Soft Boundaries for Statically Enforceable Protection Domains

by

Jonathan Bannet

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

[Signatures]

Dan S. Wallach, Chair, Assistant Professor
Computer Science

Robert Cartwright
Robert Cartwright, Professor
Computer Science

Keith Cooper, Professor
Computer Science

Houston, Texas
May, 2004
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI Microform 1425802

Copyright 2005 by ProQuest Information and Learning Company. All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346
Soft Boundaries for Statically Enforceable Protection Domains

Jonathan Bannet

Abstract

As safe languages, such as Java, find themselves leveraged to run larger, multi-task applications, many of the security concerns historically solved by hardware enforced protection domains are again sneaking their way into code. Previous schemes to enforce protection domains within the Java language have accepted run-time overhead and severe limitations on sharing semantics as a reasonable cost to achieve security. This thesis proposes soft boundaries to attain the benefits of separation without additional run-time overhead and without compromising the Java semantics. Soft boundaries use static analysis to assist the programmer in isolating components, protecting both a program’s data and its control flow. Soft boundaries can be deployed either independently or in conjunction with previous separation schemes to further bolster separation.
Acknowledgments

I'd like to thank Dan Wallach for his support, input, and vision for this research, for pushing me forward when I wanted nothing more than to quit, and for the opportunity to join his research group.

I'd like to thank Keith Cooper for his cheerful attitude and making time in his busy schedule.

I'd like to thank Robert “Corky” Cartwright for getting me interested in research in the first place. None of this would have happened without his help and guidance.

I'd like to thank Ryan Bergauer for his work on early stages of the project and his unending patience.

I'd like to thank Rassul Zarinfar for helping proof read early versions of the text. It was a time when even I couldn’t “bare” reading more than a few lines without cringing.

I'd like to thank Eric Allen. He’s been an invaluable friend and a role model, both as a researcher and as a person. When Eric gave me my first opportunity to do theoretical research I knew nothing about real computer science; when I finished working with him I knew that I knew nothing.

Last, I would like to thank Algis Rudys, my collaborator in this research, for both his help with the research and his help with in cleaning up the rough edges of this document.
Contents

Abstract ii
Acknowledgments iii
List of Illustrations vii
List of Tables viii

1 Introduction 1
1.1 Related work ................................................. 4
1.1.1 Operating system based protection domains ................. 4
1.1.2 Language based protection domains ......................... 4
1.1.3 Static analysis and security .................................. 6

2 Issues in the design and deployment of boundaries 7
2.1 Three real world programs .................................. 8
2.1.1 The Miniature Java Web Server ............................... 9
2.1.2 Pastry .................................................. 10
2.1.3 Sandstorm ............................................... 11
2.1.4 Empirical conclusions ..................................... 11
2.2 Brief taxonomy of cross-domain bugs ......................... 12
2.2.1 Pointer leakage .......................................... 13
2.2.2 Time of check to time of use bugs ......................... 13
2.2.3 Thread capture and denial of service ....................... 14

3 Design 15
3.1 The goals and limitations of soft boundaries ................. 15
3.1.1 Static analysis vs. run-time enforcement .................. 16
3.2 Soft boundaries and protecting the data of a domain .................. 17
  3.2.1 Garnering intended boundary semantics ............................... 18
  3.2.2 A specification of soft boundary with data semantics .......... 20
  3.2.3 Critical data ....................................................... 23
3.3 Soft boundaries and protecting the control flow of a domain .......... 24
  3.3.1 Specifying control semantics for soft boundaries ............... 25
3.4 Implementation of soft boundaries ...................................... 26
  3.4.1 Implementation of cross-domain data sharing analysis .......... 28
  3.4.2 Implementation of API checker ................................... 32
  3.4.3 Implementation of critical data checker .......................... 34
  3.4.4 Implementation of thread tracker .................................. 35

4 Results ................................................................. 37
  4.1 Results from domain interaction analysis ............................... 37
  4.2 Results from API checking ............................................ 41
    4.2.1 Miniature Java Web Server ...................................... 41
    4.2.2 Pastry .......................................................... 42
    4.2.3 Sandstorm ...................................................... 43
    4.2.4 Challenging the closed world assumption with API checking ... 43
  4.3 Critical data analysis ................................................. 44
    4.3.1 Miniature Java Web Server ...................................... 44
    4.3.2 Pastry .......................................................... 46
    4.3.3 Challenging the closed world assumption with critical data checking 47
  4.4 Thread tracker results ................................................. 47
    4.4.1 Miniature Java Web Server ...................................... 48
    4.4.2 Pastry .......................................................... 48
    4.4.3 Seda ........................................................... 51
    4.4.4 Challenging the closed world assumption with thread tracking ... 51

5 Conclusion ............................................................. 52
# Illustrations

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Boundaries for the Miniature Java Web Server</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of API usage.</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Example Sacrificial interface usage.</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Illustration of domain interaction sets</td>
<td>30</td>
</tr>
<tr>
<td>3.5</td>
<td>Code example for errors and warnings</td>
<td>33</td>
</tr>
<tr>
<td>3.6</td>
<td>Illustration of the thread check utility</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>Code examples of INFO-FLOW and TOCTOU sets</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>API error in Seda codebase.</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>Code demonstrating that the servlet can access the server</td>
<td>45</td>
</tr>
<tr>
<td>4.4</td>
<td>Code tracing server.</td>
<td>46</td>
</tr>
<tr>
<td>4.5</td>
<td>Thread checker analysis for Web Server</td>
<td>49</td>
</tr>
<tr>
<td>4.6</td>
<td>Thread tracker’s examples of dangerous Pastry code.</td>
<td>50</td>
</tr>
</tbody>
</table>
Tables

4.1 Profile of applications ........................................ 38
4.2 Domain interaction table ...................................... 38
4.3 Domain self mutation table ................................... 40
4.4 API errors and warnings ...................................... 41
4.5 Thread tracker results ......................................... 48
Chapter 1

Introduction

Since the early stages of multi-user computing, operating system designers have recognized the importance of protection domains, both to protect users from each other and to protect users from themselves. In operating systems such as UNIX and Windows NT, protection domains are enforced through hardware level support for the process abstraction, name spaces, access lists, and a host of other techniques. These domains deny a nosy or malicious process access to data belonging to other processes on the same system. Similarly, a novice or absent minded user is unable to accidentally corrupt the kernel, denying service to themselves and others.

Language based security mechanisms grew in parallel to their hardware counterparts. Safe languages, like Java and LISP, can prevent many of the bugs that have historically haunted the security community. Security calamities resulting from bugs such as buffer overflows can be entirely avoided. Currently, Java is one of the most commonly employed safe language and offers a series of security mechanisms, including type safety and prevention of direct memory access. These mechanisms, combined with access specifiers, facilitate the separation of components and protect the privacy of a domain’s data.

Despite its strengths, the application of Java to large, complex projects such as servlet containers, peer-to-peer systems, and operating systems has underscored the weaknesses of Java’s existing separation scheme. The current Java separation mechanisms rely on the programmer designing the program to protect sensitive information. The language itself lacks the capability to overcome a programmer’s shortcomings and guarantee that the resources of one component remain untouched by others.

Such separation is necessary to both protect secure data within a privileged component and prevent the escalation of bugs; a bug in an application running inside a server should not crash other applications running in the same server or the server itself. Programmers
sometimes run each application in its own Java Virtual Machine (JVM) and have the JVMs communicate via RMI, but due to the overhead of multiple JVMs this solution is often infeasible [2]. Consequently, there exist several proposals to bolster separation between applications in a single JVM. These proposals range from ad hoc techniques to operating system style processes.

Each proposal comes with a unique set of trade-offs (discussed in more detail in Section 1.1). Ad hoc techniques such as privilege separation [26] offer neither guarantees nor aid in making a program secure; like Java, they continue to rely entirely on the programmers’ savvy. Using hardware-style process abstractions, here call hard boundaries, are a preferable alternative because they offer guarantees about inter-domain communication. Safe termination of tasks, resource monitoring, protection of data, and protection of control flow, are all commonly offered by hard boundary implementations, but hard boundaries remain imperfect as they incur an overhead and change the Java semantics. The change in semantics is often deemed necessary to cope with the tension between Java style sharing and security; Java style sharing introduces a class of security bugs that does not exist when domains are restricted from touching each other’s data. But in cases where programmers are unwilling to accept either the semantic changes or the run-time overhead imposed by restricted sharing, previous boundary schemes are insufficient.

Lightweight proposals for both safe termination [31] and monitoring memory usage [25] exist. The run-time cost of both schemes is reasonably small, and both leave the Java sharing semantics intact. This thesis takes the next step toward an alternative to hard boundaries and focuses on protecting the data and the control flow of a task, proposing soft boundaries to that end. Soft boundaries assist the programmer in partitioning a program and protecting the resources of each component. That the JVM already has built in mechanisms for protecting both data and control flow implies that previous hard boundary schemes are employed, in large, to compensate for programmer’s errors when separating components. An alternative to throwing valuable CPU cycles at the problem and convoluting Java programming is assisting the programmer in manually partitioning the code. Soft boundaries leverage the security mechanisms already in place in the JVM; these mechanisms are seldom applied to their fullest, leading to numerous security bugs in
applications. These bugs could have been avoided or mitigated had the program been properly structured and designed with component isolation in mind. Unlike the hard boundaries used previously in language based solutions and currently in operating systems, soft boundaries allow Java style direct object sharing and do so without significant overhead. Because they limit rather than change the semantics of data sharing, backward compatibility is also made easier.

Soft boundaries investigate the application of static analysis to the problem of program separation. Static analysis has recently received increasing attention for its usefulness in ferreting bugs from large, complex applications (Section 1.1). Work to date has focused on either deducing the programmer’s intent from the code or providing additional, low level syntax to specify it. This work shows each approach to be both insufficient and error-prone when applied independently to soft boundaries. This thesis leverages simple, high level security policies.

Unfortunately, soft boundaries assume full program analysis. Although a similar approach can be applied without full program analysis, doing so requires greater effort from the programmer, making backward compatibility difficult. This thesis investigates which separation problems lend themselves to static analysis and offers solutions as appropriate. Not all separation can be statically enforced, yet where static enforcement of boundaries is fitting, it integrates seamlessly with the current Java language. Soft boundaries and static analysis have tremendous potential to improve application security while still integrating well with legacy code. Chapter two introduces several applications that could benefit from soft boundaries and some of the bugs soft boundaries must address. Chapter three discusses the capabilities of soft boundaries, their specification, and their enforcement. Chapter four presents the results of applying soft boundaries to the applications introduced in Chapter two.
1.1 Related work

1.1.1 Operating system based protection domains

Lampson posits three ways that a certain user, or program, can cause another user, or program, harm: modification of private data, access to private data, or interference with their service [21]. Lampson was the first to define protection domains, but many operating systems have since adopted his principle in the form of processes that run in independent execution spaces [7, 28]. Capability based systems extend the boundary idea to explicitly limit resource access within a system [15].

1.1.2 Language based protection domains

Designers of multi-user computer realized that safe languages could also be used to enforce cross-domain protections. One of the earliest such instances was the Burroughs B5000 [23]. As applications grew in size, complexity, and functionality, application designers began to share concerns about separation with operating system designers. Applying years of literature and research into operating system security, researchers began trying to adopt solutions similar to those proposed by operating system designers. Process-like abstractions in language-based systems have been proposed to achieve protection across security domains. Much of the recent work in this area has been done with respect to the Java programming language [6, 1, 17]. These systems use message passing or restrict sharing to achieve security. Although quite effective, each proposal has its drawbacks. Cornell’s JKernel suffers from heavy overhead associated with its capability-based approach. Communication between domains occurs through a mechanism similar to RMI and objects that are not capabilities must be copied across boundaries [17]. KaffeOS uses shared heaps to allow data to be shared between domains. KaffeOS incurs an 11% overhead when implemented over a freely available JVM, overhead that measures only the additional runtime that KaffeOS requires. This overhead is less than the JKernel, but may not translate if KaffeOS were implemented over a commercial JVM. Though superior to the JKernel, KaffeOS has several drawbacks. First, there are strict limitations on what data can be assigned to the fields of shared data and what data types can be shared [19]. Second, when
these restrictions are violated a run-time exception is thrown. Third, heaps for shared data must be created with a fixed size, which impedes sharing dynamic data structures, such as hash-tables. Fourth, KaffeOS disallows cross-domain method invocation, breaking the usual Java semantics.

Stricter domains have also been proposed. MVM offers domains for running multiple applications within the same JVM [5] that eliminate any communication between applications. Such a restriction requires the domains to have separate copies of the system classes to disallow sharing through static fields, a problem proven to be quite troublesome for standard JVMs [1].

Hawblitzel and Eiken recognized the flaws in prior separation schemes and proposed an extended type system to circumvent those flaws; they leverage their type system to facilitate thread termination and protect both domain data and domain threads [20]. They, like us, seek to use static analysis to provide separation. Luna, their Java extension, introduced shared types and required all shared objects to be of shared type. The shared types, called remote pointers, are not interchangeable with their unshareable counterparts. Thus, a function written to take an unshared Integer will reject a shared Integer. One solution they proposed is to write numerous copies of each function, one for each required combination of remote pointer and non-remote pointer, but they recognize the difficulty in writing programs under such a paradigm and, to avoid the limitations on program expressibility, propose copying data out of remote pointers at every invocation. A solution that reduces the scheme to copying data across boundaries. Finally, the new type system makes application of Luna to legacy code difficult.

The division of code into domains has also been applied informally to prevent the escalation of bugs [26]. Provos et al. partition the OpenSSH code base into domains and apply the “Principle of Least Privilege”, which is to minimize the privilege given to a component in a program [32].

Recently, other language-based mechanisms have been proposed for cross-domain protection in language-based systems. Stack inspection is used in Java to provide access control to sensitive system functions [39], while a recent Scheme system called MrEd provides support for thread termination and some resource management [14]. Soft termination pro-
vides support for the termination of individual domains in a language-based system, and finally, Price et al. [25] and Wick et al. [41] support per-domain memory accounting in Java and Scheme, respectively, using the garbage collector.

1.1.3 Static analysis and security

Static analysis has seen a fair amount of success in the security community. Wagner et al. took a large step in applying static analysis to security when they utilized integer range analysis to detect buffer overflows, the most commonly exploited security hole [37]. Chess [3] and Evans et al. [11], who independently performed more precise analysis and uncovered a wider class of bugs, including formatting bugs. To assist in gaining this precision, both schemes offer the programmer a fine-grained language for specifying assumptions over the code.

A second class of security bug, race conditions, have also been addressed using static analysis. Flanagan uses static analysis with a great degree of success to find race conditions in code [13]. The approach requires program annotation, but includes a program to aid the annotation process. Engler's work with RacerX [9], which catches deadlocks and race conditions, does not require program annotations. Both projects require significant programmer energy to distinguish false positives.

Porat applied static analysis to the problem of distinguishing mutable class fields from their static counterpart [24]. Knowing that a field is never mutated guarantees that a use of that field will not result in a time of check to time of use attack, and the information in the field can never get overwritten; the field is effectively final, though not declared as such.

Static analysis has also been used to secure information flow. The work was pioneered as early as 1975, by Denning [8], and proven sound by Valpano et al. who represent Denning's proposal with a type system [36].

Engler et al. were responsible for arguably the most interesting application of static analysis to security [10]. They demonstrated that by observing patterns in a large code base, a program can deduce potential bugs with a high degree of accuracy. Using this technique, they uncovered null pointer dereferences, deadlock issues, and even points where the programmers' unspecified invariants are violated.
Chapter 2

Issues in the design and deployment of boundaries

Application separation has been shown to have multiple benefits, including improving the robustness of code, mitigating the damage that bugs in one application can cause others, and protecting the greater system from malicious applications. The goal of language level protection domains is to provide the same level of security that hardware has afforded processes at the OS level without the added restrictions of processes. Hawblitzel and von Eicken make the case for language based separation, rather than leveraging micro-kernels or other such designs [18]. This work agrees with Hawblitzel and von Eicken arguments that language based protection allows for both a finer grain of expression and greater expressibility and focuses instead on the potential forms such protection can take.

In Java, we can formally define a domain as a subset of the set of all classes in a program, and a boundary as all points of interaction between domains. Domains have two resources, control flow and data, which both require protection; a second domain should not arbitrarily access data belonging to a protected domain, and a second domain should not interrupt the core functionality of a protected domain.

Hard boundaries accomplish separation by restricting inter-domain behavior. While some restrictions on behavior are entirely transparent—in most cases, limiting the amount of data a domain can allocate to prevent a denial of service attack affects neither the semantics of a program nor its functionality—other restrictions, such as dynamically enforced restrictions on the free sharing of data between domains, alter the Java language more deeply. For some projects, such restrictions may be acceptable, and the programmer will be willing to accept not invoking methods in other domains or severe limitations on sharing and treatment of data. But these solutions incur a performance cost and tend to violate one of the properties that makes Java so attractive to programmers. If the goals of hard boundaries
were achievable without run-time overhead and without changes to the Java semantics, they would be greatly preferable.

Java offers the tools to construct boundaries between domains. Because object pointers are unforgeable and there is no direct memory access, access specifiers on fields and methods can function to restrict access to privileged data. Such restrictions are complemented by the Java class loader, which prevents one class from using classes loaded by another, unrelated class loader. Because untrusted applications require further separation, Java offers further security mechanisms. The byte code verifier, class loader, and security manager, allow for the creation of secure yet flexible sandboxes, which can be used to safely run untrusted applications [38]. But tension lies in Java’s free sharing model. It is too easy for programmers to make mistakes and accidentally leave data unprotected. Notice that even in the process style boundary mechanisms, enforced at run-time, the danger has not been entirely eliminated. The assertion that those mechanisms make is that a programmer will not accidentally share data when all data sharing occurs explicitly. Such an assumption may not always be correct.

Soft boundaries leverage static analysis to garner some of the benefits of separation without the cost of previous schemes. Due to the sharing problems that hard boundaries do not entirely remedy, some of the conclusions that soft boundaries arrive at can also be applied to hard boundaries. This chapter begins by contextualizing soft boundaries with several real world examples. In the examples, it is postulated what boundaries and domains would look like, and it is hypothesized what purpose the boundaries could serve. The chapter ends by giving a brief taxonomy of some of the inter-domain bugs that soft boundaries must address.

2.1 Three real world programs

This section examines three real-world applications: The Java Miniature Web Server, Pastry, and Sandstorm. All three are frameworks for running other applications. To prevent the escalation of bugs, the framework itself must be robust from errors in the application.
2.1.1 The Miniature Java Web Server

It is often necessary to dynamically tailor Internet pages to the specification of each individual. One API used to this end is Common Gateway Interface (CGI). Because each CGI application runs in a separate process, CGI suffers from poor performance. Proposals exist to try and remedy CGI’s efficiency shortcomings\(^1\), but the solution, FastCGI, suffers from security flaws. Sun’s response to the conundrum of serving dynamic content is the servlet. Because servlets can run in the same JVM as the server, they can achieve performance near that of FastCGI and, due to sandboxing, are almost as secure as CGI. They have the additional advantage of being able to plug into a server and dynamically extend its functionality, without restarting the server.

Servlets have a well defined API that limits their interaction with the server and prevents them from disturbing the server. To service a request, the server invokes the servlet, passing it a request object and a response object. The request object that is passed to the servlet should contain all information required by the servlet to fulfill the request. Theoretically, the servlet should never need to call into the server; it updates the response object with its result and then returns. Further complicating the situation, servlets cannot necessarily trust even each other. Separation between the servlets themselves is required to minimize access to confidential data and prevent escalation of bugs. This extreme separation is imperative because if not properly managed, a bug in a servlet could cascade, draining valuable server resources, leaking private information, possibly even crashing the server. A boundary between a server and its servlet should prevent such escalation by enforcing strict separation.

Moreover, the server cannot trust that the servlet will always perform its task and return. Thus the server cannot trust the servlet with unnecessary amounts of confidential data or depend on the servlet’s results.

One program this thesis examines is the Miniature Java Web Server \(^2\). The Java Miniature Web Server is a lightweight web server and servlet container, consisting of 5,392 lines. The server comes prepackaged with two servlets included: one for hosting files, and an-

---
\(^1\)http://www.fastcgi.com/
\(^2\)http://tjws.sourceforge.net
other for hosting CGI. We manually partitioned the code into web-server and two servlet domains, placing only the two servlets in the latter. The servlet container's small size makes detailed analysis of the code more intuitive than in some of the other large systems that this thesis examines.

2.1.2 Pastry

Peer-to-peer (P2P) systems have been used for a variety of purposes. One of their initial goals was to offer robustness against data loss to a group of users. Each file is duplicated on several computers; if one computer crashes, the file remains safe. Since then, Gnutella and other P2P file sharing utilities have demonstrated their effectiveness as mediums for file swapping.

P2P, though filled with potential, is ripe for abuse and mistakes. A single malicious or malfunctioning application could invoke behavior in its node that would disrupt the functionality of the entire network. Such an application could, for example, change the routing table to misdirect messages. Separation between node and application should aid in minimizing the damage such applications could cause the network, and one application in the network should be unable to disturb the service of others or crash the entire network.

This means that a node's functionality should not depend entirely on its application's well-being. The application may take significant time to run or simply run forever. If the node has to wait on the application to service further messages it will either appear to be dead while it waits on the application or, if the application crashes, the node will die as well.

A boundary around the Pastry infrastructure should also prevent an application from changing the network's state in undesired ways but, unlike servlets, P2P systems lack a standard API specified by Sun. In a P2P network, the ways in which an application can affect the system's state depends on the implementation. This complicates the problem of how to specify what data should be shared; some systems desire a greater amount of sharing and interaction than that allowed to servlets.

Another program this thesis examines is FreePastry. FreePastry is a 59,405 line library for P2P applications that allows for efficient sharing of data [29]. The library handles lo-
cating data in the network and is intended to be generally secure from a malicious user tampering with it. PAST is a distributed file storage system that runs over the Pastry framework and was used during our testing process [30].

### 2.1.3 Sandstorm

For high demand Internet services such as Amazon.com or Yahoo, service latency is critical. The server should be robust enough to gracefully recover from bugs and continue servicing requests.

The Seda architecture is intended to perform well with parallelized Internet applications that may cause other architectures to sputter [40]. Due to the critical, stressful nature under which Seda operates, it is imperative that bugs in a single application cannot escalate to slow Seda down. Sandstorm is a backbone for the Seda architecture that encapsulates the important functionality to make designing software in the Seda architecture easier. The Seda package is 39,948 lines. This thesis examines Haboob, an HTTP server implemented over Sandstorm. The Haboob/Sandstorm relationship is different from either of the previous programs because there exists mutual trust.

### 2.1.4 Empirical conclusions

The examples demonstrate that not all domains need to be protected to the same degree. In the example of the Miniature Java Web Server, there may be many servlets with weak to no security requirements. In Pastry, the application examined was less security critical than the uptime of nodes, but if Pastry were to support extensible applications, the applications themselves would also require protection. While Sandstorm and Haboob are closely connected, mutually trusting programs.

The three examples above imply a hierarchical relationship. The critical domain has access to more secure data and needs to be shielded from bugs in the client because if the critical domain goes down, all applications crash. Shielding should protect the domain’s resources and restrict the other domains’ access to private data. But soft boundaries should be robust enough to apply when the hierarchical model does not. If two domains are in-
teracting and neither trusts the other, a soft boundary should protect both domains to the same degree one domain is protected in the case of asymmetric trust. The conclusions from the examples with programs sharing asymmetric trust still apply in the case of a symmetric lack of trust. With a complete lack of trust, a component should depend neither on results from another component or be able to adversely affect the other component’s state.

The examples also give further insight into the construction of soft boundaries. Several possibilities exist for how a designer can define a boundary and domains: by disjoint sets of classes, by disjoint sets of instances, or by class loader to name a few (in Java all classes must be loaded into the JVM by a class loader); a program can use several class loaders but, for security, classes loaded by unrelated class loaders can not see each other. The examples, however, support domains defined by the location of the class file on disk, the classpath. This result is unsurprising since Java’s object oriented nature implies that differing levels of trust will not typically occur within a single class.

Domains defined by class path also allow the most flexibility and ease of use. Two classes are in distinct domains if they are reachable from different paths in the class path. A single domain may include classes from several paths. If two paths in distinct domains reach the same class then the more specific takes precedence, and paths reachable by soft or hard link are given the lowest precedence. Other methods of coding boundaries also have merit, but programmers’ familiarity with class paths makes this approach simple and intuitive to coders and designers.

2.2 Brief taxonomy of cross-domain bugs

Protection domains are a well studied, high level, simple abstraction to express intent [21]. In operating systems, protection domains have been used to express that two domains do not share trust. The boundary abstraction is intended to highlight sensitive points in the higher level design allowing for specialized tools and proper program construction to add security to inter-domain actions.

The deployment of soft boundaries requires a more educated and aware programmer than the deployment of hard boundaries. The programmer must be aware of the varieties
of bugs that can occur, be able to identify their causes, and be able to implement their solutions.

Cross-boundary bugs can be quite varied. Some bugs, such as accidentally shared data, are easy to identify. Others, such as the time of check to time of use bug (TOCTOU), are harder to identify. We now present a taxonomy of cross-domain security bugs can help further define the purpose and functionality of soft boundaries, as well as aid a programmer by raising awareness of security dangers.

2.2.1 Pointer leakage

One common bugs in applications is pointer leakage. Leakage occurs when the programmer intends for a piece of data to remain private, but accidentally leaves a way for another domain to affect that data. The pointer can be used to either garner private information, or change state critical to the data's owner.

2.2.2 Time of check to time of use bugs

A time of check to time of use bug (TOCTOU) is caused by a data invariant being broken. One block of code performs a check on a piece of data and depends on the invariance of that data for subsequent operations. If another block of code changes that data after the check, but before the dependent operation, a TOCTOU bug has been coded. The TOCTOU attack, the exploitation of a TOCTOU bug to cause undesirable behavior, is an attack with particularly dangerous potential. A well-known example of a TOCTOU attack is a temporary file race condition [4].

Historically, TOCTOU bugs have been considered a multi-threaded problem, the timing of the attack required a second thread to change an invariant from under a first thread, but there is no reason that a TOCTOU bug can't also be single-threaded. Programmers make a frightening number of oversights; with a single thread, if data invariants are checked, the data is passed into another domain and, upon returning, the data is used, a TOCTOU bug could arise. The programmer expected the checked invariants to hold and just forgot to recheck them when the data returned.
Automation of TOCTOU bug detection is difficult because the bugs violate the expectations of the programmer rather than some statically verifiable property. For efficiency, convenience, or functionality, it is common programming practice for two domains to concurrently access or update a shared data structure. Two domains with pointers to a piece of data, or even simultaneously using that piece of data, do not signal a bug. What makes the concurrent access a TOCTOU bug is the programmer’s ignorance regarding the implications of the access or failure to follow locking protocols correctly. Any mechanism that makes the programmer more aware of shared data will make the programmer more capable of avoiding TOCTOU bugs.

2.2.3 Thread capture and denial of service

Thread capture occurs when a function in class Caller invokes a method f() in class Receiver and the expectation that class Receiver will return is violated. If domain A passes an object O to domain B and domain B performs a call-back into this object, domain A can cause a denial of service by having O not return control to B. Similarly, if domain B directly calls a method in a class in domain A, then domain A not returning control to domain B will cause a denial of service. Shapiro discusses further security threats arising from inter-thread communication in [33].

In a web server, a programmer may write the code used in the server, but not have control over the servlets hosted by it or, alternatively, the servlets may simply be poorly written. Robustness of the server should not depend on its servlets being bug free. From the perspective of the protected domain, calls into the untrusted domain should always be safe. Under normal circumstances—a protected domain coded with even the slightest attention to security—the absence of up-calls guarantees that a domain will not be able to either kill the thread of a protected domain or capture the thread.
Chapter 3

Design

Where traditional operating systems enforce "hard boundaries" with hardware separation, and most previous language-based solutions rely on run-time checks to do so, we wish to use static analysis to enforce task separation. In particular, we wish to leverage compile-time information about a program's behavior, available through the call graph and alias analysis, to make the program more secure.

Framing where statically enforced boundaries may be of use and understanding where soft boundaries fit into a greater protection scheme requires examining the strengths and weaknesses of static analysis. The design of soft boundaries should leverage the strengths of static analysis while simultaneously respecting that not all problems lend themselves to such analysis.

Section one defines the goals of soft boundaries. These goals are used in both the second section, to frame a discussion on mechanisms for specifying what data to protect, and the third section, to frame a discussion on mechanisms for specifying what control to protect. Section four ends the chapter with a discussion of the implementation of soft boundaries.

3.1 The goals and limitations of soft boundaries

The goal of soft boundaries is to assist the programmer in assuring that the use of resources does not violate a security policy specifying what accesses are allowed to particular resources belonging to the protected domain. We can check all of the possible paths in a program and mark points that appear dangerous. The programmer can later examine the apparently dangerous points and choose to make changes where appropriate. The programmer might adjust the policy to reflect a more reasonable sharing pattern or might utilize
security mechanisms already in the JVM (for instance, making a public method private) to eliminate the threat.

Notice that even though soft boundaries are intended to offer one part of an alternative to heavy-weight process style abstractions in Java, neither checking for thread capture, nor a meta-level check on the intentionality of shared data is mutually exclusive with prior boundary schemes. In previous Java separation schemes, where sharing is allowed, it is possible to both accidentally make a thread critical to the program’s functionality wait on an untrusted thread and to accidentally share data with another domain.

3.1.1 Static analysis vs. run-time enforcement

The first decision to make is whether to statically analyze the code to enforce soft boundaries, or enforce them through run-time checks. Static and run-time solutions each have advantages and disadvantages. Resource consumption attacks are a class of threats for which static analysis is poorly suited. Such attacks cannot be detected through static analysis and are therefore best handled at run-time rather than by static soft boundaries. Price et al. propose a lightweight solution to memory consumption attacks [25], while a solution to CPU consumption attacks is pending.

By contrast, static analysis is well-suited to combating thread capture attacks. Thread capture occurs when a thread invokes a method in another domain and that method does not return as expected; a server domain that calls into a client expecting the call to quickly return, but instead must wait for the client to use the thread for an expensive computation has had its thread captured. Determining, statically or at run-time, whether a call will return is undecidable (it is equivalent to the halting problem). Moreover, if the thread making the call is used by the server to perform some critical functionality (for instance, a regular maintenance task), the server may crash if that functionality is no longer performed. Even if the client domain is later terminated (forcing the call to return), the damage may have already been done. Such a problem is a property of the server code, and potentially exists for every client to the server. As a result, terminating the one client which is exploiting the vulnerability does not fix the vulnerability.
Since a thread capture vulnerability is a property of the code, static detection is no harder than run-time detection. However, if the run-time prevention of thread-capture checks at each method invocation site to ensure that either the method call is not cross-boundary or the thread is allowed to cross domains, noticeable overhead could be incurred. Alternately, a run-time solution could wait for a thread-capture attack to occur and remedy it; this would result in denial-of-service while an attack is underway. A solution based on static analysis of code has the advantage of analyzing every method invocation and determining which threads might cross boundaries and which of these threads are allowed to cross boundaries. Vulnerabilities are thus detected without run-time overhead.

What these two cases demonstrate is that statically determining precisely how a program will behave at run-time is undecidable, and that determining at run-time what might happen later in that program’s execution is infeasible. That is, static analysis is best suited to determining if a particular undesired state might be reached, while run-time checks charge an overhead to tell whether some undesired state has been reached. Because static analysis can be performed at compile-time, it puts the programmer in a better position for deciding if the program’s behavior is desirable.

### 3.2 Soft boundaries and protecting the data of a domain

In a Java program, incredible amounts of data can be shared among protection domains. Managed improperly, this can result in the kinds of security holes mentioned in Section 2.2. For instance, if a second domain receives an object from a secure domain, and breaks some invariant on that object held by the secure domain, the secure domain may enter an unstable state. Restrictions on data sharing and usage can mitigate this and similar dangers. Soft boundaries seek to limit sharing, mitigating these dangers without eliminating sharing entirely.

Limitations on sharing require first specifying which object instances are allowed to be shared. Even for small programs, the tremendous number of objects in a Java program make the naïve undertaking of such a task non-trivial. Furthermore, soft boundaries aim to avoid adding new types or significant syntax to assist in deriving the programmers in-
tent; such annotation aggravates application of soft boundaries to legacy code. This leaves analysis techniques that operate over the existing code. Unfortunately, for such techniques, including call graph generation and pointer alias analysis, completeness is undecidable, and these analyses tend to generate conservative overestimates. These results may bloat the number of false-positives, forcing the programmer to make a final decision as to whether a line of code is dangerous.

This section begins by considering how best to garner the intent of the programmer from a program, and establishing why some explicit expression of intent is necessary. It then proposes a way of explicitly expressing the intended boundary semantics of a program, including which objects are intended to be shared and which inter-domain communications are allowed.

3.2.1 Garnering intended boundary semantics

The ability to distinguish unintended sharing from intended sharing is necessary to assist the programmer in both finding errors in a program and improving the program's design; without such a distinction, the automation of forcing separation is impossible. In the static analysis-based security systems the systems discussed in Section 1.1.3, a code's intended functionality was deduced from either the code itself or from code-level annotations. Neither of these techniques, however, work for soft boundaries. To understand why that is, we must explore the issue of intent further.

Deducing programmer intent implicitly

Security analysis can sometimes deduce the programmer's intent from unannotated code. Consider the problem of buffer overflows. To cause a buffer overflow, a program must write past the last index of an array. The intent of the programmer, and the security policy, is implicit in the array declaration: an array declared of length $n$ is intended to have $n$ elements; a write past the $n^{th}$ location is a violation of that intent. In this case, the programmer's intent is simple to deduce because it is directly derivable from the syntax. Accordingly,
safe languages already prevent this violation by limiting the programmer's ability to write past the end of an array.

Unfortunately, programmer intent is more difficult to deduce in cases where there is no general difference between a line of code that is safe and one that is a security hazard. For instance, The getter/setter paradigm is an extremely common object-oriented paradigm, and public getter and setter methods appear in countless Java classes for manipulating private variables. However, only a program's designer knows for sure whether the private variable should be manipulable. Unfortunately, the programmer's intent is not obvious from the code.

In some cases, where intent is ambiguous, Engler et al. observe that patterns can be an effective guide in determining the programmer's intent [10]. However, patterns are less applicable when applied to domain separation; if the programmer did not write their program with proper separation in mind, the use of Java style sharing may have components interacting without any consistent strategy. In addition, boundaries are intended to supersede the program's expressed intent, offering a meta-check. Soft boundaries are intended to prevent bugs due to unintended component interaction, even if that interaction occurs commonly in the code. Consequently, using intent expressed in code to deduce boundary semantics could result in incorrect boundaries, rendering our subsequent analysis meaningless.

Considering the deductions that would need to be drawn, it is clear that automatically inferring the correct placement and semantics of soft boundaries is nigh impossible. Soft boundaries require an alternative mechanism for the programmer to specify the intent, one that is both simple and concise.

Expressing programmer intent explicitly

If intent cannot be deduced implicitly from the code, a mechanism for a programmer to express her intent explicitly must be designed. If the mechanism is overly complicated, then the programmer is likely to misstate her intent. Types are an example of a well-studied expression of programmer intent and have been leveraged for security purposes [12]. For example, Java's restriction on casting integers to objects provides Java's memory protection and type safety.
Figure 3.1: The format for a domain declaration consists of three fields, the domain name first, the domain’s location, and the domain API, delimited by "::". The domain’s location location refers to the location of the domain’s classes on the host computer’s filesystem. The keyword restrict is used to refer a domain to its wrapper API.

Java types provide very fine-grained semantics; each occurrence of a variable must be declared as a particular type, even if the type could be inferred from its use. Employing such a fine-grained approach in establishing soft-boundaries means labeling every sharable object. As discussed in Section 3.4.1, the amount of sharing that actually occurs would require copious declarations to encode and would therefore be overly prone to error. Inferring which object instances are safe to cross boundaries would run into the same problems as inferring the location of soft boundaries, discussed in Section 3.2.1.

3.2.2 A specification of soft boundary with data semantics

An ideal specification of boundaries is one sufficiently coarse-grained to be manageable by a programmer, yet capable of correctly describing the desired sharing semantics of the system. Such a specification would avoid the pitfalls of the fine-grained specification as well as the propensity of errors inherent in deducing intent implicitly from the code. At the expense of precision of expression, such a specification takes advantage of peoples’ tendency to be better with high-level abstractions than with the details of code.

The programmer specifies the placement of soft boundaries by declaring individual domains, and specifying the set of classes contained in each domain. The domain declarations takes the form

    domain name::filesystem location[::restrict=--domain’s API].
VITAL::\.../VitalClass::restrict WRAPPER
WRAPPER::\.../Api

(a)

class VitalClass implements API {
    private Foo x = new Foo(5);
    private Foo y = new Foo(3);
    public void private.access() {
        ...
    }
    public Foo access() {
        return x;
    }
    public void setNum(int number) {
        ...
    }
}

public interface Api {
    public Foo access();
}

(b)

Figure 3.2: The domain declaration for an example pair of classes is shown in (a) and the classes themselves are shown in (b). The domain declaration declares domain VITAL, containing only class VitalClass, to be wrapped by domain WRAPPER, containing only interface Api. Because only the access() function is in Api, it is the only method in VitalClass accessible by other domains. The object new Foo(5) is implicitly sharable because the access() method, which declared in the API, is implemented to return this object. By contrast, the object new Foo(3), which is never returned by an API method, remains private.
Similar to code sources used to specify protection domains for the purpose of stack inspection, the classes that comprise a domain are specified by location on the host computer’s filesystem. If the location specified is a directory, than, all classes contained in that directory and all subdirectories are included in the domain. As an exception, if a class file would be included in two domains, it is included in the domain with the most specific specifier. Class files are used because this is the level at which executable code is incorporated into Java, and the executable code is the entity from which domains need protection. The one exception is Java standard API classes, which are discussed below.

Figure 3.1 illustrates the definition of domains and boundaries for the Miniature Java Web Server (see Section 2.1.1). The first three declarations place each servlet and the server in their own domains, called CGI, FILE, SERVER respectively. The fourth domain, PROTECTION, specifies the Java Servlet API, which is used to interface into the other three domains.

The semantics of soft boundaries are determined by specifying an application programming interface (API) for the domain. The programmer provides a set of Java interfaces, which classes in the domain can implement. Access to a domain can occur only through the methods specified in these interfaces; that is, the safe entry points into a domain are those methods declared in an API interface and implemented in the domain. Any method intended for use only within a domain, such as one that exposes private state within the domain, would not be included in the domain’s API.

In a domain declaration, the domain’s API is specified by listing another domain which contains the interfaces for the current domain using the restrict keyword. In Figure 3.1, the PROTECTION domain is the soft boundaries API for the other three domains. Figure 3.2 shows a domain declaration (a) and the corresponding pair of classes (b). The class VitalClass implements interface Api. The domain VITAL, which contains class VitalClass, is wrapped by domain WRAPPER, which contains interface Api. Thus, access() is the only method in VitalClass that another domain can access. Furthermore, any data shared by an API method is implicitly sharable. Thus, the object assigned to the variable x is sharable.
Since the API domains are domains in their own right, a programmer may choose to include executing code in the API domain instead of or in addition to interfaces. This executing code can interact with the wrapped domain as if it was part of the wrapped domain. This ability is included primarily to accommodate legacy code in which the external API is composed of regular or abstract classes. Care must be taken in including executable code in an API, because external classes can invoke methods and access state in the API, and the API can invoke every method and access any state in the wrapped class.

There are two special domains. The first is for classes for which no domain is explicitly specified. The second is for classes that belong to the Java standard API. Classes without a specified domain are treated as if they are not wrapped by any API. Java standard API classes are special in two ways. First, every domain implicitly includes the Java standard API as a soft boundaries API. This encodes the policy that the Java standard API is implicitly trusted. Second, for the purpose of API analysis, instances of Java standard API classes declared in a domain are considered part of the declaring domain.

### 3.2.3 Critical data

In Section 3.1 we noted that even in hard boundaries, not all shared data was necessarily intended to be shared. Similarly, not all data shared across soft boundaries was necessarily intended to be shared. Furthermore, soft boundaries, by design, offer weaker protection than hard boundaries and require sufficient familiarity with the system that mistakes may occur; for instance, a secure domain may update the field of a shared object to point to a private object, not realizing that the initial object is shared with another domain; similarly, an API method may offer direct access to private data. Any separation scheme that allows sharing must accept the possibility that programmers might mistakenly share data that they intended to keep private.

Minimizing the quantity of both data shared between domains and the ways in which two domains can acquire shared data are partial solutions, but stronger assurance is needed that

- private data are never accidentally mutated by publicly accessible methods,
• pointers to private data are never returned by publicly accessible methods, and

• private data are never directly accessed by other domains.

Stated differently, the programmer requires stronger assurances that the API itself is correct. Our solution is to allow the programmer a second chance at protecting a domain’s private data by declaring that data as *critical* and then use static analysis to help the programmer guarantee its privacy. With each independent check, the programmer becomes less likely to accidentally share data. Although critical data analysis is a fine-grained expression and therefore inherits all the flaws of fine-grained schemes, it complements coarse-grained approaches very well, be they based on hard boundaries or soft boundaries.

Note that maintaining a variable’s privacy is a weaker claim than asserting that no information flow occurs [36]. In particular, we make no claims as to the confinement of state declared critical. It is possible that enough information might leak out to allow another domain to infer the value of the critical data. In addition, declaring an object as critical has no direct effect on objects the critical object may point to or otherwise influence; these objects must themselves be declared critical if they are.

**Declaring object instances critical**

Deriving the set of critical data objects from the code is difficult or impossible, particularly if one considers that the pattern of usage identifying an object as critical may be violated by every use in the code. To avoid this problem, the programmer must explicitly declare critical data as such. In this implementation, this is done by passing an object instance to the `Util.critical()` method. Alternatively, either a *critical* keyword could be used to declare variables as critical, so that any object instances that the variables referenced would becomes critical, or the declaration could occur outside of the code entirely.

**3.3 Soft boundaries and protecting the control flow of a domain**

In addition to protecting a domain’s state from outside interference, it is also necessary to protect a domain’s control flow. In this case, we are concerned not with calls into the
domain, which might modify or expose private state, but method calls out of the domain, which might trap the calling thread in another domain. This concern is presented in Section 2.2.3.

The solution to thread capture is to ensure that threads critical to the functionality of a protected domain never call into another domain. Only threads that serve no core function for the protected domain should be permitted to enter into another domain. If these threads are captured it can drain resources, but the impact is far less than if the captured thread serves a more important purpose within the domain.

The scheme has the added advantage that it reduces the number of locations that must be examined by the programmer. As discussed in Section 4.4, there are far fewer thread starting points in a domain than cross boundary edges out of the domain. The programmer only needs to analyze these thread starting points to determine if a thread that crosses a boundary should be allowed to do so, significantly reducing the effort involved in constructing soft boundaries. Such a scheme limits changes to those classes critical to the functioning of the code, classes from which passing control flow to an untrusted domain is dangerous.

3.3.1 Specifying control semantics for soft boundaries

As discussed in Section 3.2.1, statically determining which method calls will return after leaving a protected domain is undecidable; it reduces to the halting problem. Conservatively, one can assume that all threads calling across a boundary will not return and a distinction can be drawn between sacrificial threads, those that the programmer can afford to lose, and non-sacrificial threads, those that would deny service to the domain if captured.

Annotation of non-sacrificial threads as such can reduce the quantity of output that static analysis yields by pruning boundary crossings stemming from sacrificial threads. Such a labeling can occur within the domain declaration, but we choose instead to have a sacrificial thread’s class implement a Sacrificial interface which must be included with the code. Figure 3.3 demonstrates two classes declared to be sacrificial.

Static analysis can then assist the programmer in guaranteeing that a sacrificial thread never invokes a method in another domain. Conversely, if a non-sacrificial thread is cap-
public class MyThread implements Runnable, Sacrificial
public class Main implements Sacrificial

Figure 3.3: The code snippets offer two examples of the usage of the empty Sacrificial interface. MyThread is a thread that implements Runnable and contains a run() method; Main is a class containing the main() method. Both classes are declared sacrificial.

In contrast, Rudys et al. offer a mechanism for termination [31]. One special case is the Java standard library; since classes in the Java library can be assumed to be safe—they are assumed to return—even non-sacrificial threads are allowed to call into them.

3.4 Implementation of soft boundaries

The implementation of the tools to help enforce soft boundaries requires static analysis. In particular, their construction requires a call graph and a data flow graph. Rather than implementing such an analysis, this work leverages the control and data flow analysis in the Soot framework [16, 35, 27]. Soot is a powerful byte code optimization framework that runs over Java class files. Soot translates the initial Java code into four intermediate representations, each intended for a distinct purpose. The first, called Baf, is a stack based language, similar to the Java Byte code. Baf is then transformed into Jimple, a three address assembly language. The next translation takes Jimple and puts it in static single assignment form (SSA). The language, called Shimple, is quite similar to Jimple. The soft boundary checks are implemented as phases during these two stages. Last, the Jimple code is transformed into Grimp, a higher level language, and translated back into Java byte code.

Soot was developed as a research tool to investigate Java optimizations and has been successful as such [34]. SPARK, the pointer alias analysis package provided with Soot, was initially designed to compare various techniques for disambiguating pointer aliases [22]. Empirically, the resulting pointer alias and call graph are sufficiently accurate for our work;
in some cases, Soot concludes that a variable could point to every object or nearly every object, but most conclusions it draws seem reasonable.

**Closed world assumption** Java’s success and growth has historically been closely linked to the Internet; a program written in Java can be run on any platform, a pragmatic paradigm for writing Internet applications. The Internet has, in turn, put stress on the JVM’s security mechanisms. The desire to run mobile code, downloaded code that can not be trusted, has caused the JVM to evolve. Currently, the JVM allows for the construction of sandboxes to restrict untrusted code from accessing system resources [38]. A similar restriction is desirable for untrusted code interacting with trusted applications. Preventing mobile code from accessing private system resources is futile if the code can subsequently request the same resources from a class with which it interacts. The mobile code paradigm describes an open world, one in which new classes can be loaded into the virtual machine at run-time.

A down-side of the current implementation of soft boundaries is that they are more easily applied under a closed world assumption. The addition of new components to an existing system can create new control flow and data flow paths, paths that the initial analysis did not consider. Thus, when securing an open-world program, all possible paths that a new component may take must be made secure, whereas securing a closed-world program only requires securing paths used by components in the application. Since many Java programs function with a closed world assumption, we believe that focusing on a closed world is a reasonable, even though it is not always desirable. Under the closed-world assumption the results of the analysis are less work intensive for a designer to apply, making backward compatibility easier. We show in Chapter 4, however, that the results from the closed-world analysis can also be applied in an open-world, at the cost of more work required from the programmer.

**Ownership** One issue that must be discussed prior to the details of implementation is object ownership, as it determines how objects are treated and where violations of policy occur. The domain owning an object has access to all data and functionality of the object. As with the location of boundaries, ownership can be determined in several ways.
For example, when trying to bound the memory used by a component, shared ownership works [25]. Under shared ownership, all domains using an object can be treated equally and charged for the object. This makes sense in the case of memory accounting because all tasks pointing to an object are helping to keep that object alive in memory.

In soft boundaries, however, the data within a class should not necessarily be accessible by other domains. As a result, the data is considered as owned by a single domain. Since a class’s code can always access the private data of any instance of an object, every instance of the object should be in the same domain as the code. Thus, for the sake of soft boundaries, object ownership is established by the location of the class’s code, even if an instance is created in another domain. The one exception to this rule is system classes; each instance of a system class belongs to the domain where it was instantiated. This enables a domain to protect any container classes the domain uses.

This section begins by describing the cross-domain data sharing analysis and its implementation. It then presents an implementation for the API checker, the thread tracker, the critical data checker, and the thread tracker, in that order.

### 3.4.1 Implementation of cross-domain data sharing analysis

As hypothesized in Section 3.2.1, a fine-grained mechanism for specifying soft boundaries would yield poor results. Validation of this hypothesis requires quantifying the interactions between domains. Static analysis can aid this process by examining all possible paths and returning a conservative set of domain interactions. There are three sets of particular importance:

- The *information flow set* (INFO-FLOW) records dereferences to an object instance by two domains. The analysis uses Soot’s pointer aliasing analysis engine to identify objects at the granularity of allocation points. When our analysis determines that an object allocated at a particular location is dereferenced in two domains, if the data was not explicitly passed from one domain to the other, every dereference of that object in a domain that is not the object’s owner is added to the INFO-FLOW set.
• The \textit{time of check to time of use set} (TOCTOU) records updates to object instances by one domain and dereferenced by another. Again, Soot's pointer aliasing analysis engine is used, identifying objects at the granularity of allocation point. When our analysis determines that an object allocated at a particular location is dereferenced in one domain, and modified in another, every dereference and modification points is added to the TOCTOU set. These uses meet one requirement for a potential time of check to time of use bug, and the set is accordingly referred to as the TOCTOU set. An element in the TOCTOU set represents a line of code that may be affected by an update to an object instance.

• The \textit{public function mutation set} (FMUTATE) set is composed of locations in the code where an object instance in a domain could be modified as a result of a method call from another domain. In terms of potential damage there is no difference between an undesired, direct update and one through a public assignment method.

Figure 3.4 gives a graphical illustration of how an object enters the three interaction sets. Objects 1 through 6 in this example are owned by domain 1.

The construction of the INFO-FLOW set and the TOCTOU set rely on both the set of objects dereferenced by a domain and the set of objects mutated by a domain. For a given domain, taking the union of sets of every line's dereferences yields all object instances that the domain dereferences. Similarly, for a given domain, taking the union of sets of every line's mutations yields all object instances that the domain mutates. Due to the conservative nature of alias analysis, these sets are not complete, so neither is the analysis.

\textbf{Constructing the INFO-FLOW set} The INFO-FLOW set is constructed by first taking the union over all pairs of domains of the intersection of the sets of objects each domain dereferences. If any object in the intersection is explicitly passed from one of the domains to the other, the object is removed from the intersection. Since the set of such objects is not complete, the subtraction makes the INFO-FLOW set unsound. The union of this intersection over all pairs of domains is the set of all objects that implicitly cross some
Figure 3.4: Illustration of domain interaction sets. Blocks on either side represent two domains. Each arrow represents an operation in the domain of the arrow’s origin and is labeled with the operation that the arrow is performing. At the base of the arrow is the set in which the line of code from which the arrow originates would be a member. Domain 1 also contains an example of an element in the FMUTATE set, a public function in Object 5 mutates one of the object’s private fields and Domain 2 has a reference to Object 5.
boundary. The INFO-FLOW set is the set of all dereferences to such an object that do not occur in the domain that owns the object.

**Constructing the TOCTOU set**  The TOCTOU set is constructed by taking the set of all objects that implicitly cross some boundary, as defined above. If any of these objects are modified (that is, if a `setfield` or array store bytecode is called on one of these objects) outside of a constructor, then the potential exists for a TOCTOU. The TOCTOU set is the set of all dereferences or mutations of any such object. These are the locations where a TOCTOU bug might occur.

**Constructing the FMUTATE set**  The FMUTATE set for a domain is constructed by first building a set of extended entry points. The extended entry points are all entry methods in a class that contains a defined entry point. FMUTATE is the set of all modifications reachable from an extended entry point.

A few reasonable optimizations to the construction can significantly shrink the size of these sets. When constructing the sets of data dereferenced and mutated by a domain, it is important to ignore updates made in the constructor or updates only reachable from the constructor; otherwise, every cross domain instantiation would fall into the set. Also, when constructing the FMUTATE set, any modifications where the modification immediately follows the allocation of the object are assumed to be intentional and hence ignored. The results of this analysis are presented and further analyzed in 4.1.

One final note on construction of these sets: Soot's alias analysis package uses several kinds of nodes to represent memory locations, static fields, and object instances. However, the numbers gathered for the sharing analysis only reflect the nodes of type `Alloc`, those that represent objects for which Soot could identify the instantiation point of that object [22]. This prunes nodes that should not affect the analysis (for instance, objects instantiated implicitly by the JVM).
3.4.2 Implementation of API checker

We built the API checker to verify that a program follows the specified API (see Section 3.2.1 for a description of soft boundary APIs). The checker returns two classes of diagnostics. Warnings are returned when the API abstraction is broken. This occurs if a class which implements a public API also implements non-API public methods. It also occurs if an external class invokes an API method directly on the class instance. In these situations, although the API has not actually been violated, the underlying implementation is exposed to an API violation.

The second class of diagnostics, errors, are returned when an API has actually been violated. These occur when a non-API public method has been invoked from outside the domain or instance fields have been accessed or manipulated. In these situations, some outside class is completely circumventing the API and accessing the wrapped domain in an unintended manner.

For complete domain separation, both errors and warnings should be fixed; there is no reason to have public methods in a domain and not have those methods in the API. Figure 3.5 gives an example of errors and warnings in two interacting Java classes. The External class uses the Internal class which is wrapped by interface API.

Errors Errors involving invocations of methods not in the API are found by examining the call graph for all calls into a protected domain. It is then a simple process to both compare those calls to the methods in the API and have Soot return a line number for each offending invocation. The labeling of any direct data accesses within a protected domain is accomplished in a similar way: every field access in the program is scanned, and any accesses that violate an API are flagged.

Warnings Finding public methods not in the API involves comparing all public methods in the domain against methods declared in the API; any public methods not appearing in the API are flagged. Finally, finding methods invoked directly instead of through an interface requires checking the type of the receiver at the time of invocation. Note that this final case
Figure 3.5: A code example how the API checker would treat different cross-domain behavior. (a) shows the domain separation of the classes. (b) shows the classes themselves. Classes `Internal` and `External` have been declared in different domains, with `Internal` wrapped by the API `API`. The comments describe the errors and warnings occurring in each line of code.
is unique in that there is nothing problematic about the call itself; the issue is with the way the call is made.

**Interaction with the API** In some cases, it is desirable to treat an API as a separate domain for the purpose of analysis. One case where we chose to do this was for the Java Servlet API. In this case, the servlet container uses the Servlet API to communicate with the servlets, and the servlets use the API to communicate with the server; it is not appropriate to attach the Servlet API to any particular domain.

By contrast, in the case of the Pastry system, the API is only used to wrap the Pastry core classes. The API also implements some code that interacts with the Pastry core. In this case, it is appropriate to treat the API and the Pastry core domain as one large domain; otherwise, the amount of inter-domain interaction would be artificially high. As a result, we provide an option for specifying whether or not a domain’s API wrapper should be treated as part of the domain.

### 3.4.3 Implementation of critical data checker

We built the critical data checker to verify that a domain’s private data stays private. We wish to provide an additional line of defense, preventing access to private data even in the event that an API method allows such access. Section 3.2.3 explains why such a mechanism is useful.

Our critical data analysis is similar to building the INFO-FLOW, TOCTOU, and FMUTATE sets only over values declared critical and without subtracting the sets of data explicitly passed by methods. The construction makes the critical data analysis sound, but not complete. In addition, if any method passes critical data across a boundary either as a parameter to or return from a method call, this is flagged. The result is a complete description of the cross-boundary usage pattern of critical data. It is now up to the programmer to confirm that any instances of cross-boundary data usage are intended.
Figure 3.6: The circles represent methods, the surrounding rectangle represent classes, and the encompassing rectangle represents a domain. One method invokes a method in another domain and any methods that are consequently in the CROSS-INVOCATION set are represented with double circles.

3.4.4 Implementation of thread tracker

Finally, we built a thread tracker to check which threads cross a boundary. The thread tracker must find all threads that directly or indirectly pass control flow across a boundary. Thread tracking is motivated in Section 3.3.

Identifying what threads may cross a boundary is accomplished by first building the set of methods that pass control flow across a boundary, either directly or indirectly. For ease of reference, call this set of methods the CROSS-INVOCATION set. Given the CROSS-INVOCATION set, we examine the threads that can invoke methods in the set. Construction of the CROSS-INVOCATION set for a domain requires two steps:

1. Construct set $S'$, calls from the protected domain into a method in another domain, a set available from the call graph. CROSS-INVOCATION is initialized to the set of methods in which the invocations in $S'$ occurred.

2. Using the call graph, the thread checker determines all methods that invoke a method already in the set CROSS-INVOCATION and adds the newly found methods to the set. This step iterates until a fixed-point is found; since there are only finitely many methods in a program, it will eventually reach a fixed-point.
Figure 3.6 gives a graphic illustration of the methods that get placed in the CROSS-INVOCATION set for a single edge leaving a domain. As the diagram shows, this set is the set of all methods whose transitive closures include a control flow edge leaving the domain.

If a thread can pass control flow across a boundary then the method that starts the thread must be in CROSS-INVOCATION, since all methods that directly or indirectly pass their control flow across a boundary are in CROSS-INVOCATION. A security hazard is any non-sacrificial thread with a method in CROSS-INVOCATION. For example, the main thread of a Pastry node is critical to the functionality of the node, but there exist paths in the call graph from the main() method into the application, a security hazard. The thread tracker returns to the user the threads that pass their control across a boundary and the points at which analysis reveals control crosses. The results of the thread tracking tool are presented in Section 4.4.
Chapter 4

Results

Prototypes of the tools described in Section 3.4.1 to assist in building soft boundaries were used to examine the Miniature Java Web Server, Pastry, and Sandstorm. The Miniature Java Web Server is partitioned into three domains, one for each servlet and one for the server. Pastry and Sandstorm are each divided into two domains because, in both cases, there is a single application running on a trusted and critical backbone. Table 4.1 gives statistics on the programs. The table shows the programs to be large enough that we can expect them to exhibit interesting data-sharing behavior.

This chapter begins by presenting the results from the shared data analysis, giving a better feel for cross-boundary sharing. The results from the API checker, the critical data checker, and the thread checker are then discussed in the context of the ease with which each application could be restructured to meet the requirements imposed by the tools. The results are grouped and presented according to the tool that produced them.

4.1 Results from domain interaction analysis

The domain interaction analysis was run to give a quantitative measure of the amount of interaction among domains. The analysis attempts to give a better understanding of the trade-offs between a fine-grained specification and a high-level specification. The results are presented in Tables 4.2 and 4.3. Table 4.2 shows the results for the TOCTOU set and INFO-FLOW set analysis. The table shows that 8% to 15% of analyzed code lines require analysis. That is to say, getting a handle on the cross-domain sharing of these systems would require examining well over 1,000 statements.

The results of the FMUTATE set further support the results of the INFO-FLOW sets and TOCTOU sets. As many as 7% of the total lines are would require analysis for this
<table>
<thead>
<tr>
<th></th>
<th>Classes</th>
<th>Lines of Code</th>
<th>Public Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Analyzed</td>
<td>Total</td>
</tr>
<tr>
<td>Web Server</td>
<td>22</td>
<td>21</td>
<td>5,392</td>
</tr>
<tr>
<td>Pastry</td>
<td>493</td>
<td>240</td>
<td>59,405</td>
</tr>
<tr>
<td>Sandstorm</td>
<td>334</td>
<td>190</td>
<td>39,948</td>
</tr>
</tbody>
</table>

Table 4.1: This table holds the number of classes and lines of code for each of the applications. The total numbers show the statistics for the entire application package. The analyzed numbers show the statistics for the classes that were actually analyzed by Soot. Soot only analyzed classes that the application used.

<table>
<thead>
<tr>
<th></th>
<th>INFO-FLOW</th>
<th>TOCTOU</th>
<th>INFO-FLOW ∪ TOCTOU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Instances</td>
<td>% lines</td>
<td>Instances</td>
</tr>
<tr>
<td>Web Server</td>
<td>49</td>
<td>5.47%</td>
<td>34</td>
</tr>
<tr>
<td>Pastry</td>
<td>255</td>
<td>1.96%</td>
<td>91</td>
</tr>
<tr>
<td>Sandstorm</td>
<td>256</td>
<td>2.75%</td>
<td>153</td>
</tr>
</tbody>
</table>

Table 4.2: The two Instance columns represent the number of unique instances contributing to the INFO-FLOW and TOCTOU sets respectively. The % lines columns are a ratio of the number of lines of code requiring inspection to the number of lines of code analyzed. The final column, code to union size, is a ratio of the number of unique lines requiring inspection to the number of lines analyzed.

sort of interaction. The total number of unique lines requiring analysis is between 9% and 17% of the number of lines analyzed; this ranges from nearly one thousand to nearly three thousand lines. These results are shown in Table 4.3.

The analysis demonstrates that it is possible to explicitly label every point of interaction between domains, but doing so is burdensome and error prone. In larger programs, such as Pastry and Sandstorm, there are over 250 unique objects and approximately 10% of lines are involved in cross-domain interactions, numbers that do not even take into account the FMUTATE set. A large amount of annotation does not preclude the possibility of a fine-grained specification, but it does imply that alone, such a specification will be complex enough that it will likely miss many bugs. These results do not preclude the possibility of
public class CgiServlet extends HttpServlet {
  private void serveFile( HttpServletRequest req,
                           HttpServletResponse res, String path ) ...{
    ...
    envVec.addElement( makeEnv(SERVER_SOFTWARE,
                                 getServletContext().getServerInfo() ) );
    ...
  }
}

public class Serve implements ...{
  public static void main( String[] args ){
    ...
    serve = new Serve( arguments, printstream );
    ...
    serve.setMappingTable(mappingtable);
    serve.setRealms(realms);
    ...
  }
}

(a)

public class CgiServlet extends HttpServlet {
  private void serveFile( HttpServletRequest req,
                           HttpServletResponse res, String path ) ...{
    ...
    String queryString = req.getQueryString();
    int contentLength = req.getContentLength();
    ...
  }
}

(b)

Figure 4.1: (a) shows a dangerous entry from the INFO-FLOW and TOCTOU sets whereas (b) shows an innocuous one. Both examples are from the Java Miniature Web Server code base.
Table 4.3: Entry points are the public methods reachable from an object pointer held by another domain that eventually mutate some local state. Code to set ratio is the ratio of the number of lines that the FMUTATE set signals as requiring examination to the number of analyzed lines of code. The final column is the number of lines requiring examination for all three lines of code to the number of analyzed lines.

<table>
<thead>
<tr>
<th></th>
<th>FMUTATE</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Entry points for modifications</td>
<td>% lines</td>
</tr>
<tr>
<td>Web Server</td>
<td>49</td>
<td>5.69%</td>
</tr>
<tr>
<td>Pastry</td>
<td>70</td>
<td>0.47%</td>
</tr>
<tr>
<td>Sandstorm</td>
<td>88</td>
<td>1.17%</td>
</tr>
</tbody>
</table>

a fine-grained specification; they do, however, imply that alone a fine-grained specification insufficient.

Examining two examples of actual set elements reveals a further difficulty: taken out of context, some dangerous sharing may appear safe to the programmer. Figure 4.1 gives two examples of entries in the INFO-FLOW/TOCTOU sets. The first is a bug stemming from a subtle design flaw, and the second is safe.

In Figure 4.1(a), the server doubles as the servlet’s context, and is consequently intended to be shared with the servlets. Even though it appears that passing the server to the servlet is intentional, Section 4.3 demonstrates that the server gives the servlets access to internal data from which the servlets should be restricted. This separation bug is unlikely to be found by labeling crossing points as allowed or disallowed since it is not an unintentional sharing, but rather a flaw in the design of the server; a programmer labeling the code would label passing the server as allowed. In fact, even labeling object instances as sharable or unsharable would unlikely catch the problem. The design of the code would require the server to be declared sharable and the server’s internal structures could then be mutated by public setter methods even if the private structures were labeled unsharable and never shared. If, however, one of the server’s internal structures were to be accessed and the programmer were made aware of the access, the oversight would become obvious and the server could be redesigned to correct the flaw.
<table>
<thead>
<tr>
<th></th>
<th>Errors</th>
<th></th>
<th>Warnings</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data access</td>
<td>Non-API Invokes</td>
<td>Non-API methods</td>
<td>Direct invokes</td>
</tr>
<tr>
<td>Web Server</td>
<td>0</td>
<td>8</td>
<td>60</td>
<td>7</td>
</tr>
<tr>
<td>Pastry</td>
<td>0</td>
<td>0</td>
<td>150</td>
<td>0</td>
</tr>
<tr>
<td>Sandstorm</td>
<td>5</td>
<td>61</td>
<td>152</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4.4: This table shows the number of API errors and warnings generated by the API checker the Miniature Java Web Server, Pastry, and Sandstorm. *Data access* refers to direct access to the fields of a wrapped domain from outside. *Non-API invokes* refers to invocations of non-API methods from outside the domain. Non-API methods refers to the declaration of public non-API methods in an exposed class. Finally, *Direct invokes* refers to the number of invocations of API methods directly to the class.

For the web server, the data is only for the servlet container itself, even though the servlets were also wrapped by an API.

The difficulty with handling the interaction sets is that not all code lines that would require examination are bugs. In Figure 4.1(b), the CgiServlet has received a request and is processing it. When a request is made to the servlet, the details of the request are specified in an object of type HttpServletRequest. Thus, the req object is not only safe to dereference, the functionality of both the server and the servlets depends on the dereference occurring.

### 4.2 Results from API checking

We ran our API checker, described in Section 3.4.2, over our three test programs to evaluate how closely they follow their API. Table 4.4 reports errors and warnings in each program. For a more robust partition, the warnings could also be examined, but doing so is unnecessary. The results support the feasibility of establishing soft boundaries in real world software.

#### 4.2.1 Miniature Java Web Server

The server’s domain in the Miniature Java Web Server has eight API errors. All eight errors involve the servlets invoking methods in the server’s domain. One likely culprit is that the
Java servlet API is a subset of the API which the programmer imagined; some of the classes that make up the server are clearly intended to be used directly by the servlets even though the interaction is not specified in the Java servlet API. Upon closer examination, all eight method invocations appear to be safe to include in the API.

The web server had so few errors because the code base is small and, although the web server did not specify its API, Java provides the Servlet API which addresses most server-servlet inter-action needs. The specification provides a natural boundary to even poorly-designed code. However, as we shall see in Section 4.3, security hazards can arise, even in the presence of an API, if a programmer does not realize the danger in allowing access to private data or accidentally allows such access.

The low number of errors are in contrast to the heavy domain interaction, showing that although the servlet container and servlets have a high level of interaction, most of it follows the servlet API and is therefore presumably safe. Furthermore, the results make sense since much of the interaction between a servlet container and the servlets should be initiated by the server. The server domain calls into the servlets’ domain and passes a request object containing the data necessary for the request to be processed.

4.2.2 Pastry

We defined Pastry’s API as the commonapi directory in the Pastry distribution. Pastry had no API violations, but there were numerous API warnings. Pastry has 150 public methods not in the API. Of these public methods, most are harmless, but there are some that could cause problems if invoked, an issue explored further in Section 4.3.

It may appear troublesome that some of these public methods are a potential security hazard, and the API checker only raised warnings, but the results reflect the closed-world assumption discussed in Section 3.4. If any of these methods were actually invoked by the application, the invocation would have triggered an API violation, however, since errors returned by the API checker represent actual security hazards rather than potential security hazards, none of the methods were marked. As long as any application added to the world has the API checker run over it, the public methods are safe and invocations of methods not in the API can be caught when they occur.
public void handleEvent(QueueElementIF item) {
    ...
    BufferElement payload = resp.getPayload();
    byte paydata[] = payload.data;
    ...
}

Figure 4.2: Haboob violating the Seda API.

4.2.3 Sandstorm

Sandstorm's API was taken to be the api directory included in the Sandstorm distribution. Sandstorm, the Seda implementation, had thirteen invocations of methods not in the API and four direct data accesses. Haboob was the only application to directly access data. Figure 4.2 shows an example of one such access. In the code, class BottleNeck is violating the Seda API by choosing to bypass the existing public getter for the data field; the BufferElement class is in Seda's domain.

The accesses turn out to not be security concerns, but are still poor programming practice. Later versions Sandstorm may alter the treatment of data introducing subtle bugs that could have been avoided had Haboob been written to access Sandstorm through the proper channels.

4.2.4 Challenging the closed world assumption with API checking

Until this point it has been assumed that the applications are running in a closed world, and the focus has been on errors in the program. But the results can also aid in providing security in an open world. In an open world, however, the warnings become important, perhaps more so than the errors.

To provide separation between domains, each domain must be placed in a distinct package. It is then necessary to remove all public fields from the protected domain. Removing public fields hinders expressibility, but prevents direct field access, which is a violation of the API. Last, the access privilege of any method on which a warning was flagged must be
changed; only methods in the API are public. Restructuring the program and its packages to meet these requirements allows the Java protection mechanisms to provide separation.

The methods flagged with warnings are a subset of the methods requiring programmer examination, furthermore, the programmer must also change the access specifier on all public fields in the domain, making the process is more involved than its equivalent in a closed world.

4.3 Critical data analysis

We then ran the critical data checker, described in Section 3.4.3, over the Miniature Java Web Server and Pastry. In both cases, data that should not be modified outside the core classes was marked critical and, in both cases, the results yielded potential security concerns.

Sandstorm is a code library to save the programmer from having to worry about low-level event details. Thus, due to the nature of Sandstorm, almost all structures are safe for the application to directly or indirectly influence. Such is intended functionality, making critical data difficult to identify if it even exists. As a result, we did not perform critical data analysis over Sandstorm.

4.3.1 Miniature Java Web Server

Examining the code for the web server revealed that much of the critical data was stored as fields in the server. Using the server to store private data may be safe since the server-servlet abstraction should allow the servlet to function without knowing that the server exists and, more importantly, without getting a pointer to the server. We marked the server as critical, and critical data analysis revealed that the server object was being passed to the servlets as the servlet context. Figure 4.3 shows a line that the critical checker labeled as dereferencing the server. The method getServletContext() is inherited from the servlet's superclass, HttpServlet. In the parent class, getServletContext() calls the getServletContext() method in the ServletConfig interface. Figure 4.4 shows that the servlet's ServletConfig is instantiated with the server as its
public class CgiServlet extends HttpServlet

... private void serveFile(...){
...
    envVec.addElement(makeEnv("SERVER_SOFTWARE"
        servletContext().getServerInfo() )
...
}
...

Figure 4.3: In this code example the `servletContext()` actually returns the server, which is then dereferenced.

context, meaning that the server is in fact accessible by the servlets. This method is part of the API which would seem to imply that the API is broken. This case is interesting because the API is the Java Servlet API and consequently we know it is correct. The problem is actually a program design issue. The programmer assumed it was safe to pass the server to the servlets, which is only true if the servlets are completely trusted and free of bugs.

Although a stronger separation would deny the servlets access to the server altogether, it is still possible that passing the server to the servlets is safe because the server does not give access to private internal state. The second experiment marked the `mappingTable` and the `realmTable` critical. The former maps paths to servlets and the latter maps users to passwords, when passwords are required. Both of these structures should not be accessible to a malicious servlet, but critical analysis reveals 7 modifications to them through 4 entry methods. Of these four entry points, 2 are setters for the mapping and realms fields, and one is the `destroyServlets()` command. The remaining method is the `serve()` method, which starts the server. All four methods are security hazards, but since none are in the API, the API is correct.

The simplest solution is to prevent the servlets from obtaining a pointer to the server; do not use the server as the servlet’s context.
public class Serve implements ServletContext, RequestDispatcher
...
private void addServlet(...){
...
    servlet.init( new ServeConfig( (ServletContext) this,
             initParams, urlPat ));
    registry.put( urlPat, servlet );
...
}
}

Figure 4.4: The serve class is the server. Notice that it implements ServerContext. The
method shown here is the method used by the server to add new servlets. ServeConfig
implements ServletConfig. Thus the ServletConfig for all servlets is the new
ServeConfig with this set as its context.

4.3.2 Pastry

In Pastry, the node holds data critical to the functioning of the pastry network. This data
includes the node’s ID and the routing table. If an application could change its node’s ID
then a malicious application could rapidly change the ID, causing surrounding nodes to
interpret each change as a new node appearing on the network. Each surrounding node
would consequently delete a subset of their keys and each change would propagate down
the network.

The node’s ID was declared critical and the checker was run over the code. Although
the application does not explicitly change the node’s ID, three of the eight public functions
that the tool identified as dangerous allow the application to change the ID in a dangerous
manner: setdigit(...), xor(...), and setbit(...). Even though the informa-
tion critical to the node ID is declared private, any of these functions could be used to
perpetrate the attack just mentioned, however, none of these methods are in the API so the
API is correct.
4.3.3 Challenging the closed world assumption with critical data checking

In Section 3.2.3, critical data checking was said to give the program designer a mechanism to check that the given API is correct, but the critical data checker can serve a second purpose and help make the transition from a closed world to an open one. Using the critical data analysis to this end would require extending the implementation to examine all lines in a domain, rather than just those involved in Soot's analysis.

Returning to the sample applications, as discussed in Section 4.2, the API Checker reports that the Java Miniature Web Server has 60 public methods not in the API and Pastry has 150 public methods not in the API. Restructuring the program so that none of the methods are public could be a burdensome task, and the programmer may wish to only change those methods that absolutely require changing. The results of the critical data analysis can be used to aid in determining if a public method not in the API can be added to the API. In Pastry, for example, critical analysis reveals that \texttt{setdigit(...)}, \texttt{xor(...)}, and \texttt{setbit(...)} are dangerous. Consequently, a programmer could not add any of those methods to the API. In the case of these three methods, the solution is simply that the methods should not be public, and as they are not intended for external use anyway, such a change should be easy to implement. In other cases, the solution may require changing the package structure of the application.

Using the critical analysis to determine which methods can safely be added to the API is not without risk. Critical analysis uses a fine-grained expression and, as Section 4.1 concludes, fine-grained expressions applied to soft-boundaries are error prone. It is therefore imperative that adding methods to the API based on the results of critical analysis results is done conservatively and with care.

4.4 Thread tracker results

Finally, we ran the thread tracker, described in Section 3.4.4, over our three test programs: Miniature Java Web Server, Pastry, and Seda. The results support the earlier assertion that aggregating thread capture analysis by thread significantly simplifies analysis 3.3.1.
<table>
<thead>
<tr>
<th></th>
<th>Web Server</th>
<th>Pastry</th>
<th>Seda</th>
</tr>
</thead>
<tbody>
<tr>
<td># Edges crossing boundaries</td>
<td>79</td>
<td>24</td>
<td>55</td>
</tr>
<tr>
<td># Unique starting classes</td>
<td>5</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4.5: The table gives the results of applying the thread tracker to the Miniature Java Web Server.

### 4.4.1 Miniature Java Web Server

When run over the Miniature Java Web Server, the thread tracker returned that nearly all boundary edges trace back to a class called ServeConnection, which starts a new thread for each servlet. In addition, a few boundary edges traced back to anonymous inner classes. The only questionable control flow decision made is illustrated in 4.5. Additional servlets, programmed by the user, are all instantiated using the same thread; if one servlet’s constructor does not return, it could cause a DOS. But such an error should be caught when the server is first started and, consequently, cause minimal damage. Were the server to be extended to dynamically load servlets, however, each servlet would have to be constructed in its own thread.

### 4.4.2 Pastry

Of the four classes in which threads start, two were dangerous. Figure 4.6 displays the code for three dangerous function calls that occur in the Pastry code base. In (a), Pastry has received a message and is enquiring with the application whether to forward it. In (b), Pastry has received a message for the current application and is delivering it. And in (e), Pastry is informing the application that the network’s leaf set has changed. These functions are defined in `rice.P2P.commonAPI.Application` as the functions through which the Pastry node communicates with an API so the invocations are necessary; the problem is that they occur in the main thread. One solution to this problem is for each node to keep a thread pool and use a separate thread for each communication between node and application.
public class Serve implements...
    public static void main( String[] args ){
        ...
        new Thread(new Runnable() {
            public void run(){
                serve.readServlets(servFile);
            }
        }).start();
        ...
    private void readServlets(File servFile) {
        ...
        while (se.hasMoreElements()){
            servletname = (String)se.nextElement();
            addServlet(servletname, (String)servletstbl.get(servletname),
                       (Hashtable)parameterstbl.get(servletname));
        }
        ...
    }
    ...

Figure 4.5: The Serve class uses a new thread to call readServlets which determines which adds all servlets besides the default two. If one servlet’s constructor does not return then the loop can add no more servlets.
public class PastryEndpoint ... {
    public final boolean enrouteMessage(...) {
        ...
        return application.forward((RouteMessage) msg);
        ...
    }
}

(a)

public void leafSetChange(...) {
    ...
    application.update(nh, wasAdded);
    ...
}

(b)

public void leafSetChange(...) {
    ...
    application.deliver(rm.getTarget(), pMsg.getMessage());
    ...
}

(c)

Figure 4.6: The three examples each show a line in the Pastry code base where Pastry calls into the application. The methods forward, update, and deliver are the functions through which Pastry talks with its applications but, under the current implementation, communication is insecure.
4.4.3 Seda

The thread tracker collapses 55 crossings into 10 starting classes. In the worst case, to separate Sandstorm from Haboob requires examining only 10 classes. But Seda is interesting in that none of these interactions are dangerous because the server application running over Seda is trusted. In fact, starting any unnecessary threads to try and increase separation would counter productively slow an efficiency critical system. In this case, any further control flow separation is undesirable.

4.4.4 Challenging the closed world assumption with thread tracking

The results of thread tracking are not dependent on the closed world assumption and apply just as easily in an open world. The threats revealed to both Pastry and the web server also apply in an open world.

Because Soot analyzes only those code lines reachable from the program's staring point, however, the results presented here are a subset of the results necessary to secure the programs in an open-world; a dynamically loaded class could cause the program to reach a line that it did not previously. To extend the analysis to the open-world would require examining all paths from a public method or thread starting point in the domain.
Chapter 5

Conclusion

The designers of multi-user computer systems have long recognized a need for protection domains to provide separation between multiple tasks running on a single system, and the increasing deployment of mission critical software, such as web servers, in language based systems has in turn created a similar need in language based systems. This separation provides security for distrustful tasks and can prevent a failure in one task from causing failures in other tasks. This work focuses on providing separation in Java applications. Most existing mechanisms for providing separation semantics in Java use a hard boundary model based on a process abstraction and enforce separation at run-time. They accept run-time overhead and changes to the Java sharing semantics as an acceptable cost. But they also break Java's free sharing paradigm. This thesis proposes soft boundaries, a language-based approach to separation, that does not change the Java sharing semantics and is enforced by static analysis.

This thesis presented three static analysis tools used to enforce soft boundaries. The first, the API checker, confirms that the only way one domain interacts with another is through a method specified in the domain's interface. The second, the thread checker, verifies that a protection domain has knowledge of every thread that might pass its thread of control to another protection domain. Finally, the critical data checker checks both that certain pieces of state marked as critical by the programmer are never passed to another protection domain and that no other protection domain can influence the value of the critical data. Although the work focuses on whole program analysis, it is also shown how the results can be applied to separate components in an open world.

These tools, when run over three different programs, revealed a number of interesting inter-domain interactions. Many of the interactions found would become serious security concerns in an adversarial environment, and Pastry's overuse of its main thread could cause
problems even in the current implementation. In addition, the number of locations noted as requiring either fixes or further annotation is manageable, even though there were a number of false-positives. Unfortunately, because the intentions of the original coders was not always clear even to a human analyst, getting an accurate percentage of false positives is not possible.

These results demonstrate that static analysis is a useful tool for supporting protection domain separation. These tools can complement or supplant many run-time boundary enforcement techniques currently in use. They are useful for analyzing legacy code as well as in the design of new systems. These tools, combined with earlier language-based security mechanisms, provide many of the protection domain semantics available in traditional operating systems.
Bibliography


