application of the sandbox that lead to vulnerabilities. In fact, out of 36 cases, we did not observe a single instance where the sandbox was used as intended (i.e. many real-world uses of the sandbox are not security related, and those that are security related use the sandbox in odd ways). We believe the low number of cases is due to the difficulty of using the sandbox caused by complexity we cannot remove. This motivated the development of tools to automate much of the sandboxing process, which we discussed in Chapter 4. These tools automatically derive a starter security policy from bytecode, help the user refine and double check the policy, and perform almost all of the work required to apply the sandbox to even subsets of Java applications. We used these tools to sandbox components in a number of real world applications that were previously difficult to sandbox securely by hand.

Having dealt with complexity in the Java sandbox, we turned to general sandbox complexity in Chapter 5. We used complex file formats and their sandboxed viewers as case studies in sandbox complexity. Many of these sandboxes are *in situ*, existing within the system to be defended. When these sandboxes fail, often due to implementation bugs or oversights on the part of users manually applying the sandbox, the system we want to defend is also compromised. We narrowed our scope to PDF and Adobe Reader to develop

and evaluate a sandbox that leverages cloud computing environments to run untrusted computations in approximately ephemeral computing environments. Essentially, the computations are run in a virtual machine away from the system we want to defend. State is carefully persisted outside of the virtual machine when the computation ends and the virtual machine is destroyed. While this approach provides a means to mitigate sandbox failures, it also allows us to customize the computing environment for the computation that is being executed. This customization allows us to design attack surfaces we are prepared to defend.

We hope this work inspires future efforts to carefully consider the desired attack surface instead of simply dealing with whatever attack surface results from the rest of the software development process. More broadly, we hope this thesis inspires additional work to improve sandboxes.

Appendix A

Sample MAJIC Report

MAJIC generates a report when transforming an application to capture the following points:

- Remaining tasks that must be manually carried out
- Interfaces that were generated for sandboxed classes
- High level transformations made to sandboxed classes
- Policy updates made for the new sandbox, including listings of new permissions and sandboxed class loaders
- The main methods updated to set the custom policy object
- Bytecode transformations made to unsandboxed and sandboxed classes
- Updates to JAR files (- means a file was deleted, + a file was added, and -/+ a file was updated)

A sample report begins on the next page.

Remaining Tasks

The application is not fully sandboxed until the remaining tasks are completed:

- Digital signatures were stripped from the following JARs because at least one class contained in each was updated; resign them if their integrity and authentication is important:
  - /home/mmaass/Downloads/CompanyDemo/AuthPlus_sandboxed/AuthPlus.jar
- Move majic.policy and MAJIC_sandboxed_classes.jar to your preferred location outside of the sandboxed application's classpath
- Update all execution scripts to add /home/mmaass/Downloads/CompanyDemo /AuthPlus_sandboxed/MAJIC_sandbox.jar to the classpath
- Set all execution scripts to run the application with: -Djava.security.manager -Djava.security.policy=[preferredLocation]/majic.policy -Dmajic.sandboxed.loc= [preferredLocation]
- (OPTIONAL) Update majic.policy to sandbox the application as a whole with a stricter policy

Generated Interfaces

The following interfaces were generated to replace the types for objects created from sandboxed classes:

- Created interface org.tinyradius.packet.MAJIC.interfaces.AccessRequest from org.tinyradius.packet.AccessRequest
- Created interface org.tinyradius.util.MAJIC.interfaces.RadiusException from org.tinyradius.util.RadiusException
- Created interface org.tinyradius.packet.MAJIC.interfaces.AccountingRequest from org.tinyradius.packet.AccountingRequest
- Created interface org.tinyradius.packet.MAJIC.interfaces.RadiusPacket from org.tinyradius.packet.RadiusPacket
- Created interface org.tinyradius.util.MAJIC.interfaces.RadiusEndpoint from org.tinyradius.util.RadiusEndpoint
- Created interface org.tinyradius.attribute.MAJIC.interfaces.IntegerAttribute from org.tinyradius.attribute.IntegerAttribute
- Created interface org.tinyradius.dictionary.MAJIC.interfaces.Dictionary from org.tinyradius.dictionary.Dictionary
- Created interface org.tinyradius.dictionary.MAJIC.interfaces.DictionaryParser from org.tinyradius.dictionary.DictionaryParser
- Created interface org.tinyradius.attribute.MAJIC.interfaces.VendorSpecificAttribute from org.tinyradius.attribute.VendorSpecificAttribute
- Created interface org.tinyradius.dictionary.MAJIC.interfaces.MemoryDictionary from org.tinyradius.dictionary.MemoryDictionary
- Created interface org.tinyradius.dictionary.MAJIC.interfaces.WritableDictionary from org.tinyradius.dictionary.WritableDictionary
- Created interface org.tinyradius.util.MAJIC.interfaces.RadiusUtil from org.tinyradius.util.RadiusUtil
- Created interface org.tinyradius.util.MAJIC.interfaces.RadiusClient from org.tinyradius.util.RadiusClient
- Created interface org.tinyradius.dictionary.MAJIC.interfaces.AttributeType from org.tinyradius.dictionary.AttributeType
- Created interface org.tinyradius.attribute.MAJIC.interfaces.RadiusAttribute from org.tinyradius.attribute.RadiusAttribute
- Created interface org.tinyradius.attribute.MAJIC.interfaces.StringAttribute from org.tinyradius.attribute.StringAttribute
- Created interface org.tinyradius.dictionary.MAJIC.interfaces.DefaultDictionary from

org.tinyradius.dictionary.DefaultDictionary

Sandboxed Classes

The following transformations were performed to generate classes that will run in the sandbox's protection domain:

- Renamed org.tinyradius.packet.AccessRequest to org.tinyradius.packet.MAJIC.sandboxed.AccessRequest and added org.tinyradius.packet.MAJIC.interfaces.AccessRequest as an interface
- Renamed org.tinyradius.util.RadiusException to org.tinyradius.util.MAJIC.sandboxed.RadiusException and added org.tinyradius.util.MAJIC.interfaces.RadiusException as an interface
- Renamed org.tinyradius.packet.AccountingRequest to org.tinyradius.packet.MAJIC.sandboxed.AccountingRequest and added org.tinyradius.packet.MAJIC.interfaces.AccountingRequest as an interface
- Renamed org.tinyradius.packet.RadiusPacket to org.tinyradius.packet.MAJIC.sandboxed.RadiusPacket and added org.tinyradius.packet.MAJIC.interfaces.RadiusPacket as an interface
- Renamed org.tinyradius.util.RadiusEndpoint to org.tinyradius.util.MAJIC.sandboxed.RadiusEndpoint and added org.tinyradius.util.MAJIC.interfaces.RadiusEndpoint as an interface
- Renamed org.tinyradius.attribute.IntegerAttribute to org.tinyradius.attribute.MAJIC.sandboxed.IntegerAttribute and added org.tinyradius.attribute.MAJIC.interfaces.IntegerAttribute as an interface
- Renamed org.tinyradius.dictionary.Dictionary to org.tinyradius.dictionary.MAJIC.sandboxed.Dictionary and added org.tinyradius.dictionary.MAJIC.interfaces.Dictionary as an interface
- Renamed org.tinyradius.dictionary.DictionaryParser to org.tinyradius.dictionary.MAJIC.sandboxed.DictionaryParser and added org.tinyradius.dictionary.MAJIC.interfaces.DictionaryParser as an interface
- Renamed org.tinyradius.attribute.VendorSpecificAttribute to org.tinyradius.attribute.MAJIC.sandboxed.VendorSpecificAttribute and added org.tinyradius.attribute.MAJIC.interfaces.VendorSpecificAttribute as an interface
- Renamed org.tinyradius.dictionary.MemoryDictionary to org.tinyradius.dictionary.MAJIC.sandboxed.MemoryDictionary and added org.tinyradius.dictionary.MAJIC.interfaces.MemoryDictionary as an interface
- Renamed org.tinyradius.dictionary.WritableDictionary to org.tinyradius.dictionary.MAJIC.sandboxed.WritableDictionary and added org.tinyradius.dictionary.MAJIC.interfaces.WritableDictionary as an interface
- Renamed org.tinyradius.util.RadiusUtil to org.tinyradius.util.MAJIC.sandboxed.RadiusUtil and added org.tinyradius.util.MAJIC.interfaces.RadiusUtil as an interface
- Renamed org.tinyradius.util.RadiusClient to org.tinyradius.util.MAJIC.sandboxed.RadiusClient and added org.tinyradius.util.MAJIC.interfaces.RadiusClient as an interface
- Renamed org.tinyradius.dictionary.AttributeType to org.tinyradius.dictionary.MAJIC.sandboxed.AttributeType and added org.tinyradius.dictionary.MAJIC.interfaces.AttributeType as an interface
- Renamed org.tinyradius.attribute.RadiusAttribute to org.tinyradius.attribute.MAJIC.sandboxed.RadiusAttribute and added org.tinyradius.attribute.MAJIC.interfaces.RadiusAttribute as an interface
- Renamed org.tinyradius.attribute.StringAttribute to org.tinyradius.attribute.MAJIC.sandboxed.StringAttribute and added org.tinyradius.attribute.MAJIC.interfaces.StringAttribute as an interface
- Renamed org.tinyradius.dictionary.DefaultDictionary to

org.tinyradius.dictionary.MAJIC.sandboxed.DefaultDictionary and added org.tinyradius.dictionary.MAJIC.interfaces.DefaultDictionary as an interface

Policy Updates

The following transformations were made to MAJICPolicy to create the protection domain for this sandbox:

- Updated the `getPermissions(ProtectionDomain domain)` and `implies(ProtectionDomain domain, Permission permission)` to create a protection domain for MAJIC using isr.cmu.edu.sandbox.MAJICLoader to identify the domain. The following class loaders also load classes into this sandbox's domain:
  - org.tinyradius.dictionary.DefaultDictionary

  The following permissions are granted:
  - `permission java.lang.RuntimePermission "exit.1";`
  - `permission java.net.SocketPermission "localhost", "connect,accept, listen, resolve";`
  - `permission java.lang.RuntimePermission "modifyThread";`
  - `permission java.lang.RuntimePermission "accessDeclaredMembers";`
  - `permission java.util.PropertyPermission "org.apache.commons.logging.LogFactory", "read";`

Generated ClassLoaders

The following class loaders were generated to load classes into this sandbox's protection domain:

- Created isr.cmu.edu.sandbox.MAJICLoader to load classes into the sandboxed protection domain

Updated Main Methods

The following classes had their main methods updated to set the policy to isr.cmu.edu.sandbox.MAJICPolicy:

- org.company.appsec.ap.AuthPlus

Transformations in org.tinyradius.attribute.MAJIC.sandboxed.StringAttribute

The following transformations were made to org.tinyradius.attribute.MAJIC.sandboxed.StringAttribute to ensure it uses classes in the sandbox's protection domain where appropriate:

- Sandboxed a call to org.tinyradius.attribute.RadiusAttribute.<init>:()V in org.tinyradius.attribute.MAJIC.sandboxed.StringAttribute.<init>:()V
- Sandboxed a call to org.tinyradius.attribute.RadiusAttribute.<init>:()V in org.tinyradius.attribute.MAJIC.sandboxed.StringAttribute.<init>:(ILjava/lang/String;)V
- Sandboxed a call to org.tinyradius.attribute.RadiusAttribute.setAttributeType:(I)V in org.tinyradius.attribute.MAJIC.sandboxed.StringAttribute.<init>:(ILjava/lang/String;)V
- Sandboxed a call to org.tinyradius.attribute.MAJIC.sandboxed.StringAttribute.setAttributeValue:(Ljava/lang /String;)V in org.tinyradius.attribute.MAJIC.sandboxed.StringAttribute.<init>:(ILjava/lang /String;)V
- Sandboxed a call to org.tinyradius.attribute.RadiusAttribute.getAttributeData:()[B in org.tinyradius.attribute.MAJIC.sandboxed.StringAttribute.getAttributeValue:()Ljava/lang/String;
- Sandboxed a call to org.tinyradius.attribute.RadiusAttribute.getAttributeData:()[B in

org.tinyradius.attribute.MAJIC.sandboxed.StringAttribute.getAttributeValue:()Ljava/lang/String;
- Sandboxed a call to org.tinyradius.attribute.RadiusAttribute.setAttributeData:([B)V in org.tinyradius.attribute.MAJIC.sandboxed.StringAttribute.setAttributeValue:(Ljava/lang/String;)V
- Sandboxed a call to org.tinyradius.attribute.RadiusAttribute.setAttributeData:([B)V in org.tinyradius.attribute.MAJIC.sandboxed.StringAttribute.setAttributeValue:(Ljava/lang/String;)V

Transformations in org.tinyradius.dictionary.MAJIC.sandboxed.DefaultDictionary

The following transformations were made to org.tinyradius.dictionary.MAJIC.sandboxed.DefaultDictionary to ensure it uses classes in the sandbox's protection domain where appropriate:

- Sandboxed a call to org.tinyradius.dictionary.MemoryDictionary.<init>:()V in org.tinyradius.dictionary.MAJIC.sandboxed.DefaultDictionary.<init>:()V
- Sandboxed a call to org.tinyradius.dictionary.MAJIC.sandboxed.DefaultDictionary.<init>:()V in org.tinyradius.dictionary.MAJIC.sandboxed.DefaultDictionary.<clinit>:()V
- Sandboxed a call to org.tinyradius.dictionary.MAJIC.sandboxed.DefaultDictionary.class$:(Ljava/lang/String;)Ljava/lang/Class; in org.tinyradius.dictionary.MAJIC.sandboxed.DefaultDictionary.<clinit>:()V
- Sandboxed a call to org.tinyradius.dictionary.MAJIC.sandboxed.DictionaryParser.parseDictionary:(Ljava/io/InputStream;Lorg/tinyradius/dictionary/MAJIC/interfaces/WritableDictionary;)V in org.tinyradius.dictionary.MAJIC.sandboxed.DefaultDictionary.<clinit>:()V

SNIP ... 9 more ... (edited for brevity)

Transformations in org.company.appsec.ap.OTPAuthenticator

The following transformations were made to org.company.appsec.ap.OTPAuthenticator to ensure it uses classes in the sandbox's protection domain where appropriate:

- Sandboxed a call to org.tinyradius.util.RadiusClient.<init>:(Ljava/lang/String;Ljava/lang/String;)V in org.company.appsec.ap.OTPAuthenticator.authenticateCredentials:(Ljava/lang/String;Ljava/lang/String;)Z
- Sandboxed a call to org.tinyradius.util.RadiusClient.authenticate:(Ljava/lang/String;Ljava/lang/String;)Z in org.company.appsec.ap.OTPGoldCardAuthenticator.authenticateCredentials:(Ljava/lang/String;Ljava/lang/String;)Z
- Sandboxed a call to org.tinyradius.util.RadiusException.getMessage:()Ljava/lang/String; in org.company.appsec.ap.OTPAuthenticator.authenticateCredentials:(Ljava/lang/String;Ljava/lang/String;)Z

Updates to /home/mmaass/Downloads/CompanyDemo/AuthPlus_sandboxed/AuthPlus.jar

Action	Target
-/+	org.company.appsec.ap.AuthPlus
-/+	org.company.appsec.ap.OTPAuthenticator

Updates to MAJIC_sandboxed_classes.jar

Action	Target
+	org/tinyradius/util/MAJIC/sandboxed/RadiusException.class
+	org/tinyradius/dictionary/MAJIC/sandboxed/Dictionary.class
+	org/tinyradius/attribute/MAJIC/sandboxed/VendorSpecificAttribute.class
+	org/tinyradius/util/MAJIC/sandboxed/RadiusUtil.class
+	org/tinyradius/attribute/MAJIC/sandboxed/RadiusAttribute.class
+	org/tinyradius/util/MAJIC/sandboxed/RadiusEndpoint.class
+	org/tinyradius/dictionary/MAJIC/sandboxed/DefaultDictionary.class
+	org/tinyradius/dictionary/MAJIC/sandboxed/AttributeType.class
+	org/tinyradius/packet/MAJIC/sandboxed/AccessRequest.class
+	org/tinyradius/dictionary/MAJIC/sandboxed/MemoryDictionary.class
+	org/tinyradius/attribute/MAJIC/sandboxed/IntegerAttribute.class
+	org/tinyradius/dictionary/MAJIC/sandboxed/DictionaryParser.class
+	org/tinyradius/packet/MAJIC/sandboxed/AccountingRequest.class
+	org/tinyradius/dictionary/MAJIC/sandboxed/WritableDictionary.class
+	org/tinyradius/packet/MAJIC/sandboxed/RadiusPacket.class
+	org/tinyradius/attribute/MAJIC/sandboxed/StringAttribute.class
+	org/tinyradius/util/MAJIC/sandboxed/RadiusClient.class

Updates to /home/mmaass/Downloads/CompanyDemo /AuthPlus_sandboxed/MAJIC_sandbox.jar

Action	Target
+	org/tinyradius/dictionary/MAJIC/interfaces/DictionaryParser.class
+	org/tinyradius/dictionary/MAJIC/interfaces/AttributeType.class
+	isr/cmu/edu/sandbox/MAJICLoader.class
+	org/tinyradius/dictionary/MAJIC/interfaces/DefaultDictionary.class
+	org/tinyradius/attribute/MAJIC/interfaces/RadiusAttribute.class
+	org/tinyradius/dictionary/MAJIC/interfaces/Dictionary.class
+	org/tinyradius/dictionary/MAJIC/interfaces/WritableDictionary.class
+	org/tinyradius/util/MAJIC/interfaces/RadiusUtil.class
+	org/tinyradius/attribute/MAJIC/interfaces/IntegerAttribute.class
+	isr/cmu/edu/sandbox/MAJICPolicy.class
+	org/tinyradius/util/MAJIC/interfaces/RadiusException.class
+	org/tinyradius/packet/MAJIC/interfaces/RadiusPacket.class
+	org/tinyradius/attribute/MAJIC/interfaces/VendorSpecificAttribute.class
+	org/tinyradius/packet/MAJIC/interfaces/AccountingRequest.class
+	org/tinyradius/attribute/MAJIC/interfaces/StringAttribute.class
+	org/tinyradius/packet/MAJIC/interfaces/AccessRequest.class

+	org/tinyradius/dictionary/MAJIC/interfaces/MemoryDictionary.class
+	org/tinyradius/util/MAJIC/interfaces/RadiusClient.class
+	org/tinyradius/util/MAJIC/interfaces/RadiusEndpoint.class

Bibliography

Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM. ISBN 1-59593-226-7. doi: 10.1145/1102120.1102165. URL `http://doi.acm.org/10.1145/1102120.1102165`. 2.6

Devdatta Akhawe, Prateek Saxena, and Dawn Song. Privilege separation in HTML5 applications. In *USENIX Security*, Security'12, pages 23–23, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2362793.2362816`. 6.2

Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security*, USENIX Security'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association. ISBN 888-7-6666-5555-4. URL `http://dl.acm.org/citation.cfm?id=1929820.1929836`. 2.6

Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *IEEE Symposium on Security and Privacy*, SP '08, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3168-7. doi: 10.1109/SP.2008.30. URL `http://dx.doi.org/10.1109/SP.2008.30`. 2.6

Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy Bounds Checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security*, SSYM'09, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855768.1855772`. 2.6

F. Al Ameiri and K. Salah. Evaluation of Popular Application Sandboxing. In *International Conference for Internet Technology and Secured Transactions (ICITST), 2011*, pages 358 –362, Washington, DC, USA, December 2011. IEEE. 2.1

Diego Albuquerque, Bruno Cafeo, Alessandro Garcia, Simone Barbosa, Silvia Abrahão, and António Ribeiro. Quantifying usability of domain-specific languages. *Journal of Systems and Software*, 101(C):245–259, March 2015. ISSN 0164-1212. doi: 10.1016/j.jss.2014.11.051. URL `http://dx.doi.org/10.1016/j.jss.2014.11.051`. 6.1

Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: A virtual mobile smartphone architecture. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '11, pages 173–187, New York, NY, USA, 2011. ACM.

ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043574. URL http://doi.acm.org/10.1145/2043556.2043574. 2.6, 2.4.2

Fabien Autrel, Nora Cuppens-Boulahia, and Frédéric Cuppens. Enabling dynamic security policy in the Java security manager. In *Conference on Foundations and Practice of Security*, FPS'12, pages 180–193, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-37118-9. doi: 10.1007/978-3-642-37119-6_12. URL http://dx.doi.org/10.1007/978-3-642-37119-6_12. 4.7

Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp Styp-Rekowsky. AppGuard – Fine-grained policy enforcement for untrusted Android applications. In *Workshop on Data Privacy Management and Autonomous Spontaneous Security - Volume 8247*, pages 213–231, New York, NY, USA, 2014. Springer-Verlag New York, Inc. ISBN 978-3-642-54567-2. doi: 10.1007/978-3-642-54568-9_14. URL http://dx.doi.org/10.1007/978-3-642-54568-9_14. 4.7

Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: Preventing sql injection attacks using dynamic candidate evaluations. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '07, pages 12–24, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: 10.1145/1315245.1315249. URL http://doi.acm.org/10.1145/1315245.1315249. 2.6

Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(02):131–177, March 2005. ISSN 1469-7653. doi: 10.1017/S0956796804005453. URL http://journals.cambridge.org/article_S0956796804005453. 3

Dhananjay Bapat, Kevin Butler, and Patrick McDaniel. Towards automated privilege separation. In *Conference on Information Systems Security*, ICISS'07, pages 272–276, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-77085-2, 978-3-540-77085-5. URL http://dl.acm.org/citation.cfm?id=1779274.1779307. 6.2

Jeffrey M. Barnes. *Software Architecture Evolution*. PhD thesis, Carnegie Mellon University, 2013. 2.2

Massimo Bartoletti, Gabriele Costa, Pierpaolo Degano, Fabio Martinelli, and Roberto Zunino. Securing Java with local policies. *Journal of Object Technology*, 8(4):5–32, June 2009. ISSN 1660-1769. doi: 10.5381/jot.2009.8.4.a1. URL http://www.jot.fm/contents/issue_2009_06/article1.html. 4.7

Bernard Berelson. *Content Analysis in Communication Research*. Free Press, Glencoe, IL, USA, 1952. 2.2

Donna Bergmark, Paradee Phempoonpanich, and Shumin Zhao. Scraping the ACM Digital Library. *SIGIR Forum*, 35:1–7, September 2001. ISSN 0163-5840. doi: 10.1145/511144.511146. URL http://doi.acm.org/10.1145/511144.511146. 15

David Berlind. VMware shows Android based virtual machines. http://www.informationweek.com/mobile/mobile-devices/ces-2012-vmware-shows-android-based-virtual-machines/d/d-id/1102135?, 2012. Accessed: 2015-04-30. 2.4.2

F. Besson, T. Blanc, C. Fournet, and A.D. Gordon. From stack inspection to access control: A security analysis for libraries. In *Computer Security Foundations Workshop*, pages 61–75, June 2004. doi: 10.1109/CSFW.2004.1310732. 3

Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security*, SSYM'05, pages 17–17, Berkeley, CA, USA, 2005. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251398.1251415`. 2.6

S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, October 2006. doi: http://doi.acm.org/10.1145/1167473.1167488. 3.4.3, 4.5.5

Thomas Blasing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *Conference on Malicious and Unwanted Software (MALWARE)*, pages 55–62, 2010. 3.6

David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security*, SSYM'04, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251375.1251380`. 6.2

David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *Conference on Computer Aided Verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22109-5. URL `http://dl.acm.org/citation.cfm?id=2032305.2032342`. 6.1

Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, 2002. 4.2.1

Ken Buchanan, Chris Evans, Charlie Reis, and Tom Sepez. Chromium blog: A tale of two pwnies (Part 2). `http://blog.chromium.org/2012/06/tale-of-two-pwnies-part-2.html`, June 2012. 5.2.1

Mihai Bucicoiu, Lucas Davi, Razvan Deaconescu, and Ahmad-Reza Sadeghi. XiOS: Extended application sandboxing on iOS. In *ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 43–54, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3245-3. doi: 10.1145/2714576.2714629. URL `http://doi.acm.org/10.1145/2714576.2714629`. 4.7

David Budgen and Pearl Brereton. Performing systematic literature reviews in software engineering. In *International Conference on Software Engineering*, ICSE '06, pages 1051–1052, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: 10.1145/1134285.1134500. URL `http://doi.acm.org/10.1145/1134285.1134500`. 2.2

Tony Capaccio. China counterfeit parts in u.s. military boeing, l3 aircraft. http://www.bloomberg.com/news/articles/2011-11-07/counterfeit-parts-from-china-found-on-raytheon-boeing-systems, 2011. 1.1

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 *ACM Conference on Computer and Communications Security (CCS)*, CCS '10, pages 212–223, New York, NY, USA, 2010a. ACM. ISBN 978-1-4503-0245-6. doi: 10.1145/1866307.1866332. URL http://doi.acm.org/10.1145/1866307.1866332. 2.1

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 *ACM Conference on Computer and Communications Security (CCS)*, pages 212–223. ACM, 2010b. ISBN 978-1-4503-0245-6. doi: 10.1145/1866307.1866332. URL http://doi.acm.org/10.1145/1866307.1866332. 3.6, 4.7

Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI '06, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298470. 2.6

Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '09, pages 45–58, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629581. URL http://doi.acm.org/10.1145/1629575.1629581. 2.6

P. Centonze, R.J. Flynn, and M. Pistoia. Combining static and dynamic analysis for automatic identification of precise access-control policies. In *Annual Computer Security Applications Conference (ACSAC)*, pages 292–303, Dec 2007. doi: 10.1109/ACSAC.2007.39. 4.2

Cert-IST. Anatomy of a malicious PDF file. http://goo.gl/VlLmU, February 2010. URL http://www.cert-ist.com/eng/ressources/Publications_ArticlesBulletins/VersVirusetAntivirus/malicious_pdf/. 5

Ramesh Chandra, Taesoo Kim, Meelap Shah, Neha Narula, and Nickolai Zeldovich. Intrusion recovery for database-backed web applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '11, pages 101–114, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043567. URL http://doi.acm.org/10.1145/2043556.2043567. 2.2

Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '08, pages 39–50, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-810-7. doi: 10.1145/1455770.1455778. URL http://doi.acm.org/

10.1145/1455770.1455778. 2.6

Eric Yawei Chen, Jason Bau, Charles Reis, Adam Barth, and Collin Jackson. App Isolation: Get the security of multiple browsers with just one. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '11, pages 227–238, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046734. URL http://doi.acm.org/10.1145/2046707.2046734. 2.6

Shuo Chen, David Ross, and Yi-Min Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '07, pages 2–11, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: 10.1145/1315245.1315248. URL http://doi.acm.org/10.1145/1315245.1315248. 2.6

Andy Chou. On detecting heartbleed with static analysis. http://security.coverity.com/blog/2014/Apr/on-detecting-heartbleed-with-static-analysis.html, 2014. 1.1

Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding Caller-sensitive Method Vulnerabilities: A Class of Access Control Vulnerabilities in the Java Platform. In *International Workshop on State Of the Art in Program Analysis*, SOAP 2015, pages 7–12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3585-0. doi: 10.1145/2771284.2771286. URL http://doi.acm.org/10.1145/2771284.2771286. 3.1.1, 4

Michael R Clarkson and Fred B Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. 5.1

Zack Coker, Michael Maass, Tianyuan Ding, Claire Le Goues, and Joshua Sunshine. Evaluating the flexibility of the Java sandbox. In *Annual Computer Security Applications Conference*, ACSAC 2015, pages 1–10, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3682-6. doi: 10.1145/2818000.2818003. URL http://doi.acm.org/10.1145/2818000.2818003. 2.4.2

Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code. In *International World Wide Web Conference (WWW)*, pages 281–290, 2010. ISBN 978-1-60558-799-8. doi: 10.1145/1772690.1772720. URL http://doi.acm.org/10.1145/1772690.1772720. 3.6

Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Parsimonious server security. In *USENIX Conference on System Administration*, LISA '00, pages 355–368, Berkeley, CA, USA, 2000. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1045502.1045548. 4.7

Richard S. Cox, Steven D. Gribble, Henry M. Levy, and Jacob Gorm Hansen. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy*, SP '06, pages 350–364, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2574-1. doi: 10.1109/SP.2006.4. URL http://dx.doi.org/10.1109/SP.2006.4. 2.1, 2.6

J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy*,

pages 292–307, Washington, DC, USA, May 2014. IEEE. doi: 10.1109/SP.2014.26. 2.6

John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '07, pages 351–366, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294295. URL http://doi.acm.org/10.1145/1294261.1294295. 2.6

Mary Ann Davidson. No, you really cant. http://seclists.org/isn/2015/Aug/4, 2015. 1.1

Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. FlowFox: A web browser with flexible and precise information flow control. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '12, pages 748–759, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382275. URL http://doi.acm.org/10.1145/2382196.2382275. 2.1, 2.6

Guillaume Delugre. Bypassing ASLR and DEP on Adobe Reader X - Sogeti ESEC Lab. http://esec-lab.sogeti.com/post/Bypassing-ASLR-and-DEP-on-Adobe-Reader-X, June 2012. 1.3.1, 5.2.1

Norman K. Denzin and Yvonna S. Lincoln. Introduction: The Discipline and Practice of qualitative research. In *The Sage Handbook of Qualitative Research*. Sage, Thousand Oaks, CA, USA, 4th edition, 2011. ISBN 978-1-4129-7417-2. 2.2

Brandon Dixon. PDF X-RAY. https://github.com/9b/pdfxray_public, 2012. 5

Alan M. Dunn, Michael Z. Lee, Suman Jana, Sangman Kim, Mark Silberstein, Yuanzhong Xu, Vitaly Shmatikov, and Emmett Witchel. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI'12, pages 61–75, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387887. 2.6

Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '05, pages 17–30, New York, NY, USA, 2005. ACM. ISBN 1-59593-079-5. doi: 10.1145/1095810.1095813. URL http://doi.acm.org/10.1145/1095810.1095813. 2.3.3, 2.6

Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI '06, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298463. 2.6

Jose Esparza. peepdf - PDF analysis and creation/modification tool. http://code.google.com/p/peepdf/, 2012. 5

Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC:

Group collaboration using untrusted cloud resources. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924967. 2.6

Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Analyzing and Detecting Malicious Flash Advertisements. In *Annual Computer Security Applications Conference (ACSAC)*, pages 363–372, 2009. ISBN 978-0-7695-3919-5. doi: 10.1109/ACSAC.2009.41. URL http://dx.doi.org/10.1109/ACSAC.2009.41. 3.6

Cdric Fournet and Andrew D. Gordon. Stack Inspection: Theory and Variants. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 307–318, 2002. ISBN 1-58113-450-9. doi: 10.1145/503272.503301. URL http://doi.acm.org/10.1145/503272.503301. 3

Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. Shellzer: a tool for the dynamic analysis of malicious shellcode. In *Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 61–80, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23643-3. doi: 10.1007/978-3-642-23644-0_4. URL http://dx.doi.org/10.1007/978-3-642-23644-0_4. 5

Steve Friedl. Best practices for UNIX chroot() operations. http://www.unixwiz.net/techtips/chroot-practices.html, 2012. 5.1

Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Symposium on Foundations of Software Engineering*, FSE '10, pages 147–156, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-791-2. doi: 10.1145/1882291.1882315. URL http://doi.acm.org/10.1145/1882291.1882315. 1

Lee Garber. Have Java's Security Issues Gotten out of Hand? In *2012 IEEE Technology News*, pages 18–21, 2012. 3

Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '03, pages 193–206, New York, NY, USA, 2003a. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945464. URL http://doi.acm.org/10.1145/945445.945464. 2.6

Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM Symposium on Operating Systems Principles*, SOSP '03, pages 193–206, New York, NY, USA, 2003b. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945464. URL http://doi.acm.org/10.1145/945445.945464. 5.2.1

Dimitris Geneiatakis, Georgios Portokalidis, Vasileios P. Kemerlis, and Angelos D. Keromytis. Adaptive defenses for commodity software through virtual application partitioning. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '12, pages 133–144, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382214. URL http://doi.acm.org/10.1145/2382196.2382214. 2.1

Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vi-

taly Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *ACM Conference on Computer and Communications Security (CCS)*, pages 38–49. ACM, 2012. 3.6

Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI'12, pages 47–60, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL `http://dl.acm.org/citation.cfm?id=2387880.2387886`. 2.6

Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security*, Security'12, pages 40–40, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2362793.2362833`. 2.6

Li Gong. Java security: a ten year retrospective. In *Annual Computer Security Applications Conference (ACSAC)*, pages 395–405, 2009. 1.5, 3.7

Li Gong and Gary Ellison. *Inside Java (TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003. 3

Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, 1997. 3

Dan Goodin. At hacking contest, Google Chrome falls to third zero-day attack (Updated). `http://arstechnica.com/business/news/2012/03/googles-chrome-browser-on-friday.ars`, March 2012. 1.3.1, 3

Adam Gowdiak. Security Vulnerabilities in Java SE. Technical Report SE-2012-01 Project, Security Explorations, 2012. 3, 3.1

P.J. Graydon, J.C. Knight, and E.A. Strunk. Assurance based development of critical systems. In *Dependable Systems and Networks*, pages 347–357, Washington, DC, USA, June 2007. IEEE. doi: 10.1109/DSN.2007.17. 2.4.1

Kevin W Hamlen, Greg Morrisett, and Fred B Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(1):175–205, 2006. 5.1

Norm Hardy. The Confused Deputy: (or Why Capabilities Might Have Been Invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, October 1988. ISSN 0163-5980. URL `http://dl.acm.org/citation.cfm?id=54289.871709`. 3.1.3

James Hartley. Current findings from research on structured abstracts. *Journal of the Medical Library Association*, 92(3):368–371, 2004. 2.4.1

Guy Helmer, Johnny Wong, and Subhasri Madaka. Anomalous Intrusion Detection System for Hostile Java Applets. *J. Syst. Softw.*, 55(3):273–286, January 2001. ISSN 0164-1212. doi: 10.1016/S0164-1212(00)00076-5. URL `http://dx.doi.org/10.1016/S0164-1212(00)00076-5`. 3.6

Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'd my gadgets go? In *IEEE Symposium on Security and Privacy*, SP '12, pages 571–585, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4681-0. doi: 10.1109/SP.2012.39. URL http://dx.doi.org/10.1109/SP.2012.39. 2.6

Jim Hogg. Control Flow Guard. http://blogs.msdn.com/b/vcblog/archive/2014/12/08/visual-studio-2015-preview-work-in-progress-security-feature.aspx, 2015. Accessed: 2015-04-30. 2.4.2

Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Librando: Transparent code randomization for just-in-time compilers. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '13, pages 993–1004, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516675. URL http://doi.acm.org/10.1145/2508859.2516675. 2.6

David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004. ISSN 0362-1340. doi: 10.1145/1052883.1052895. URL http://doi.acm.org/10.1145/1052883.1052895. 3.2.2

Wine HQ. AppDB Adobe Reader. http://goo.gl/Fx9pd, 2014. 5.3.3

IBM Security Systems. IBM X-Force threat intelligence report. http://www.ibm.com/security/xforce/, February 2014. 3

Hajime Inoue and Stephanie Forrest. Inferring Java security policies through dynamic sandboxing. In Hamid R. Arabnia, editor, *Programming Languages and Compilers*, pages 151–157. CSREA Press, 2005. ISBN 1-932415-75-0. URL http://dblp.uni-trier.de/db/conf/plc/plc2005.html#Inoue05. 4.1.2, 4.7

IntelliJ. IntelliJ IDEA inspections list (632). http://www.jetbrains.com/idea/documentation/inspections.jsp, 2014. 3.7

S. Jana, D.E. Porter, and V. Shmatikov. TxBox: Building secure, efficient sandboxes with system transactions. In *IEEE Symposium on Security and Privacy*, pages 329–344, Washington, DC, USA, May 2011. IEEE. doi: 10.1109/SP.2011.33. 2.6, 5.1

Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007. 2.4.2

Leonard Kaufman and Peter J Rousseeuw. *Finding groups in data: An introduction to cluster analysis*, volume 344. John Wiley & Sons, New York, NY, USA, 2009. 2.2.3

T. Kelly. *Arguing Safety - A Systematic Approach to Safety Case Management*. PhD thesis, Department of Computer Science, University of York, 1999. 2.4.1

Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: Lightweight kernel protection against return-to-user attacks. In *USENIX Security*, Security'12, pages 39–39, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2362793.2362832. 2.6

Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In

Conference on Computer Aided Verification, CAV '08, pages 423–427, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70543-7. doi: 10.1007/978-3-540-70545-1_40. URL http://dx.doi.org/10.1007/978-3-540-70545-1_40. 6.1

Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering - A systematic literature review. *Information and Software Technology*, 51(1):7–15, January 2009. ISSN 0950-5849. doi: 10.1016/j.infsof.2008.09.009. URL http://dx.doi.org/10.1016/j.infsof.2008.09.009. 2.2

Dietrich Knauth. Defective parts threaten space, missile missions: Gao. http://www.law360.com/articles/259610/defective-parts-threaten-space-missile-missions-gao, 2011. 1.1

J. Knight. The importance of security cases: Proof is good, but not enough. *IEEE Security Privacy Magazine*, 13(4):73–75, July 2015. ISSN 1540-7993. doi: 10.1109/MSP.2015.68. 2.4.1

Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. The state of the art in end-user software engineering. *ACM Computing Surveys*, 43(3):21:1–21:44, April 2011. ISSN 0360-0300. doi: 10.1145/1922649.1922658. URL http://doi.acm.org/10.1145/1922649.1922658. 6.1

Klaus H. Krippendorff. *Content Analysis: An Introduction to Its Methodology*. Sage, Thousand Oaks, CA, USA, 2013. ISBN 1-4129-8315-0. 2.2

Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294293. URL http://doi.acm.org/10.1145/1294261.1294293. 2.6

Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992. ISSN 0360-0300. doi: 10.1145/130844.130856. URL http://doi.acm.org/10.1145/130844.130856. 1

Mary Landesman. Free PDF readers: Alternatives to Adobe Reader and Acrobat. http://antivirus.about.com/od/securitytips/tp/Free-Pdf-Readers-Alternatives-To-Adobe-Reader-Acrobat.htm, 2010. 5

Pavel Laskov and Nedim Srndic. Static detection of malicious JavaScript-bearing PDF documents. In *Annual Computer Security Applications Conference (ACSAC)*, pages 373–382, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0672-0. doi: 10.1145/2076732.2076785. URL http://doi.acm.org/10.1145/2076732.2076785. 5

Anna Lázár, Dániel Ábel, and Tamás Vicsek. Modularity measure of networks with overlapping communities, 2009. 2.3

Du Li and Witawas Srisa-an. Quarantine: A Framework to Mitigate Memory Errors in

JNI Applications. In *Conference on Principles and Practice of Programming in Java (PPPJ)*, pages 1–10, 2011. ISBN 978-1-4503-0935-6. doi: 10.1145/2093157.2093159. URL http://doi.acm.org/10.1145/2093157.2093159. 3.6, 4.7

Ninghui Li, Ziqing Mao, and Hong Chen. Usable mandatory integrity protection for operating systems. In *IEEE Symposium on Security and Privacy*, SP '07, pages 164–178, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2848-1. doi: 10.1109/SP.2007.37. URL http://dx.doi.org/10.1109/SP.2007.37. 2.6

Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. MiniBox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 409–420, Philadelphia, PA, June 2014a. USENIX Association. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_yanlin. 2.1

Zhiwei Li, Warren He, Devdatta Akhawe, and Dawn Song. The emperor's new password manager: Security analysis of web-based password managers. In *USENIX Security*, 2014b. 3.6

Dawn Lim. Counterfeit chips plague u.s. missile defense. http://www.wired.com/2011/11/counterfeit-missile-defense/, 2011. 1.1

W.C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5): 23–30, Sept 1994. ISSN 0740-7459. doi: 10.1109/52.311048. 1

C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. Protecting against unexpected system calls. In *USENIX Security*, SSYM'05, pages 16–16, Berkeley, CA, USA, 2005. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251398.1251414. 2.6

R.P. Lippmann and K.W. Ingols. An Annotated Review of Past Papers on Attack Graphs. Technical Report Project Report IA-1, Lincoln Laboratory, Massachusetts Institute of Technology, 03 2005. URL https://www.ll.mit.edu/mission/cybersec/publications/publication-files/full_papers/0502_Lippmann.pdf. 6.3

Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, and David Svoboda. *The CERT Oracle Secure Coding Standard for Java*. SEI Series in Software Engineering. Addison-Wesley Professional, 1st edition, September 2011. ISBN 978-0-321-80395-5. 4

Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association. ISBN 1-880446-10-3. URL http://dl.acm.org/citation.cfm?id=647054.715771. 4.7

Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *IEEE Symposium on Security and Privacy*, SP '09, pages 331–346, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3633-0. doi: 10.1109/SP.2009.33. URL http://dx.doi.org/10.1109/SP.2009.33. 2.6

Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. AdJail: Practical

enforcement of confidentiality and integrity policies on web advertisements. In *USENIX Security*, USENIX Security'10, pages 24–24, Berkeley, CA, USA, 2010. USENIX Association. ISBN 888-7-6666-5555-4. URL http://dl.acm.org/citation.cfm?id=1929820.1929852. 2.6

Michael Maass, William L. Scherlis, and Jonathan Aldrich. In-nimbo Sandboxing. In *Symposium and Bootcamp on the Science of Security*, HotSoS '14, pages 1:1–1:12, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2907-1. doi: 10.1145/2600176.2600177. URL http://doi.acm.org/10.1145/2600176.2600177. 2.4.1

Martin Maechler, Peter Rousseeuw, Anja Struyf, Mia Hubert, and Kurt Hornik. *cluster: Cluster Analysis Basics and Extensions*. R Foundation for Statistical Computing, Vienna, Austria, 2014. R package version 1.15.2 — For new features, see the 'Changelog' file (in the package source). 2.2.3

Davide Maiorca, Giorgio Giacinto, and Igino Corona. A pattern recognition system for malicious PDF files detection. In *International Conference on Machine Learning and Data Mining in Pattern Recognition*, MLDM'12, pages 510–524, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31536-7. doi: 10.1007/978-3-642-31537-4_40. URL http://dx.doi.org/10.1007/978-3-642-31537-4_40. 5

Davide Maiorca, Igino Corona, and Giorgio Giacinto. Looking at the bag is not enough to find the bomb: An evasion of structural methods for malicious PDF files detection. In *ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 119–130, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1767-2. doi: 10.1145/2484313.2484327. URL http://doi.acm.org/10.1145/2484313.2484327. 5

D. Malkhi, M.K. Reiter, and A.D. Rubin. Secure execution of Java applets using a remote playground. In *IEEE Security and Privacy*, pages 40–51, May 1998. doi: 10.1109/SECPRI.1998.674822. 5.2.3

P.K. Manadhata and J.M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, May 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.60. 6.3

Heiko Mantel. On the composition of secure systems. In *IEEE Symposium on Security and Privacy*, pages 88–101. IEEE, 2002. 5.1

Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with API integrity and multi-principal modules. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '11, pages 115–128, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043568. URL http://doi.acm.org/10.1145/2043556.2043568. 2.6

Lorenzo Martignoni, Pongsin Poosankam, Matei Zaharia, Jun Han, Stephen McCamant, Dawn Song, Vern Paxson, Adrian Perrig, Scott Shenker, and Ion Stoica. Cloud Terminal: Secure access to sensitive applications from untrusted systems. In *USENIX Annual Technical Conference*, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2342821.2342835. 5.2.3

A. Mauczka, C. Schanes, F. Fankhauser, M. Bernhart, and T. Grechenig. Mining security

changes in FreeBSD. In *Mining Software Repositories (MSR)*, pages 90–93, Washington, DC, USA, May 2010. IEEE. doi: 10.1109/MSR.2010.5463289. 2.4.2

Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. *SIGPLAN Notices*, 47(10):683–702, October 2012. ISSN 0362-1340. doi: 10.1145/2398857.2384666. URL http://doi.acm.org/10.1145/2398857.2384666. 6.1

Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *USENIX Security*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267336.1267351. 2.2, 2.6

Richard A. McCormack. Boeing's planes are riddled with chinese counterfeit electronic components. http://www.manufacturingnews.com/news/counterfeits615121.html, 2012. 1.1

Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. How low can you go?: Recommendations for hardware-supported minimal TCB code execution. *SIGOPS Oper. Syst. Rev.*, 42(2):14–25, March 2008a. ISSN 0163-5980. doi: 10.1145/1353535.1346285. URL http://doi.acm.org/10.1145/1353535.1346285. 5.2.2

Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. *SIGOPS Oper. Syst. Rev.*, 42(4):315–328, April 2008b. ISSN 0163-5980. doi: 10.1145/1357010.1352625. URL http://doi.acm.org/10.1145/1357010.1352625. 5.2.2

Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010a. IEEE Computer Society. ISBN 978-0-7695-4035-1. doi: 10.1109/SP.2010.17. URL http://dx.doi.org/10.1109/SP.2010.17. 2.6

Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158. IEEE, 2010b. 5.2.2

J. McLean. Is the trusted computing base concept fundamentally flawed? In *IEEE Symposium on Security and Privacy*, SP '97, pages 2–, Washington, DC, USA, 1997. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=882493.884390. 4

Liz McQuarrie, Ashutosh Mehra, Suchit Mishra, Kyle Randolph, and Ben Rogers. Inside Adobe Reader Protected Mode - part 1 - design. http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-part-1-design.html, October 2010a. 5.2.1

Liz McQuarrie, Ashutosh Mehra, Suchit Mishra, Kyle Randolph, and Ben Rogers. Inside adobe reader protected mode - part 2 - the sandbox process. http://blogs.adobe.com/security/2010/10/inside-adobe-reader-protected-mode-part-2-the-sandbox-process.html, October 2010b. 5.2.1

Liz McQuarrie, Ashutosh Mehra, Suchit Mishra, Kyle Randolph, and Ben Rogers. Inside Adobe Reader Protected Mode - Part 3 - Broker Process, Policies, and Inter-Process Communication. http://blogs.adobe.com/asset/2010/11/inside-adobe-reader-protected-mode-part-3-broker-process-policies-and-inter-process-communication.html, November 2010c. 5.2.1

Leo A. Meyerovich and Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *IEEE Symposium on Security and Privacy*, SP '10, pages 481–496, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4035-1. doi: 10.1109/SP.2010.36. URL http://dx.doi.org/10.1109/SP.2010.36. 2.6

Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, May 2006. ISBN 0-7356-2214-0. 5.4.2

James Mickens. Pivot: Fast, synchronous mashup isolation using generator chains. In *IEEE Symposium on Security and Privacy*, SP '14, pages 261–275, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.24. URL http://dx.doi.org/10.1109/SP.2014.24. 2.6

Alexander Moshchuk, Helen J. Wang, and Yunxin Liu. Content-based Isolation: Rethinking isolation policy design on client systems. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '13, pages 1167–1180, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516722. URL http://doi.acm.org/10.1145/2508859.2516722. 2.6

Mozilla Security Team. We are the Mozilla Security Community, Ask Us Anything! http://www.reddit.com/r/netsec/comments/1b3vcx/we_are_the_mozilla_security_community_ask_us/c93bm9a, April 2013. 1.2

MSDN. Understanding and working in protected mode internet explorer. http://msdn.microsoft.com/en-us/library/bb250462(v=vs.85).aspx, February 2011. 5.1

G. Nakamoto, J. Schwefler, and K. Palmer. Desktop demilitarized zone. In *Military Communications Conference*, pages 1487–1492, Nov 2011. doi: 10.1109/MILCOM.2011.6127516. 5.2.3

Nedim Srndic and Pavel Laskov. Detection of malicious PDF files based on hierarchical document structure. In *Network and Distributed System Security Symposium*, 2013. 5

Peter G. Neumann. Inside risks: Insecurity about security? *Communications of the ACM*, 33:170–170, August 1990. ISSN 0001-0782. doi: 10.1145/79173.79113. URL http://doi.acm.org/10.1145/79173.79113. 4

Peter G. Neumann. *Computer Related Risks*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995. ISBN 0-201-55805-X. 1.1

Anh Nguyen-Tuong, J.D. Hiser, M. Co, J.W. Davidson, J.C. Knight, N. Kennedy, D. Melski, W. Ella, and D. Hyde. To b or not to b: Blessing OS commands with software DNA shotgun sequencing. In *Dependable Computing Conference*, pages 238–249, Washington, DC, USA, May 2014. IEEE. doi: 10.1109/EDCC.2014.13. 2.4.1

Ruslan Nikolaev and Godmar Back. VirtuOS: An operating system with kernel virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '13, pages 116–132, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522719. URL http://doi.acm.org/10.1145/2517349.2522719. 2.6

NIST. National vulnerability database (NVD) national vulnerability database (CVE-2010-2937). https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-2937, 2010a. 5.5.2

NIST. National vulnerability database (NVD) national vulnerability database (CVE-2010-3019). https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3019, 2010b. 5.5.1

NIST. National vulnerability database (NVD) national vulnerability database (CVE-2011-2949). https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-2949, 2011. 5.5.2

NIST. Vulnerability Summary for CVE-2012-0507. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-0507, June 2012. 3.1.3

NIST. Vulnerability Summary for CVE-2012-4681. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-4681, October 2013a. 3.1.3

NIST. National vulnerability database (NVD) national vulnerability database (CVE-2013-0768). https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0768, 2013b. 5.5.1

Ben Niu and Gang Tan. Monitor integrity protection with space efficiency and separate compilation. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '13, pages 199–210, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516649. URL http://doi.acm.org/10.1145/2508859.2516649. 2.6

NRC. *Critical Code: Software Producibility for Defense*. The National Academies Press, 2010. URL http://www.nap.edu/openbook.php?record_id=12979. 6.2

Jorge Obes and Justin Schuh. Chromium blog: A tale of two pwnies (Part 1). http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html, May 2012. 5.2.1

Terri Oda, Glenn Wurster, P. C. van Oorschot, and Anil Somayaji. SOMA: Mutual approval for included content in web pages. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '08, pages 89–98, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-810-7. doi: 10.1145/1455770.1455783. URL http://doi.acm.org/10.1145/1455770.1455783. 2.6

Microsoft Office Support. What is protected view? - word - office.com. http://office.microsoft.com/en-us/word-help/what-is-protected-view-HA010355931.aspx, 2012. 5.2.1

Jeong Wook Oh. Recent Java exploitation trends and malware. Technical Report BH-US-12, Black Hat, 2012. URL https://media.blackhat.com/bh-us-12/Briefings/Oh/BH_US_12_Oh_Recent_Java_Exploitation_Trends_and_Malware_WP.pdf. 3

OpenHUB. OpenSSL project summary. https://www.openhub.net/p/openssl, 2015a.

1.1

OpenHUB. Spring project summary. https://www.openhub.net/p/spring, 2015b. 1.1

OpenSSL. Openssl vulnerabilities. https://www.openssl.org/news/vulnerabilities.html, 2015. 1.1

Oracle. Default Policy Implementation and Policy File Syntax. http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html, 2014a. 3.1.1

Oracle. Java Virtual Machine Tool Interface. https://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/, 2014b. 3.2.2

Oracle. Permissions in the JDK. http://docs.oracle.com/javase/7/docs/technotes/guides/security/permissions.html, 2014c. 3.1.1

Oracle. Permissions in the JDK. http://docs.oracle.com/javase/7/docs/technotes/guides/security/permissions.html, 2014d. 4.1.2

Oracle. HPROF: A Heap/CPU Profiling Tool. http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html, 2015a. 4.5.5

Oracle. Chapter 5. Loading, Linking, and Initializing. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3, 2015b. 4.4.3

Oracle. Default Policy Implementation and Policy File Syntax. http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html, 2015c. 4.1.1

C. Owen, D. Grove, T. Newby, A. Murray, C. North, and M. Pope. PRISM: Program replication and integration for seamless MILS. In *IEEE Symposium on Security and Privacy*, pages 281–296, Washington, DC, USA, May 2011. IEEE. doi: 10.1109/SP.2011.15. 2.6

Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the Gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy*, SP '12, pages 601–615, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4681-0. doi: 10.1109/SP.2012.41. URL http://dx.doi.org/10.1109/SP.2012.41. 2.6

Bryan Parno, Jonathan M. McCune, Dan Wendlandt, David G. Andersen, and Adrian Perrig. CLAMP: Practical prevention of large-scale data leaks. In *IEEE Symposium on Security and Privacy*, SP '09, pages 154–169, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3633-0. doi: 10.1109/SP.2009.21. URL http://dx.doi.org/10.1109/SP.2009.21. 2.6

Patrik. Two new vulnerabilities in Adobe Acrobat Reader. http://www.f-secure.com/weblog/archives/00001671.html, April 2009. 5

Mathias Payer, Tobias Hartmann, and Thomas R. Gross. Safe Loading - A foundation for secure execution of untrusted programs. In *IEEE Symposium on Security and Privacy*, SP '12, pages 18–32, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4681-0. doi: 10.1109/SP.2012.11. URL http://dx.doi.org/10.1109/SP.2012.11. 2.6, 5.1

Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy*, SP '08, pages 233–247, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3168-7. doi: 10.1109/SP.2008.24. URL `http://dx.doi.org/10.1109/SP.2008.24`. 2.6

Nick L. Petroni, Jr. and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '07, pages 103–115, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: 10.1145/1315245.1315260. URL `http://doi.acm.org/10.1145/1315245.1315260`. 2.6

Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. ADsafety: Type-based verification of javascript sandboxing. In *USENIX Security*, SEC'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2028067.2028079`. 2.2

Chromium Project. Chromium sandbox. `http://www.chromium.org/developers/design-documents/sandbox/`, 2012. 5.2.1

Niels Provos. Improving host security with system call policies. In *USENIX Security*, SSYM'03, pages 18–18, Berkeley, CA, USA, 2003. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251353.1251371`. 2.2, 2.6, 2.4.2, 4.7

Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *USENIX Security*, 2003. 3.6

Cynthia D. Mulrow R. Brian Haynes, Douglas G. Altman Edward J. Huth, and Martin J. Gardner. More Informative Abstracts Revisited. *Annals of Internal Medicine*, 113(1):69–76, 1990. doi: 10.7326/0003-4819-113-1-69. URL `http://dx.doi.org/10.7326/0003-4819-113-1-69`. 2.4.1

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014. URL `http://www.R-project.org/`. 2.2.3

M.A. Rahman and E. Al-Shaer. A formal framework for network security design synthesis. In *Conference on Distributed Computing Systems (ICDCS)*, pages 560–570, July 2013. doi: 10.1109/ICDCS.2013.70. 6.3

Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic html. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI '06, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL `http://dl.acm.org/citation.cfm?id=1298455.1298462`. 2.1, 2.6

Michael F. Ringenburg and Dan Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '05, pages 354–363, New York, NY, USA, 2005. ACM. ISBN 1-59593-226-7. doi: 10.1145/1102120.1102166. URL `http://doi.acm.org/10.1145/1102120.1102166`. 2.3.3, 2.6

Benjamin D. Rodes, John C. Knight, Anh Nguyen-Tuong, Jason D. Hiser, Michele Co, and Jack W. Davidson. A case study of security case development. In *Engineering Systems for Safety*, pages 187–204, Newcastle upon Tyne, UK, February 2015. Safety-Critical Systems Club. 2.4.1

Dan Rosenberg. Poking holes in AppArmor profiles. `http://blog.azimuthsecurity.com/2012/09/poking-holes-in-apparmor-profiles.html`, 2012. Accessed: 2015-04-30. 2.3.2

J. M. Rushby. Design and verification of secure systems. *SIGOPS Oper. Syst. Rev.*, 15(5):12–21, December 1981. ISSN 0163-5980. doi: 10.1145/1067627.806586. URL `http://doi.acm.org/10.1145/1067627.806586`. 1.3

Giovanni Russello, Arturo Blas Jimenez, Habib Naderi, and Wannes van der Mark. FireDroid: Hardening security in almost-stock Android. In *Annual Computer Security Applications Conference*, ACSAC '13, pages 319–328, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2015-3. doi: 10.1145/2523649.2523678. URL `http://doi.acm.org/10.1145/2523649.2523678`. 4.7

Paul Sabanal and Mark Yason. Playing in the Reader X sandbox. *Black Hat USA Briefings*, July 2011. 5.2.1

Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, 1975. 1.3

Prateek Saxena, David Molnar, and Benjamin Livshits. SCRIPTGARD: Automatic context-sensitive sanitization for large-scale legacy web applications. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '11, pages 601–614, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046776. URL `http://doi.acm.org/10.1145/2046707.2046776`. 2.6

Fabian Scherschel. Google warns of using Adobe Reader - Particularly on Linux. `http://www.h-online.com/security/news/item/Google-warns-of-using-Adobe-Reader-particularly-on-Linux-1668153.html`, August 2012. 5

Johannes Schlumberger, Christopher Kruegel, and Giovanni Vigna. Jarhead Analysis and Detection of Malicious Java Applets. In *Annual Computer Security Applications Conference (ACSAC)*, pages 249–257, 2012. ISBN 978-1-4503-1312-4. doi: 10.1145/2420950.2420988. URL `http://doi.acm.org/10.1145/2420950.2420988`. 3.6

Fred B. Schneider. Towards fault-tolerant and secure agentry. In *International Workshop on Distributed Algorithms*, WDAG '97, pages 1–14, London, UK, UK, 1997. Springer-Verlag. ISBN 3-540-63575-0. URL `http://dl.acm.org/citation.cfm?id=645954.675785`. 4

Margrit Schreier. *Qualitative Content Analysis in Practice*. SAGE Publications Ltd, Thousand Oaks, CA, USA, 1st edition, March 2012. ISBN 978-1849205931. 2.1, 2.2

Z. Cliffe Schreuders, Tanya McGill, and Christian Payne. The state of the art of application restrictions and sandboxes: A survey of application-oriented access controls and their shortfalls. *Computers & Security*, 32:219–241, February 2013a. ISSN 01674048. doi:

10.1016/j.cose.2012.09.007. URL http://linkinghub.elsevier.com/retrieve/pii/S0167404812001435. 2.1

Z.Cliffe Schreuders, Christian Payne, and Tanya McGill. The Functionality-Based Application Confinement Model. *International Journal of Information Security*, 12(5): 393–422, 2013b. ISSN 1615-5262. doi: 10.1007/s10207-013-0199-4. URL http://dx.doi.org/10.1007/s10207-013-0199-4. 4.7

Justin Schuh. Chromium Blog: The road to safer, more stable, and flashier flash. http://blog.chromium.org/2012/08/the-road-to-safer-more-stable-and.html, August 2012. 5.2.1

David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *USENIX Security*, USENIX Security'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association. ISBN 888-7-6666-5555-4. URL http://dl.acm.org/citation.cfm?id=1929820.1929822. 13, 2.6, 2.4.2

Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294294. URL http://doi.acm.org/10.1145/1294261.1294294. 2.6

Peter Sewell and Jan Vitek. Secure composition of insecure components. In *IEEE Computer Security Foundations Workshop*, pages 136–150. IEEE, 1999. 5.1

Vivek Shandilya, Chris B. Simmons, and Sajjan Shiva. Use of attack graphs in security systems. *Journal of Computer Networks and Communications*, 2014:1–13, 2014. doi: 10.1155/2014/818957. URL http://dx.doi.org/10.1155/2014/818957. 6.3

Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: Taming the native beast of the jvm. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '10, pages 201–211, New York, NY, USA, 2010a. ACM. ISBN 978-1-4503-0245-6. doi: 10.1145/1866307.1866331. URL http://doi.acm.org/10.1145/1866307.1866331. 2.4, 2.3.1, 2.6

Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: Taming the Native Beast of the JVM. In *ACM Conference on Computer and Communications Security (CCS)*, pages 201–211, 2010b. ISBN 978-1-4503-0245-6. doi: 10.1145/1866307.1866331. URL http://doi.acm.org/10.1145/1866307.1866331. 3.6, 4.7

Lenin Singaravelu, Calton Pu, Hermann Hartig, and Christian Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev.*, 40(4):161–174, April 2006. ISSN 0163-5980. doi: 10.1145/1218063.1217951. URL http://doi.acm.org/10.1145/1218063.1217951. 5.2.2

Abhishek Singh and Shray Kapoor. Get Set Null Java Security. http://www.fireeye.com/blog/technical/2013/06/get-set-null-java-security.html, June 2013. 3, 3.1.3

Charles Smutz and Angelos Stavrou. Malicious PDF detection using metadata and struc-

tural features. In *Annual Computer Security Applications Conference (ACSAC)*, ACSAC '12, pages 239–248, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1312-4. doi: 10.1145/2420950.2420987. URL http://doi.acm.org/10.1145/2420950.2420987. 5

Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. On breaking SAML: Be whoever you want to be. In *USENIX Security*, pages 21–21, 2012. 3.6

Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Conference on Information Systems Security*, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89861-0. doi: 10.1007/978-3-540-89862-7_1. URL http://dx.doi.org/10.1007/978-3-540-89862-7_1. 6.1

C. Spearman. The proof and measurement of association between two things. *American Journal of Psychology*, 15:88–103, 1904. 2.2.3

StackOverflow. What is the point of invokeinterface? http://stackoverflow.com/questions/1504633/what-is-the-point-of-invokeinterface, 2015. 4.5.5

Scott Stender. Inside adobe reader protected mode - part 4 - the challenge of sandboxing. http://blogs.adobe.com/asset/2010/11/inside-adobe-reader-protected-mode-part-4-the-challenge-of-sandboxing.html, November 2010. 5.2.1

Marc Stiegler, Alan H. Karp, Ka-Ping Yee, Tyler Close, and Mark S. Miller. Polaris: Virus-safe computing for windows XP. *Communications of the ACM*, 49(9):83–88, September 2006. ISSN 0001-0782. doi: 10.1145/1151030.1151033. URL http://doi.acm.org/10.1145/1151030.1151033. 5.2.1

J. Stuckman and J. Purtilo. Mining security vulnerabilities from linux distribution metadata. In *Software Reliability Engineering Workshops (ISSREW)*, pages 323–328, Washington, DC, USA, Nov 2014. IEEE. 2.4.2

Jeffrey Stylos and Brad A. Myers. The implications of method placement on api learnability. In *Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 105–112, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-995-1. doi: 10.1145/1453101.1453117. URL http://doi.acm.org/10.1145/1453101.1453117. 6.1

Mengtao Sun and Gang Tan. JVM-Portable Sandboxing of Java's Native Libraries. In *European Symposium on Research in Computer Security (ESORICS)*, pages 842–858, 2012. 3.6, 4.7

Weiqing Sun, R. Sekar, Gaurav Poothia, and Tejas Karandikar. Practical Proactive Integrity Preservation: A basis for malware defense. In *IEEE Symposium on Security and Privacy*, SP '08, pages 248–262, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3168-7. doi: 10.1109/SP.2008.35. URL http://dx.doi.org/10.1109/SP.2008.35. 2.2, 2.6

David Svoboda. Anatomy of Java Exploits. http://www.cert.org/blogs/certcc/post.cfm?EntryID=136, 2013. 3, 4

David Svoboda and Yozo Toda. Anatomy of Another Java Zero-Day Exploit. https://oracleus.activeevents.com/2014/connect/sessionDetail.ww?SESSION_ID=2120, September 2014. 4

Adobe Systems. Adobe PDF library SDK | Adobe Developer Connection. http://www.adobe.com/devnet/pdf/library.html, August 2012. 5

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Eternal war in memory. In *IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4. doi: 10.1109/SP.2013.13. URL http://dx.doi.org/10.1109/SP.2013.13. 2.4.2

Jack Tang. Exploring Control Flow Guard in Windows 10. http://blog.trendmicro.com/trendlabs-security-intelligence/exploring-control-flow-guard-in-windows-10/, 2015. Accessed: 2015-09-10. 2.4.2

Shuo Tang, Haohui Mai, and Samuel T. King. Trust and protection in the Illinois browser operating system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924945. 2.2

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 *Asia Pacific Software Engineering Conference (APSEC)*, pages 336–345, December 2010. doi: http://dx.doi.org/10.1109/APSEC.2010.46. 3, 3.2.1

Christopher Theisen. Automated attack surface approximation. In *Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 1063–1065, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2807563. URL http://doi.acm.org/10.1145/2786805.2807563. 6.3

Zacharias Tzermias, Giorgos Sykiotakis, Michalis Polychronakis, and Evangelos P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *European Workshop on System Security*, EUROSEC '11, pages 4:1–4:6, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0613-3. doi: 10.1145/1972551.1972555. URL http://doi.acm.org/10.1145/1972551.1972555. 5

Peleus Uhley and Rajesh Gwalani. Inside flash player protected mode for firefox. http://blogs.adobe.com/asset/2012/06/inside-flash-player-protected-mode-for-firefox.html, June 2012. 5.2.1

W3C. 4.8.11 the canvas element - HTML5. http://www.w3.org/TR/html5/the-canvas-element.html#the-canvas-element, March 2012. 5.5.1

Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.*, 27(5):203–216, December 1993. ISSN 0163-5980. doi: 10.1145/173668.168635. URL http://doi.acm.org/10.1145/173668.168635. 1.2, 2.1

Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *IEEE Symposium on Security and Privacy*, pages 52–63, 1998a. doi: 10.1109/SECPRI.1998.

674823. 3

Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *IEEE Symposium on Security and Privacy*, pages 52–63, 1998b. doi: 10.1109/SECPRI.1998.674823. 4.1.1

Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *Symposium on Operating Systems Principles*, SOSP '97, pages 116–128, New York, NY, USA, 1997. ACM. ISBN 0-89791-916-5. doi: 10.1145/268998.266668. URL `http://doi.acm.org/10.1145/268998.266668`. 4

Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and communication abstractions for web browsers in mashupos. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '07, pages 1–16, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294263. URL `http://doi.acm.org/10.1145/1294261.1294263`. 2.6

Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. Compac: Enforce component-level access control in Android. In *ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 25–36, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2278-2. doi: 10.1145/2557547.2557560. URL `http://doi.acm.org/10.1145/2557547.2557560`. 4.7

Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '09, pages 545–554, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-894-0. doi: 10.1145/1653662.1653728. URL `http://doi.acm.org/10.1145/1653662.1653728`. 2.6

Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary Stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '12, pages 157–168, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382216. URL `http://doi.acm.org/10.1145/2382196.2382216`. 2.6

Charles B. Weinstock, Howard F. Lipson, and John Goodenough. Arguing Security: Creating Security Assurance Cases. Technical report, Software Engineering Institute, Carnegie Mellon University, 2007. 2.4.1

Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. Device driver safety through a reference validation mechanism. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI'08, pages 241–254, Berkeley, CA, USA, 2008. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855741.1855758`. 2.6

Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *IEEE Conference on Automated Software Engineering (ASE)*, pages 323–333, Nov 2013. doi: 10.1109/ASE.2013.6693091. 6.2

Glenn Wurster and Paul C. van Oorschot. A control point for reducing root abuse of

file-system privileges. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '10, pages 224–236, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0245-6. doi: 10.1145/1866307.1866333. URL http://doi.acm.org/10.1145/1866307.1866333. 2.6

Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for android applications. In *USENIX Security*, Security'12, pages 27–27, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2362793.2362820. 2.2, 2.6, 4.7

Yan Yan, Massimilliano Menarini, and William Griswold. Mining software contracts for software evolution. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 471–475, Washington, DC, USA, Sep 2014. IEEE. 2.4.2

B. Yee, D. Sehr, G. Dardyk, J.B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, pages 79–93, Washington, DC, USA, May 2009. IEEE. doi: 10.1109/SP.2009.25. 2.6, 2.4.2, 5.1

A.A. Younis, Y.K. Malaiya, and I. Ray. Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability. In *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, pages 1–8, Jan 2014. doi: 10.1109/HASE.2014.10. 6.3

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazieres. Making information flow explicit in histar. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI '06, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298481. 2.3.3, 2.6

Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '11, pages 29–40, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046713. URL http://doi.acm.org/10.1145/2046707.2046713. 2.6

Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *USENIX Security*, SEC'13, pages 369–382, Berkeley, CA, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4. URL http://dl.acm.org/citation.cfm?id=2534766.2534798. 2.1

Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, L. Szekeres, S. McCamant, D. Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy*, pages 559–573, Washington, DC, USA, May 2013. IEEE Computer Society. doi: 10.1109/SP.2013.44. 2.1, 2.6

Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '11, pages 203–216, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043576.

URL http://doi.acm.org/10.1145/2043556.2043576. 2.6

Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security*, SEC'13, pages 337–352, Berkeley, CA, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4. URL http://dl.acm.org/citation.cfm?id=2534766.2534796. 2.6, 2.4.1

Qun Zhong, Nigel Edwards, and Owen Rees. Operating System Support for the Sandbox Method and Its Application on Mobile Code Security. Technical Report HPL-97-153, HP Laboratories Bristol, 1997. URL http://www.hpl.hp.com/techreports/97/HPL-97-153.pdf. 4