Language and Framework Support for Reviewably-Secure Software Systems

by

Adrian Matthew Mettler

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division
of the
University of California, Berkeley

Committee in charge:
Professor David Wagner, Chair
Professor Deirdre Mulligan
Professor Dawn Song

Fall 2012

Language and Framework Support for Reviewably-Secure Software Systems

Abstract

Language and Framework Support for Reviewably-Secure Software Systems

by

Adrian Matthew Mettler
Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David Wagner, Chair

My thesis is that languages and frameworks can and should be designed to make it easier for programmers to write reviewably secure systems. A system is reviewably secure if its security is easy for an experienced programmer to verify, given access to the source code. A security reviewer should be able, with a reasonable amount of effort, to gain confidence that such a system meets its stated security goals. This dissertation includes work on on language subsetting and web application framework design.

It presents Joe-E, a subset of the Java programming language designed to enforce object-capability security, simplifying the task of verifying a variety of security properties by enabling sound, local reasoning. Joe-E also enforces determinism-by-default, which permits functionally-pure methods to be identified by their signature. Functional purity is a useful property that can greatly simplify the task of correctly implementing and reasoning about application code. A number of applications of the Joe-E language are presented and evaluated.

The second part of this dissertation presents tool and framework enhancements for improving the security of web applications. I present techniques for retrofitting existing web applications to use template systems effectively to prevent cross-site scripting and content injection vulnerabilities while preserving functionality. I also show how HTML templates can be rewritten to normalize their output, improving the assurance of security provided by automatic escaping and other static analyses.

These two applications of my thesis demonstrate that practical enhancements to languages and frameworks can support developers in creating more secure software that is easier to review. Continued improvement in language and framework support for reviewability is a promising approach toward improving the security provided by modern software.

To everyone who has believed in me, especially my family.

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I thank my advisor, David Wagner, for his enthusiasm, encouragement, and sage advice. It has been a pleasure and a privilege working with such a dedicated, perceptive, and supportive mentor. I would also like to thank thesis committee members Dawn Song and Deirdre Mulligan, as well as qualifying exam committe member Koushik Sen, for their helpful comments and suggestions.

Mark Miller, Marc Stiegler, and Tyler Close provided inspiration and invaluable advice in the philosophy and design of Joe-E, and I owe them a debt of gratitude.

I wish to give special thanks to the coauthors of my thesis research: Naveen Sastry, Matt Finifter, Tyler Close, Akshay Krishnamurthy, and David Wagner. This dissertation would not be what it is today without their contributions.

I have had the opportunity to work with several talented undergraduates over the course of my graduate studies, including Matt Finifter, Kanav Arora, Akshay Krishnamurthy, Anne Edmundson, Brian Holtkamp, and Emanuel Rivera. I would like to thank all of them for their hard work and active engagement.

Several chapters in this dissertation have benefited from the generous efforts of colleages providing feedback on earlier drafts. I thank David Molnar, Karl Chen, Arel Cordero, Tyler Close, Toby Murray, Sandro Magi, Mike Samuel, Erika Chin, Cynthia Sturton, Adrienne Felt, Devdatta Akhawe, Matt Finifter, Prateek Saxena, Blaine Nelson, and others I have unfortunately forgotten for providing feedback on papers and presentations in progress. I also thank shepherds Sriram Rajamani, Úlfar Erlingsson, and Todd Millstein, and anonymous conference reviewers for their helpful feedback on submitted papers.

I would like to thank the Siebel Scholars Foundation for the fellowship support as well as their informative, stimulating conferences and other events. I also thank the National Science Foundation for research grants that have supported my graduate studies.

La Shana Porlaris, Christopher Hunn, Angie Abbatecola, and Willa Walker have done a great job helping me navigate the Berkeley bureaucracy, and I want to thank them for making the experience as painless as possible.

I would like to thank the CSGSA folks, the poker night gang, and others including Devdatta Akhawe, Norm Aleks, Dominic Antonelli, Marco Barreno, Peter and Zuska Bodik, Karl Chen, Kuang Chen, Erika Chin, Arel Cordero, Alex Fabrikant, Drew Fisher, Lisa Fowler, Jon Kuroda, Todd Kosloff, Robert Letzler, Prateek Mittal, Blaine Nelson, Matt Piotrowski, Mariana Raykova, Charles Reiss, Barret Rhoden, Justin Samuel, Jeremy Schiff,

Rusty and Dara Sears, Justine Sherry, Alex Simma, Isabelle Stanton, Mao Taketani, David Zhu, and probably a lot more I've accidentally left out, for their conversation and comradery during my time here at Berkeley. It's been great getting to know and spend time with all of you.

Finally, I thank my family for their support, especially my mother, without whose gentle nudging I would likey have taken even longer to finish.

Chapter 1

Introduction

My thesis is that languages and frameworks can and should be designed to make it easier for programmers to write *reviewably secure* systems. A system is reviewably secure if it meets its security goals in a way that facilitates security review. A security reviewer should be able, with a reasonable amount of effort, to gain confidence that such a system meets its stated security goals.

This goal can be accomplished in two main ways. The first is through the use of abstractions that remove hazards that can lead to insecure software. The second is by enabling and encouraging implementation patterns that facilitate reasoning about the security properties of the resulting system. This approach allows programmers to identify security properties and write their code to ensure these properties in a way that can be confidently verified in a security code review.

1.1 Background

Central to this dissertation is the ability to improve application security through the building of applications that support security review. A major way to do this is the creation and use of safe abstractions that enable reasoning about security at a high level and relieve the programmer and security reviewer of the burden of considering errors made in low-level details and obscure edge cases. In addition to firming up abstractions for general-purpose programming languages, this dissertation also introduces abstractions specific to web application development.

1.1.1 Security Review

Security bugs pose a substantial cost to users of software, and can also be harmful to the reputation of software companies.[1] While bugs are often found and patched, we have no

[1] Since they are so hard to avoid or predict, no software company wants to risk liability for buggy software, and thus it is routine to disclaim liability for misbehaving software, placing the burden of failure on the

assurance that there are not many more yet to be discovered. The worst case scenario from a security perspective is vulnerabilities that are not yet publicly known being exploited by stealthy attackers.

Many security bugs can be used to completely compromise the integrity of a computer system. The presence of these bugs necessitates frequent patching of systems and sometimes implementation of workarounds or disabling of features until such patches can be applied. Many systems are left vulnerable until a convenient time for patch application, making stealth exploits of the vulnerabilities (or others not yet disclosed) a constant risk. For those with strict security requirements, current commodity software in its standard configuration is not a reasonable option.

Unlike in other engineering disciplines, it is difficult to “over-engineer” software to avoid catastrophic failure. When building a bridge or building, one can compensate for possible inaccuracy in one’s calculations by adding a “safety factor,” intentionally making the structure (for example) 50% stronger than the calculated necessary strength. While there are occasions where unexpected stresses have led to failure of such structures (e.g. the Tacoma Narrows Bridge), for the most part the set of failure modes is well-known and understood. While some security threats such as brute-force attacks against encryption or passwords can be quantified, most are qualitative rather than quantitative. Bugs are notoriously difficult to predict.

Additionally, architectural structures are built with redundancy – if a handful of bolts or welds on a bridge turn out to be faulty, it will not fall down. The weight is simply shifted to adjacent bolts, which are provisioned to support the extra weight. The added expense of a few extra fasteners is well justified by the substantial increase in safety and expected longevity of the structure. This kind of redundancy is very rarely built into software. The cost of creating multiple independent implementations of the same software component is generally seen as cost-prohibitive.

In architecture, novel or complex designs for which well-established rules of thumb do not exist can be tested for strength and durability. If one wants to make sure that a bridge can carry ten tons of live load, one can put fifteen tons on it and monitor the structure to verify that it is bearing the load without damage. Testing is less effective for software. It can be useful in evaluating performance and reliability for non-malicious inputs (e.g. a trace or random data). Unfortunately, simple testing does little to demonstrate security, since the failure case may be very unlikely and hard to determine without examining the program. Directed testing techniques do a much better job in finding deeper security bugs, but no matter how many bugs are found via testing, there is very little assurance more bugs do not remain.

The only reliable way to be sure that a program has no exploitable bugs is to prove the program’s correctness with respect to a security policy. Such a proof is not easy to construct, especially if the program is written in a commodity language. Such languages are generally not designed to make formal verification easy.

consumer.

As an alternative to formal verification, security reviews with access to source code can be helpful in finding and eliminating security bugs in software systems. Most such reviews, however, are necessarily limited in scope. They aim to find bugs, or perhaps to eliminate certain types of bugs in a portion of the code, as generally software systems are large and complex enough that it is not possible to address all possible security flaws in a reasonable amount of time. Like testing, reviews can greatly improve the security of software, but rarely are able provide assurance that no flaws remain.

1.1.2 Programming Languages and Abstractions

Historically, programming languages have become progressively more high-level, abstracting away details exposed by their predecessors. In a new language, additional details are handled by the language infrastructure, where code generated by a well-tested compiler or a single carefully-written library implementation replaces application code to perform the same function. In general, this trend is beneficial for programmer productivity, software reliability, code readability and maintainability, and security. Each such advance has come at some cost in flexibility and performance relative to a carefully-tuned custom implementation provided by an expert programmer. However, over time a number of abstractions have become standard, seen as well worth these costs.

The very first stored-program computers required programmers to code in machine language. The earliest programming languages were assembly languages, in which operations are just mnemonics for machine code instruction sequences.[2] Even such a basic language is much easier to use than numeric opcodes, as the assembly instructions are easier to interpret and remember. Additionally, assembly languages allow locations in memory to be given names, making it easier to remember what data is stored where, and allowing labels to be given to jump targets. This may result in reduced errors, as the assembler will ensure that the mnemonic opcodes are valid and have the right number of parameters, and that labels used will be properly defined. The hazard of accidentally typing the wrong numeric opcode or address is greatly reduced. Assembly language provides a higher level of abstraction over machine code, as the mnemonic opcodes replace their corresponding bit sequences and labels replace memory addresses. The advantage of the abstraction is that assembly code is much easier to read and understand than machine code.

A major development in raising the level of languages was the creation of memory-safe and type-safe languages, with automatic memory management and compiler checks to ensure that in-memory data structures are accessed in a consistent manner. Such languages remove expressivity in the form of programmer control over memory management, but avoid a whole class of memory-management bugs. They also reduce the possible damage of a number of bugs that can still occur: an out-of-bounds array access, for example, will halt the program or propagate to an exception handler instead of corrupting the program's state. This change

[2] Initially, the task of "assembling" from a human-readable program in assembler to machine code was considered not worth the computer's time, and was performed by hand; but once such a rote task became cost-effective to perform by machine, the benefits of doing so were widely recognized.

is a big win for security: low-level memory management bugs, which are easy to introduce in older languages, often have devastating consequences for security.

Object-oriented languages add yet another level of abstraction. Each type of object can hide some of its implementation details from clients, providing a custom abstraction by way of its public interface. Users of such objects are limited in expressivity compared to having direct access to the object's internals. The advantage is that the designer of a class can, by providing an appropriate interface, defend against accidental or malicious misuse by the class's clients. If these abstraction boundaries are properly enforced, they can provide a secure mechanism for objects belonging to different parties to interact, without either having to fully trust the other.

Object encapsulation means that libraries can raise the level of abstraction for programmers without making changes to the language. This allows new abstractions to be introduced more easily than with a new language, as existing compilers and other tools can continue to be used. Libraries can introduce new concepts and higher-level interfaces that allow their users to operate at a higher level of abstraction and avoid errors that might be made using lower-level constructs. A popular example is the string class in C++. The base language includes character arrays from C, an error-prone construct as it is easy to run off the end of a character array, reading or writing other variables in memory. Strings, however, provide an abstraction of a sequence of characters of known length, and ensure that all operations stay within the bounds of the string.

Encapsulation has traditionally been seen primarily as a tool to guard against accidental misuse. It has been used for information hiding, modularity, and improving reliability. In this dissertation, I explore the use of object encapsulation to be capable of defending against intentional as well as accidental misuse. Encapsulation has the potential to establish well-defined boundaries inside an application that can benefit security and reviewability. Establishing this stronger notion of encapsulation requires some language support, which is a contribution of this dissertation and is described later in the introduction.

1.1.3 Web Templating Languages

As web applications have grown in popularity and importance to our daily lives, languages and frameworks geared specifically toward web development have proliferated. A common component of such frameworks is support for *web templating languages*, which are designed specifically for building the output logic of a website. These outputs are in HTML (hypertext markup language), the language used to specify the structure and content of web pages.

Web templating languages combine snippets of HTML markup and output logic code into a single *template*. The template is essentially a document with "holes" in the form of variables or code snippets. When receiving a request for the template's URL, the web server fills in the holes by printing the corresponding variables or executing the code. Web templates are a popular mechanism for developing the presentation component of a web application. Popular templating languages include PHP, JSP, and ASP, as well as a number of templating frameworks for Python and Ruby.

The manner in which templating languages combine trusted, developer-authored HTML with potentially untrusted content leads to a significant security concern: the possibility of injecting unwanted HTML content into the output document. Cross-site scripting (XSS) is the most dangerous type of HTML injection attack and involves the insertion of malicious JavaScript code. Malicious script can force users to invisibly perform actions that enrich the attacker, such as performing click fraud or transferring money to the attacker's bank account. They can also steal users' session credentials, allowing the attacker to co-opt their session.

In general, injection of any kind of unintended content into a website can be problematic. Even just the ability of an attacker to compromise the visual layout of a site can be sufficient for vandalism attacks. A more damaging attack that can also be carried out with a limited injection is the ability to spoof the trusted parts of the site's UI and fool users into entering their credentials into a malicious form that is submitted to a server controlled by the attacker.

HTML injection attacks can be prevented by properly sanitizing or escaping untrusted inputs before including them in web templates. Web template languages provide mechanisms for defending against these attacks. The earliest template languages, such as JSP, ASP, and PHP, alternate static content with snippets of a highly expressive language, which can be used to specify sanitization and escaping logic if necessary. Most recent template languages are more restrictive, designed to encourage greater separation between display and business logic. In these languages dynamic content is inserted via variable inclusions, which can specify escaping filters. These filters, usually standard ones provided by the language, can safely escape content for various HTML contexts.

It is not uncommon for more recent template languages to provide support for automatic escaping, in which the default behavior for a variable interpolation is to be escaped using an escape function in order to prevent injection of HTML markup or JavaScript. Systems that perform automatic escaping with a single escape function provide partial but not complete defense against injection attacks. This is because documents can include untrusted content in a number of distinct contexts, only some of which can be handled using any single escaping function. More sophisticated automatic escaping systems are context-aware, and thus can escape variables appropriately for the context in which they are used.

1.2 Summary

My thesis work improves the security of software by designing a language and enhancing a web application framework to facilitate reasoning about security. This makes it easier for programmers to design their systems to be secure in such a way that a security reviewer can verify the software's security.

My thesis work can be divided into two main parts: Joe-E and work on improving the security and auditability properties of templating languages used in web applications.

1.2.1 Joe-E: A Security-Oriented Subset of Java

In Chapters 2–6, I present Joe-E, a language designed to facilitate the building of secure software systems. By reliably enforcing memory-safety and reference-propagation rules, it provides a basis for reasoning about the behavior of programs. This can be used to make it easier to review software for security.

To be sure that a program is completely free of bugs, it would be necessary to formally verify the correctness of the entire program against a specification of its intended behavior. This is a very difficult task, and one that generally requires a lot of manual work, as the proof of correctness at a minimum mirrors the complexity of the program. In practice, writing a proof is often much more difficult than simply understanding the program's operation; code must usually be carefully written and extensively annotated to facilitate formal verification. This approach is not studied further in this dissertation.

If we are concerned with specific security or compliance requirements, a partial form of verification should be sufficient. Instead of requiring that the entire program is correct, we can instead identify the security properties we wish the system to provide and verify that these are maintained by the program. These properties will be application-specific; they can include integrity requirements on critical data structures, rules for the use of various system resources, and confidentiality requirements on information. Ideally, verifying such a property would only require reviewing the code that implements the functionality clearly relevant to that property. Unfortunately, current languages in popular use make verifying even such limited program correctness properties very difficult. This can often be traced to a lack of reliable isolation between components of a program. Without isolation, reviewers must take the whole program into consideration when verifying a security property.

In some languages, it is always necessary to verify the whole program because the language allows any line of code to read and write any variable defined elsewhere in the program. This means that the effective trusted code base for any security property is the entire program, resulting in "TCB explosion". Such explosion is the case for languages such as C or C++ that do not enforce memory-safety, allowing access to arbitrary memory addresses. More subtly, memory-safe languages can also make objects more available than might be expected. This can occur, for example, if there is a facility to rewrite the base classes of the language (as in JavaScript or Ruby) or if security-relevant objects can be reached from global variables. If an object relevant to a security property is addressable by code that has not been verified to maintain that property, we can no longer be confident of security.

In this thesis, I describe Joe-E, a language designed to address these challenges, facilitating security review of just the parts of a program that actually participate in enforcing a security property. Joe-E is based on Java and inherits the abstractions it provides for modularity and encapsulation.

As a memory-safe language, Java ensures that references can only be created from scratch by constructing a new object and that existing references can only spread by reference copying (e.g. by passing one as an argument). Joe-E, which is based on Java, inherits its memory-safety. Java allows *native methods*, however, that are executed as machine code and thus are not guaranteed to preserve the language's memory-safety properties. Joe-E forbids application code from defining native methods. While Java allows arbitrary objects to be stored in global variables, potentially exposing state that is sensitive or should not be modified, in Joe-E the global scope only contains *immutable* (non-modifiable) objects that are essentially constants. This helps Joe-E programs avoid TCB explosion.

Another important aspect of a language designed for code review is support for encapsulation. For security-sensitive objects that must be shared, it is still possible to maintain security guarantees if there is a way to enforce strong object encapsulation. Encapsulated objects do not expose their internal state, and instead provide a limited interface to the rest of the program. Such objects can hide their internal state and maintain security invariants on this state. Unfortunately, many languages provide ways to bypass encapsulation. Even if these features are rarely used, their presence means that it is not possible to verify an object's security in isolation: one must also check all users of the object to ensure that it they do not bypass the encapsulation.

Unlike Java, Joe-E provides support for strong object encapsulation. In Java, encapsulation is provided by *private fields*, which can be accessed by code within the object but are hidden outside it. However, Java's Reflection API (application programming interface) can be used to override this protection. To prevent breaches of encapsulation, Joe-E disables this part of the Reflection API.

Object isolation and encapsulation serve to ensure that objects can be appropriately protected from access that could violate their security properties. However, isolation and encapsulation are not enough to enforce security restrictions on resources external to the program. Most languages provide library methods that allow any part of the program to affect any external resource it likes. This makes it very hard for a part of a program to, for example, enforce an invariant about a file in the file system: the programmer would have to check all of the rest of the program and ensure that it does not access the same file.

In Joe-E, all external resources are represented as objects, each of which provides the ability to interact with an underlying operating system resource. Such objects can only be constructed given a capability that grants the ability to do so. This allows the same memory-safety properties that facilitate reasoning about in-memory objects to be applicable to other system resources such as files and network connections. By reasoning about reference propagation in Joe-E programs, it is possible to check security restrictions on external resources.

Joe-E provides a simplified model for reasoning about the security of a program: all *authority* (ability to observe any state or perform any action) is represented by references to objects, also known as *capabilities.* Code can always perform computation using objects it creates itself and the constant values available in the global scope. When it's done, it can return a result to its caller. Anything else, such as interacting with the outside world, requires an additional, explicit capability. Due to this, one can reason about what code can and cannot do based on which capabilities it can and cannot obtain. This reasoning is useful for verifying security properties of Joe-E programs.

Joe-E supports two basic forms of reasoning: what code can do, based on the capabilities that it has; and what can be done to an object or resource, based on which portions of the program can obtain a capability to the object or resource (directly or indirectly).

The first type of reasoning can be used to restrict untrusted or partially-trusted code. This permits securely combining code from a number of sources. Today, installing a plugin for an application is a dangerous operation. Generally, plugins are trusted with the full authority of the program they extend. Joe-E provides a way to solve this problem. Since Joe-E provides object isolation by default and secure encapsulation, plugins can be provided with a narrower interface that securely limits the damage they can do, reducing the risks of installing them.

One specific and powerful instance of this type of reasoning is the way that Joe-E enables the verification of methods as *functionally pure.* Functionally pure methods perform computations that depend only their arguments and have no side effects, like evaluating a mathematical function. Joe-E makes it easy to verify that a method has these properties. In particular, any Joe-E method all of whose arguments are immutable will be functionally pure. The method will be deterministic since the behavior of a method in Joe-E depends only on the objects it can observe, and the only observable state aside from the method's arguments will be constant global state. The method can have no side effects because it cannot obtain any capabilities to external objects, and thus cannot modify them. Any two invocations to a functionally pure method with equivalent arguments, even on different instances of the program running on different machines, will yield the same result. This property has a number of useful applications for security, as described in Chapter 5.

The second form of reasoning is useful for verifying security properties that relate to a particular resource (object). Joe-E makes it possible to track down the places where a reference to the object of interest can propagate to. If the use of the object in all of these locations is verified to maintain the object's security invariants in the face of arbitrary behavior of the rest of the code base, a security reviewer can be confident that security is maintained without having to look at the rest of the code. This approach is feasible because the arbitrary, unknown behavior of the rest of the program is still limited by the properties Joe-E enforces: if the object of concern is never handed to unreviewed code directly, but is always encapsulated in a defensively-programmed wrapper, reviewing the wrapper obviates the need to review the rest of the code.

Implementing a new language is only a productive exercise if the language is used. To encourage adoption of a new language, it is beneficial to leverage existing tools and programmer experience. Joe-E is designed to be attractive to programmers due to the fact that it is a subset of the well-known Java language. This means that every Joe-E program is just a Java program that has been verified to belong to the Joe-E subset. Joe-E programs run on an unmodified Java virtual machine in exactly the same way as Java programs, because they are Java programs. Joe-E provides a set of libraries that can be used by Joe-E programs, but these libraries are just normal Java libraries, and do not require any special support from the Java infrastructure.

Familiarity to Java programmers is furthered by making Joe-E as large a subset as possible while preserving the object-capability language properties. Most of the features of the Java language are included in Joe-E, so programmers can use these features when writing Joe-E programs. Features are only left out when they are inherently incompatible with the security properties Joe-E achieves.

1.2.2 Improving Security of Template-Based Web Applications

The second main portion of my dissertation (Chapters 7–9) concerns improving the security and auditability of web applications, in particular those built using web templating languages. I address two problems in the area of defending against content injection in web templates. The first is the problem of retrofitting existing code to make use of contextual automatic escaping in a way that protects against attacks while also preserving application functionality. The second is work on improving the soundness of contextual autoescaping and other defenses against content injection performed on the server.

Web frameworks are an increasingly popular tool for building web applications. They have been successful in introducing abstractions that can cut developer time as well as reduce the opportunity for bugs, with a corresponding increase in security. I identify web framework enhancements and automated tools that significantly simplify the task of converting existing applications to effectively use context-sensitive escaping. By reducing the number of trusted but error-prone exemptions from automatic escaping, these techniques can improve the level of security delivered by applications.

My approach has two modes of operation that provide a trade-off between retrofit effort and the security guarantee provided. In *mitigation mode*, it provides defense limited to cross-site scripting, but without requiring any modifications to the target application. In *strict mode*, it defends against all server-side content injection attacks, and reduce the need for manual opt-outs from automatic escaping by supporting common patterns for constructing safe HTML. Specifically, my system integrates with the web application framework's object-relational mapper to handle trusted HTML stored in its database. It also recognizes code patterns that build up HTML programmatically and provides an automatic rewriting tool to support it in a safe way.

I demonstrate my approach on a version of the Django template system that I have modified to dynamically enforce context-sensitive escaping of template variables. In my evaluation of eleven real-world open-source Django applications, I show that my approach is applicable to a variety of applications and helps make them reviewably secure from cross-site scripting with little effort.

The second problem is the important soundness issue of correctly inferring HTML parse contexts in templates. Contextual autoescaping fundamentally depends on the server being able to reliably determine how a client's browser will parse the output of the template. The automatic escaper must be able to predict how the document will be parsed in order to escape untrusted inputs correctly. If the browser parses the document differently from what is expected, malicious inputs may be incorrectly escaped and thus still able to attack the user. I describe a technique to substantially increase assurance that browsers will parse the document as predicted by server-side automatic escaping.

In order to handle variation in document parsing between browsers, I present a way to statically normalize templates, so that the HTML they generate will be predictably and unambiguously parsed by browsers. By appropriate normalization of a template, this guarantee is provided for every one of that template's infinite number of possible outputs. This task is complicated by template language features such as conditionals, template inclusion, and template extension. My approach adjusts the placement of a template's directives as well as modifying its HTML content to be as unambiguous as possible.

I aim to support all browsers in common use, including older ones that are not compliant with the latest HTML5 specification. This work provides a more solid foundation for defenses against cross-site scripting and other security analyses that rely on inferring how HTML documents will be parsed by browsers. I apply our methods to a collection of templates from real-life open-source web applications and demonstrate that their output can be normalized with reasonable developer effort.

I hope that my work in improving reviewability by language subsetting and leveraging web frameworks will encourage programmers to write secure software in a way that is practical for others to gain confidence in. I suggest that designers of future languages and systems consider reviewability for security in their designs and work to enable programmers to build systems that reliably achieve their security goals in a reviewable way.

Chapter 2

Joe-E: An Object-Capability Subset of Java

This chapter is based on a paper [41] coauthored with David Wagner and Tyler Close, presented at the Internet Society's 17th Annual Network and Distributed System Security Symposium in March 2010.

2.1 Introduction

This chapter describes the design and implementation of a programming language, called Joe-E, that supports development of secure systems. Joe-E improves upon today's languages in two important dimensions. First, Joe-E makes software more robust by reducing the number and impact of inadvertent bugs and security vulnerabilities in benign software. Second, Joe-E provides flexible mechanisms to reduce a program's vulnerability to software components and allow the safe usage of untrusted code. Both characteristics help to make code more amenable to code review and security audits, an important property when we place trust in computer systems' correct operation and attack resilience. In particular, Joe-E supports construction of systems following a "secure by design" approach, where security is designed in from the start, as well as "design for review", where the code is architected and written specifically to make it as easy as possible for code reviewers to verify that the application meets its security goals.

Joe-E is based upon the Java programming language. We show that a relatively small number of simple restrictions suffice to define a subset of Java that provides the security properties of an *object-capability language*. (In an object-capability language, all program state is contained in objects that cannot be read or written without a reference, which serves as an unforgeable capability. All external resources are also represented as objects. Objects encapsulate their internal state, providing reference holders access only through prescribed interfaces.)

A major contribution of our work is that we bring the security benefits of object-capability systems to a popular language. Additionally, we show how Java's static type system can be used to simplify the assurance of security properties statically, as opposed to via runtime checks used by the dynamically-typed object-capability languages found in prior work.

Memory-safe languages like Java make it much easier to design robust systems and reason about their security properties than non-memory-safe languages, but in Java it is still difficult to reason about higher-level security properties, particularly when composing code with varying levels of trust or when auditing the security of a program. With Joe-E we are able to support richer ways of combining code entrusted to varying degrees while reviewably maintaining security properties.

Providing secure encapsulation. Consider Fig. 2.1(a), which illustrates how one might build an append-only log facility. Provided that the rest of the program is written in Joe-E, a code reviewer can be confident that log entries can only be added, and cannot be modified or removed. This review is practical because it requires only inspection of the `Log` class, and does not require review of any other code. Consequently, verifying this property requires only *local* reasoning about the logging code.

Perhaps surprisingly, Java does not support this kind of local reasoning. Because Java allows the definition of native methods which can have arbitrary behavior and violate Java's safety properties, all bets are off unless one is sure that the program does not use any such methods. Even if the program uses no native methods, the append-only property of the above code is not guaranteed. Java's reflection framework includes the ability to ignore the visibility specifier on a field, which would allow a reference-holder of the `Log` object to retrieve the `StringBuilder` contained within as if its field were declared to be public. This would violate the append-only property, as it would then be possible to perform arbitrary operations on the `StringBuilder`. While we might intuitively expect that the rest of the program would be unlikely to exploit these weaknesses, we would have to read all of the code of the entire application to be sure.

Joe-E removes these and other encapsulation-breaking features from Java in order to support building and reasoning about secure systems. This makes building sound, self-contained application reference monitors possible. Because these reference monitors are written as part of the application software itself, this provides a powerful mechanism for enforcing security policies: the programmer has the full power of the Joe-E programming language for expressing these security properties, and does not need to learn a new security specification language to specify them. We anticipate that this will aid developers in implementing custom security abstractions.

Reliable, auditable behaviors such as this logging system can be valuable not only for security but also in enforcing compliance with specifications and regulations. A tamper-proof logging subsystem can provide greater assurance that transactions are recorded in compliance with Sarbanes-Oxley and HIPAA record-keeping requirements.

Capabilities and least privilege. In the example above, only the parts of the program that have access to an instance of the log object will be able to add log entries; the rest of the program will be unable to affect that log instance. In particular, a reference to a `Log` object is a *capability* to append entries to that log. We can control which parts of the program receive the power to append to the log by controlling who receives a reference to the log object. The rules for propagation of these capabilities are exactly the rules for propagation of references in a type-safe language, which should already be familiar to the programmer; we expect this will make it easier for programmers to reason about capability propagation.

For instance, we might have an application where it is critical that every incoming network request be logged. We could provide the component that dispatches incoming requests a capability to the log, so it can log every incoming request. By examining the dispatcher component, we can verify that every incoming request is logged using only local reasoning. If required, we could also verify that no other log entries are added, by checking that no other component can receive a reference to the log.

As another example, a device capable of radio communication may need to be limited to certain frequencies via software in order to receive FCC certification. Even if the hardware interface exposed by the device driver would permit communication on disallowed frequencies, in a language like Joe-E it is possible to build a wrapper that reliably limits how the radio is used. In Joe-E, it would be necessary to review only the interface that such a wrapper presents to the rest of the application, as there is no way for Joe-E code to violate the wrapper's encapsulation of the dangerous radio capability.

Capabilities also support least privilege. Code can only write to the log if it has a capability to the log object. Code that is not explicitly passed this capability has no access to it, which means that by default the overwhelming majority of code is verifiably incapable of writing to the log. Our experience is that this encourages a style of programming where only the code that legitimately needs the power to append to the log receives a capability to do so.

Analysis of who has access to an object and the principle of least privilege are both subverted when capabilities are stored in global variables and thus are potentially readable by any part of the program. Once an object is globally available, it is no longer possible to limit the scope of analysis: access to the object is a privilege that cannot be withheld from any code in the program. Joe-E avoids these problems by verifying that the global scope contains no capabilities, only immutable data.

The Java standard library also provides a large number of capabilities to all Java code, for example, the ability to write to any file that the JVM has access to. In the context of our example, this would include the file where the log is ultimately output. For this reason, Joe-E allows access to only a safe subset of the standard Java libraries.

Modular security abstractions. Suppose we want to provide a software subsystem the ability to append entries to the log, but we want to ensure that those entries can be attributed to that subsystem. Fig. 2.1(b) shows a solution. We can construct an `AttributedLogger`

```
public final class Log {
  private final StringBuilder content;

  public Log() {
    content = new StringBuilder();
  }

  public void write(String s) {
    content.append(s);
  }
}
```

```
public class AttributedLogger {
  private final Log log;
  private final String id;

  public AttributedLogger(Log log, String id) {
    this.log = log; this.id = id;
  }

  public void write(String message) {
    // prevents message spoofing via newlines
    for (String line : message.split("\n")) {
      log.write(id + ": " + line + "\n");
    }
  }
}
```

Figure 2.1: (a) On the left, an append-only logging facility. (b) On the right, an extension that associates each line in the log with an assigned name or other identifier.

```
public interface Decoder extends Immutable {
    /** Returns a bitmap; retval[x][y][c] is the value
        at position (x,y) of color channel c. */
    byte[][][] decode(byte[] imagedata);
}
```

Figure 2.2: An untrusted image decoder might implement this interface.

object, specifying a name that will identify the subsystem, and pass it to the subsystem. Log entries written through this interface will then have the specified name prepended to them. In this way we can provide logging services to multiple subsystems in a program, while preventing them from impersonating each other. In this example, we have built a new security abstraction (the `AttributedLogWriter`) on top of an existing security abstraction (the append-only `Log`). The ability to layer security abstractions in this way is important for modularity.

Untrusted code and extensibility. Joe-E also allows applications to safely execute and interact with untrusted code. This safety is a result of the fact that Joe-E objects spring to life with no capabilities other than the ones passed to them when they were constructed. They can only acquire additional capabilities that they are explicitly passed. As a result, Joe-E is well suited to execution of untrusted code, since untrusted code written in Joe-E cannot harm anyone if it is not passed any dangerous capabilities. Partially trusted code can be granted only capabilities appropriate to its function and the level of trust placed in it.

This aspect of Joe-E provides support for secure extensibility. For instance, consider a graphics viewer program that can be extended with plugins for various file formats. We'd like to be able to download a plugin that interprets new image files, without exposing ourselves to attack from malicious code. We want to ensure that the worst a malicious plugin could do is incorrectly decode an image, but for instance it must not be able to send network packets, write to the filesystem, or interfere with decoding of other images.

In Joe-E, we could enforce this by requiring plugins to be written in Joe-E and to implement the interface in Fig. 2.2. For instance, a JPEG decoder could implement this interface, interpreting the data passed to it as a JPEG image and converting the result to a bitmap to be displayed. If the plugin is only invoked through this interface, Joe-E guarantees the following remarkable security property: multiple invocations of this method will be independent, and no state can be retained or leaked from invocation to invocation. This ensures both confidentiality (because information about a confidential image cannot leak into other images, even if the plugin is buggy) as well as integrity (even if the plugin contains bugs that can be exploited, say, by a maliciously constructed image, these exploits cannot interfere with the decoding of other images or otherwise harm the rest of the system, except by decoding the malicious image to an unexpected bitmap).

The `Immutable` interface, defined by the Joe-E library, is treated specially by the language: the Joe-E verifier checks that every object implementing this interface will be (deeply) immutable, and raises a compile-time error if this cannot be automatically verified. Since the `Decoder` interface extends `Immutable`, decoding plugins will necessarily be stateless. Also, because only byte arrays can flow across this interface, it is easy to verify (thanks to the static type system) that plugins will never receive a capability that allows them to interact with any other system component.

Reviewable, rich behavioral properties. Joe-E can be used to enforce rich, application-specific behavioral security properties. Fig. 2.3 defines a currency system. If used, for instance, in an online game, it would be easy to verify that trades between players cannot generate money from nothing. A `Currency` object provides the power to mint new money in the corresponding currency; it is impossible to do so without a reference to this object. A `Purse` can be used to hold and transfer money in a particular currency, but does not grant the power to mint new money.

Note that this API is general enough to support multiple currencies, and can easily be audited for correctness, even in the presence of multiple mutually-distrusting and potentially malicious clients. In particular, to verify that the currency API cannot be abused, one only need examine the code of the `Currency` and `Purse` classes—nothing more. From this code we can deduce, for instance, that it is only possible to create money in a currency if one has a reference to the corresponding `Currency` object. This kind of local reasoning is made possible because Joe-E enforces encapsulation boundaries that follow the program's lexical scoping structure.

Joe-E enables us to concentrate trust in a small, comprehensively reviewable portion of the code, which serves as the trusted computing base (TCB) for a specific security property.

```
public final class Currency { }

public final class Purse {
    private final Currency currency;
    private long balance;

    /** Create a new purse with newly minted money,
        given the Currency capability. */
    public Purse(Currency currency, long balance) {
        this.currency = currency;
        this.balance = balance;
    }

    /** Create an empty purse with the same currency
        as an existing purse. */
    public Purse(Purse p) {
        currency = p.currency; balance = 0;
    }

    /** Transfer money into this purse from another. */
    public void takeFrom(Purse src, long amount) {
        if (currency != src.currency
            || amount < 0 || amount > src.balance
            || amount + balance < 0) {
            throw new IllegalArgumentException();
        }
        src.balance -= amount;
        balance += amount;
    }

    public long getBalance() {
        return balance;
    }
}
```

Figure 2.3: A secure abstraction that supports flexible use of currencies.

Here the `Purse` only needs to be trusted to correctly enforce the security properties associated with the currency, and not for other purposes. This pattern encourages architecting a program so that for each desired security property, we can identify a small TCB for that property. Such a software architecture can, in turn, significantly reduce the cost of verifying security properties of the application.

2.2 Goals and Overview

We have three primary design goals for the Joe-E language. First, we want Joe-E to be usable by programmers. Second, we want Joe-E to support construction of secure systems. Third, we want to make it easier to verify that the resulting systems meet their security requirements, and ease the task of security code reviews. We elaborate on these goals below, and sketch Joe-E's approach to each of those goals.

2.2.1 Ease of use

To minimize barriers to adoption of Joe-E and reduce the learning curve for new Joe-E programmers, the language should be as familiar as possible for programmers. Joe-E should minimize as much as possible the requirement to learn new concepts or idiosyncratic syntax. To address this goal, the Joe-E programming language is based upon Java (§ 2.3.1).

Also, as much as possible, Joe-E programmers should be able to use existing development tools, build on existing libraries, and integrate with legacy systems. We partially support this goal by designing Joe-E as a subset of Java (§ 2.3.1) and exposing a capability-secure subset of the Java class libraries (§ 2.4.2).

Joe-E should support construction of new modules, written from scratch with security and Joe-E in mind. To receive the full benefits of Joe-E, software must be structured in a way that is compatible with good capability design principles. We do *not* aim to add security to existing Java code. Legacy Java code will most likely not be valid Joe-E, and even if it were, legacy code often fails to be structured in a way that respects capability principles. It is explicitly not a goal of this work to make it easy to transform arbitrary existing Java code into Joe-E; Joe-E is intended for newly written code.

While existing Java code may not transform easily to Joe-E, Java code can easily make use of modules written in Joe-E. For example, an existing Java application may add support for plugins implemented in Joe-E, thereby limiting the damage that plugin authors can cause to the main application. Similarly, a large Java application may be incrementally migrated to Joe-E by rewriting its component modules. Because any Java component of a combined application is unrestricted in privilege, it must be considered part of the trusted computing base. This Java component has the potential to, via the abstraction-breaking features of Java, violate the security properties of Joe-E code, and so requires the same level of careful review required for an all-Java application. Use of Joe-E components neither facilitates nor complicates review of Java code; the benefit is a reduction of the amount of Java code to be reviewed.

Additionally, we desire Joe-E to have the expressivity and scalability to support large, real-world systems. We do not want our abstractions or implementation to place restrictions on the scale or complexity of applications that can be written in the language.

2.2.2 Supporting secure software

To facilitate construction of secure systems, Joe-E should:

1. *Encourage least privilege.* Joe-E is intended to help programmers achieve the principle of least privilege, at a fine level of granularity in their program, so that each subsystem, module, and object receives only the minimum privilege it needs to accomplish its task. Joe-E should minimize barriers to least-privilege design of software.

   Joe-E supports this goal through safe defaults: by default, each block of code has no privileges to access system resources, and can acquire such privilege only if some other entity passes it an appropriate capability (§ 3.4). In comparison, the default in most other software platforms is that code runs with all of the privileges of the user who invoked it, and must explicitly drop privileges if that is desired; Joe-E reverses this presumption (§ 2.4.2). Joe-E's libraries provide a capability interface to system resources (e.g., the filesystem and network). Also, applications written in Joe-E can devise their own security abstractions that divide up privileges into smaller pieces appropriate to the application domain (§ 2.5.5), further supporting least-privilege programming. We expect that systems built in this way will be more robustly secure, because the effect of bugs and vulnerabilities is limited: the fewer privileges a component has, the less harm it can do if it misbehaves or runs amok.

2. *Isolate untrusted code.* We want programs to be able to run untrusted or mobile code safely. Moreover, we want programs to be able to interact usefully and efficiently with the untrusted code—and in particular, we want to be able to run untrusted code in the same JVM as trusted code. This implies that simple isolation is not enough; programs must be able to "poke holes in the sandbox" to enable controlled sharing. We would like the trusted program and untrusted program to be able to share access to common data structures, and we want cross-domain calls to be as efficient as a method call.

   Because Joe-E code receives, by default, no capabilities, it is safe to execute untrusted code that is written in Joe-E (§ 3.4). We can limit what the untrusted code can do, by limiting what capabilities we provide to it; and conversely, we can grant the untrusted code limited powers by passing it appropriate capabilities. For instance, we can enable the untrusted code to write to a single file on the filesystem, by passing it a capability for that file. In Joe-E, data structures can be shared between components simply by passing a reference to the data structure, and cross-domain calls are a method call.

3. *Enable safe cooperation.* As a generalization of the previous point, we also want to enable mutually distrusting subsystems to interact safely. Each party should be able to limit its exposure, should the counter-party be malicious. Joe-E helps with this goal by supporting strong encapsulation, down to the object granularity. Each object can be written to enforce its invariants while protecting itself from code that makes use of it (§ 2.5.3).

2.2.3 Supporting security code review

Joe-E should help programmers follow a "design for review" philosophy, where the software architecture and implementation are carefully chosen to facilitate security code review. Joe-E should:

1. *Enable reasoning about privileges.* It is not enough for Joe-E to enable least privilege and isolation; it should also be feasible for reviewers to verify that these security goals are achieved. Accordingly, Joe-E should help reviewers upper-bound the set of capabilities a particular block of code might ever gain access to, or upper-bound the portions of the program that might ever gain access to a particular capability. Joe-E should also make it possible to write code so that these upper bounds are precise and easily verifiable. To help with this, Joe-E is designed to enable several powerful patterns of reasoning about the flow of capabilities in the program (§ 3.7.4).

2. *Support modular reasoning.* Joe-E should make it easier to reason about security properties. If the program is written appropriately, it should be feasible to verify a security property by examining a small fraction of the code. If the object O implements some security abstraction, it should be possible to reason about the security properties of this abstraction (e.g., the invariants maintained by O) just by looking at the source code for O and the objects O relies upon. In particular, if *client* objects $C_1, \ldots, C_n$ make use of O, we should be able to verify the correctness of O without examining the code of any client C_i. We call this *modular analysis.* Modular analysis is critical if security code review is to scale to large programs.

   Joe-E's strategy for supporting modular reasoning about security relies heavily on flexible support for isolation of untrusted code (§ 2.5.3). Also, many of our restrictions on Joe-E code support modular reasoning: the more we restrict what Joe-E code can do, the more we can restrict the possible behaviors of each client C_i, which makes it easier to ensure that they do not violate O's invariants.

3. *Support reasoning about mutability.* Shared mutable state is a headache for reasoning about security, because it introduces the potential for race conditions, time-of-check-to-time-of-use vulnerabilities, and surprising consequences of aliasing. Joe-E should help programmers avoid these risks by providing first-class support for reasoning about mutability and immutability. In particular, Joe-E should make it easy for programmers to build data structures that are transitively immutable, should provide support for

static verification of this fact, and should reflect these immutability properties in the static type system. Joe-E addresses this by extending the Java type system with immutability annotations (§ 2.5.4), by providing library support for programming with immutable data, and by forbidding mutable global variables (§ 2.4.2, § 2.5.2).

2.3 Approach

Our approach to improving language security is through the use of an object-capability language. Such languages permit a default-deny, least-privilege approach to the authority granted to parts of a program as it executes.

The central feature of object-capability languages is that they use object references (pointers to objects) to represent all of the privileges that can be used by a program. In the simplest case, these simply point to encapsulated memory-resident objects. Having a pointer to such an object grants the ability to interact with it via its public interface. Since access to the object is limited to the interface, the object can be designed to maintain the privacy and integrity of its internal state even when passed to untrusted code.

For many purposes, a system that can only operate on in-memory objects is not enough. Most programs need to interface with other resources on the system or network. In object-capability languages, these resources are represented as objects defined by special library classes. Reference to such an object allows interaction with the external resource via a library-defined public interface. In this way, files on disk and network connections are naturally represented as objects.

Access to all references in an object-capability language is governed by program scope. At any point in time, the program can only make use of the capabilities that are reachable from its in-scope references. For such an approach to be sound, the language must be memory-safe: it must be impossible to "forge" a pointer to an object, such as by performing a type cast operation on a memory address.

To get the most benefit from this approach, we want the minimal set of privileges we can bestow on part of a program to be as small as possible. We'd like the "default state" for running code to be one in which no harm can be done unless we explicitly trust it with a reference. For this to be the case, the global scope (which is available everywhere in the program) should not allow access to any authority we would want to deny to completely-untrusted code.

Ideally, we want code to be unable to do anything unless we have granted it a capability to do so. In real systems, we may need to relax this slightly for practical reasons; it may be easy to limit access to in-memory objects and external resources, but too difficult to prevent code from consuming CPU cycles or memory, or failing to return in a timely manner. Our approach is to place no limits on the purely computational power of untrusted code, limiting only its access to data and external resources. If the global scope grants no access to privileges of concern, one can enforce least privilege on a fine-grained basis by ensuring that each scope in the program's execution only has access to the capabilities it needs. More importantly, it is possible to reason about the authority with which different parts of the program are trusted. Every component of the program has only the capabilities that have been passed to it.

In contrast with most other object-capability languages, which use dynamic typing, in Joe-E we can leverage the Java type system to place static restrictions on how capabilities can propagate as a program executes. With this approach we are able to restrict the flow of capabilities while reducing the need for reference monitors and explicit dynamic checks in order to guarantee security properties.

2.3.1 Subsetting

Many new languages have been proposed over the years, but relatively few have seen widespread adoption. Programmers have large amounts of experience with and code in existing languages, and thus are reluctant to invest in switching to a new language.

A number of new languages have been defined as extensions to existing languages. This has the advantage of leveraging developer experience and preserving a greater degree of familiarity than defining a new language from scratch. Unfortunately, programs written in the extended language become incompatible with tools (debuggers, interpreters, profilers, IDEs) designed for the original language. Developers are wary of becoming locked into such extended languages, as they are not guaranteed to maintain the same level of support as the base language going forward.

We take a different approach: we define the Joe-E language as a *subset* of Java. Every Joe-E program is simply a Java program that satisfies additional language restrictions that are verified by the Joe-E verifier. We avoid adding new features to Java or making changes to Java's semantics; instead, we impose restrictions on the source code that every valid Joe-E program must satisfy (see Fig. 2.4 and § 2.4). The Joe-E verifier checks that these restrictions are met, but not does not transform the program in any way. This approach allows use of the standard Java tools, compiler, and runtime, as well as allowing Joe-E programs to coexist with Java code and libraries.[1] More importantly, this allows us to leverage programmers' experience with the Java language, while introducing security-oriented programming patterns. Joe-E can be thought of as simply an idiomatic way to write Java code, using conventions that facilitate a style of reasoning. The Joe-E verifier ensures that all checked code conforms to these conventions.

2.4 Design of Joe-E

The Joe-E language restrictions are chosen so it will be intuitive and predictable to the programmer which programs will pass the Joe-E verifier. We avoid sophisticated program analysis, instead favoring programming rules that are simple to state. For similar

[1] There is also no need to present formal semantics for the Joe-E language, as they are identical to those of Java.

Enforce reference unforgeability
- prohibit defining native methods

Prevent unexpected reference propagation
- require all throwables to be immutable

Remove ambient authority
- tame Java APIs that provide access to the outside world without an explicit capability
- require all `static` fields to be `final` and of an immutable type

Enforce secure encapsulation
- prohibit overriding `finalize()`
- tame Java reflection API
- prevent catching `Errors`
- prohibit `finally` keyword

Figure 2.4: Overview of restrictions that Joe-E imposes to enforce capability security.

reasons, we avoid whole-program analysis. Instead, the Joe-E verifier analyzes each source file individually. This file-at-a-time approach also helps scalability and lets us support open-world extensibility: new code can be added to the system, without invalidating the analysis previously performed on files that have not changed.

2.4.1 Memory Safety and Encapsulation

Memory-safe languages like Java provide the foundation for sound object-capability languages, as they ensure object references cannot be forged. In Java, references cannot be created by pointer arithmetic or casting integers to pointers, but rather can only be obtained by copying existing references. The site at which an object is created using the `new` operator is initially the sole holder of a reference to the new object and has control over how the object is shared. This memory safety property can be broken through the use of native methods, so Joe-E prevents the definition of such methods.

The access modifier `private` allows an object to encapsulate a reference to another object in such a way that it can only be accessed via the enclosing object's methods. The public interface of the enclosing class then dictates the policy for use of the wrapped object. Capability-secure programming relies crucially on the security of this encapsulation property. Java's reflection API provides a facility for disabling access checks on methods and fields, allowing malicious clients to bypass object encapsulation. To ensure that encapsulation cannot be broken, we do not expose this facility to Joe-E code.

Another Java feature with surprising consequences is the ability to define custom finalization behavior, by overriding the `finalize()` method. The garbage collector invokes user-defined `finalize()` code when an otherwise dead object is collected. This can violate

```
public class OddInt {
   final int content;

   public OddInt(int content) {
      if ((content % 2) == 0)
         throw new IllegalArgumentException();
      this.content = content;
   }
}

class EvilOuterClass {
   OddInt stash;
   class NotReallyOddInt extends OddInt {
      NotReallyOddInt() {
         super(0);
      }
      void finalize() {
         stash = this;
      }
   }
}
```

Figure 2.5: `finalize()` can violate object invariants, subverting encapsulation. In this example, `stash` can contain an object whose `content` field is uninitialized and thus has the value of zero.

object invariants that could be crucial to security, breaking encapsulation. See Fig. 2.5, which illustrates how malicious code (`EvilOuterClass`) could construct an `OddInt` instance that holds an even integer, subverting the checks in the `OddInt` constructor. Joe-E prevents these encapsulation-breaking attacks by prohibiting Joe-E code from defining custom finalizers.

2.4.2 Removing Ambient Authority

The privileges provided by Joe-E's global scope are strictly limited. We prevent Joe-E code from reading or modifying any mutable state or external resource without an explicit capability to do so.

This is perhaps our most significant and visible departure from Java's architecture. In Java, even code that starts out without any references has essentially all the privileges of the program; its lack of references does little to contain it. The authority that it needs to perform these tasks is available as an "ambient" property of the process: it is available to all code, in every scope. In Joe-E, no authority is ambiently available, so the resources needed by Joe-E code must be explicitly provided, typically as constructor arguments. This design refactoring is the same as that done for "dependency injection", where code that depends on a resource is provided with a reference to the resource, instead of constructing or accessing the resource directly. In dependency injection, this refactoring is done to better support the configuration and testing of software. In Joe-E, this refactoring additionally supports security review of software.

Taming the Java class library

The Java library defines many static methods that have side effects on the outside world, as well as many constructors that create objects permitting similar effects. This is a major source of ambient authority in Java. For example, `File` has a constructor that will take a string and return an object representing the file with that name. The resulting object can be used to read, write, or delete the named file. Absent explicit access control by the Java security manager or the operating system, this allows any Java code full control over the filesystem. In Joe-E, we wish to ensure that code can only have access to a file if a capability for the file (or a superdirectory) is within that code's dynamic scope. Consequently, we must not allow the aforementioned `File` constructor in Joe-E's global scope.

We define a subset of the Java libraries that includes only those constructors, methods, and fields that are compatible with the principle that all privileges must be granted via a capability. We call this activity *taming*, because it turns an unruly class library into a capability-secure subset. The Joe-E verifier allows Joe-E programs to mention only classes, constructors, methods, and fields in this tamed subset. If the source code mentions anything outside of this subset, the Joe-E verifier flags this as an error.

Taming helps eliminate ambient authority, because it ensures library methods that provide ambient authority are not accessible to Joe-E programs. We also use taming to expose only that subset of the Java library that provides *capability discipline*. Intuitively, we'd expect that a reference to a `File` object would provide access to the file that the object represents (or, in case it represents a directory, access to the directory and all files/subdirectories within that subtree of the filesystem hierarchy), and nothing more. Unfortunately, the `getParentFile()` method on `File` violates this expectation: it can be used to walk up the directory hierarchy to obtain a capability for the root directory, so access to any one `File` would grant access to the entire filesystem. This prevents fine-grained control over delegation of file capabilities, so we exclude methods, such as `getParentFile()`, that violate capability discipline.

In some cases, due to the design of the Java libraries, there are methods with important functionality that are not safe to expose. For instance, consider the `File(File dir, String child)` constructor. This constructor gives a way to access a file with a specified name within a specified directory. This pattern of obtaining a specified subfile is a capability-compatible method for attenuating existing authority, but Java happens to specify this constructor to have additional behavior that is not compatible with our security model: if the `dir` argument is null, the constructor treats the `child` argument as an absolute rather than relative path. This means that `new File(null, path)` can be used to access any file on the filesystem, so this constructor must not be exposed to Joe-E code. Joe-E programmers still need some way to traverse the directory hierarchy, and unfortunately there is no other constructor in the Java library that provides this important functionality. While we can't allow Joe-E code to call the unsafe constructor directly, we provide a wrapper method in the Joe-E library with the desired functionality. The wrapper checks at runtime that the `dir` argument is non-null before invoking the original constructor[2]. In general, our strategy is to tame away all unsafe methods from the Java libraries, then add wrappers to the Joe-E library if important functionality has been lost.

Taming a library is unfortunately a time-consuming and difficult task, and a place where a mistake could violate soundness of our security goals. The security review of the DarpaBrowser, which included a review of the taming database provided by the E language, found that a number of methods violating capability discipline had been inadvertently allowed [73]. While we have attempted to be more conservative when taming Joe-E code, checking each method for safety before enabling it and erring on the side of caution when unsure, it is possible that we also enabled some method that we should not have. We consider the difficult and critical nature of this process to be a substantial weakness in our approach, and an area in which there is substantial room for improvement in future work. In particular, tools to validate or safely automate taming decisions would be very helpful. (We anticipate that a relatively small fraction of classes in a typical Java classpath implementation are valid Joe-E in their current form, but those that are would be safe to permit.)

Mutable state

In addition to being able to observe or affect external state outside the JVM, ambient authority to modify program state can also be problematic. Untrusted extensions could corrupt critical internal data structures if the global scope provides the ability to do so. For the purposes of security audits, such exposure means that every line of code in the program must be examined to ensure that security properties on globally accessible state are maintained.

In Java, this risk arises with fields declared `static`, since these fields are not associated with an object instance and thus access is not governed by a capability. For this reason, Joe-E requires all `static` state to be transitively immutable. In particular, all `static` fields declared in Joe-E code must be of a type that is statically known not to provide access to any mutable objects: the object itself and all objects it transitively points to must be immutable.

To facilitate this goal, we provide a *marker interface*, `org.joe_e.Immutable`, to identify classes claimed to be transitively immutable. The Joe-E verifier checks that any class that is a subtype of `Immutable` satisfies the following rule: all instance fields must be `final` and their declared type must be either a primitive type or a reference type that also implements `Immutable`. All other classes are assumed to be potentially mutable.

We make no attempt to infer immutability types. Joe-E's philosophy is to require programmers to explicitly declare the properties of their code. The Joe-E verifier is responsible solely for verifying these properties, and performs no inference. This design decision

[2] Portions of the Joe-E library are written in unrestricted Java rather than Joe-E and thus can call arbitrary Java methods. This gives us the ability to write such wrappers.

is intended to make the behavior of the Joe-E verifier more intuitive and predictable for programmers.

Some classes from the Java library, like `String`, are immutable but we cannot rewrite them to implement the `Immutable` interface, because we do not modify the Java libraries. The verifier treats these classes as if they implement the interface.

2.4.3 Exceptions and Errors

Exceptions introduce a number of complications for an object-capability language. They provide a potentially unexpected means of transferring control and references between objects. In particular, objects reachable from the exception itself are implicitly passed up the stack from where the exception is thrown to where the exception is caught. If the exception contains a capability, this can lead to propagation of privileges that a developer might not expect, which might introduce unexpected security vulnerabilities.

To see how this can cause unpleasant surprises, suppose Alice calls Bob. Bob has some special capability that she lacks, and Bob wants to avoid leaking this to her. At some point, Bob might need to invoke Chuck to perform some operation, passing this capability to Chuck. If (unbeknownst to Bob) Chuck can throw an exception that Bob doesn't catch, this exception might propagate to Alice. If this exception contains Bob's precious capability, this might cause the capability to leak to Alice, against Bob's wishes and despite Chuck's good intentions. See Fig. 2.6 for an example.

The problem is that it is hard to tell, just by looking at the code of `Bob`, that Bob's private capability can leak to the caller of `m()`. This is a barrier to local reasoning about the flow of capabilities. To avoid these kinds of problems, Joe-E requires all exception types to be immutable.[3] This prevents storing capabilities in exceptions, precluding attacks like the one described above.

An important guarantee provided by Joe-E is that no code is able to execute once an error is thrown. This is necessary for two reasons. First, the behavior of the JVM after a `VirtualMachineError` is technically undefined [36, §6.3]. Second, continuing to execute after an error has been thrown can have hard-to-predict consequences. For example, an object's invariants can be violated if an error (such as running out of memory) is encountered during execution right when the object is in a temporarily inconsistent state. In many cases, these errors can be intentionally triggered by the invoking software component, for example by allocating a lot of memory or recursing deeply to use up stack space before invoking the object under attack. If a malicious caller could catch such an error, the caller would be well-positioned to exploit the violated invariant. Preventing Joe-E code from executing after any error is thrown prevents such attacks. Without such a guarantee, it would be unreasonably difficult to build secure abstractions and maintain object invariants in the face of attack.

We prohibit Joe-E code from including any `catch` block that could catch an error: for

[3] The `Throwable` class provides a little-used facility to rewrite the stack trace in an exception, preventing exceptions from being truly immutable. This facility is disabled in Joe-E via the taming mechanism.

```
class E extends RuntimeException {
   public Object o;
   public E(Object o) { this.o = o; }
}
class Bob {
   // cap was intended to be closely held
   private Capability cap;
   void m() {
      new Chuck().f(cap);
   }
}
class Chuck {
   void f(Capability cap) {
      ... do some work ...
      throw new E(cap);
   }
}

class Alice {
   void attack() {
      Bob bob = ...;
      try {
         bob.m();
      } catch (E e) {
         Capability stolen = (Capability) e.o;
         doSomethingEvil(stolen);
      }
   }
}
```

Figure 2.6: There is a security risk, if exceptions can contain capabilities.

```
InputStream in = ...
try {
  // use the stream
} finally {
  in.close();
}
```

```
InputStream in = ...
Exception e = null;
try {
  // use the stream
} catch (Exception e2) {
  e = e2;
}
in.close();
if (e != null) { throw e; }
```

```
InputStream in = ...
try {
  // use the stream
} catch (Exception e) {
  try { in.close(); }
  catch (Exception e2) {}
  throw e;
}
in.close();
```

Figure 2.7: Transformation to avoid the use of the `finally` keyword. On the left is Java code that uses `finally`. The middle shows a transformed version with the same semantics that can be used in Joe-E. The right shows an alternative, with different semantics, that we have found useful in our experience.

the syntactic construct `catch (T e) { ... }`, we check that the type `T` is not `Throwable`, `Error`, or any subtype of `Error`.

In addition, we prohibit `finally` clauses, as code in a `finally` clause can execute after an error is thrown. The `finally` clause could exploit the inconsistent state directly, or it could throw its own exception that masks the pending error, effectively catching and suppressing the error. Technically, the lack of `finally` clauses does not limit expressivity, as one can explicitly catch `Exception` to ensure that an action takes place whenever any non-error throwable disrupts normal control flow. See the middle of Fig. 2.7 for an example[4]. In our experience writing Joe-E code for the Waterken server, the prohibition on `finally` clauses was not a serious problem, and in retrospect the replacement code used in Waterken (shown on the right side of Fig. 2.7) is arguably better anyway, as it avoids masking the original exception in case the `finally` clause throws its own exception. The Joe-E specification [39, §4.8] contains further discussion and analysis of these workarounds.

2.5 Programming Patterns

To facilitate our goal of "design for review", Joe-E was designed specifically to enable several powerful patterns of reasoning about security.

2.5.1 Reachability and Object Graph analysis

The basic rule for reasoning in capability systems is that a capability can only be accessed from dynamic scopes to which it was passed. In order to bound the possible risk posed by bugs or malicious behavior in any given part of the program, we can consider the graph of objects reachable from the scope at that program point. This can be determined

[4] Elaborations on this idiom can handle more complex use cases, e.g., where the original code also contains one or more `catch` blocks, and when the original exception signature must be maintained. This idiom does not require duplicating code.

by constructing a graph with a node for each object in the program, and an edge for each field pointer. The authority of a point of execution is bounded by the subset of the graph reachable from the variables in scope at the time.

The graph generated by this technique is very conservative, as it ignores the behavior of classes on the path from the variables in scope to the capabilities reachable from them. A substantial advantage of object-capability languages over basic capability systems is the ability to attenuate authorities via encapsulated reference monitors, which allow only partial access to the ultimate capability. In practice, programmers can incrementally refine the crude bounds obtained through naive reachability analysis by taking into account the behavior of classes along this path. We have found that, in well-designed systems, this style of reasoning is effective at enabling code reviewers to focus their attention on a small fraction of the code at a time. We made use of it during a security review of Waterken when checking the capabilities the infrastructure makes available to application code.

2.5.2 Leveraging Static Typing

Type safety, as provided by Java and other statically type-safe languages, can also be of use in reasoning about programs and the distribution of authorities to parts of a program.

Because the capabilities granted to a method are specified by its arguments (including any associated instance or enclosing object), the signature of a method serves as a security policy. Since the method can be invoked only with capabilities that satisfy its method signature, it can subsequently obtain access only to capabilities reachable from these arguments, or new objects it can create through public constructors and static methods. Hence, the set of methods exposed by an interface or class can serve as a complete security policy for the objects that implement it, provided that other components of the system are verified to interact with the object solely through this interface. The image decoding example in Fig. 2.2 is an example of this type of reasoning.

When analyzing code of a class to verify it meets its security goals, it is necessary not only to examine the textual code of the class itself, but also to understand the behavior of any external methods that it invokes. This often requires identifying what classes those method invocations might resolve to. Static method calls are easy: static methods cannot be overridden, so it each static method maps directly to a specific implementation. The static method's documentation can be consulted and its source code can be examined. In comparison, instance methods are more difficult, as they can be overridden. There are two basic approaches to justify trust placed in instance methods: based on the object's provenance, or based on its type.

1. *Provenance.* In the first approach, we justify relying upon the behavior of methods of an external object based on the external object's provenance or origin. For example, an object that the code constructs itself is known to have behavior consistent with its known concrete type. Provenance-based reasoning can also arise from transitive trust relationships. For example, consider an object O that calls a method on object P that

it trusts to return an object Q with specified behavior. The provenance of Q then makes it safe for O to invoke its methods regardless of its type.

2. *Type.* If we know the declared type of the external object, then in some cases this typing information makes it possible to rely upon the behavior of that object.

   The simplest example of using trusted types to ensure desired behavior is calling an instance method on an object belonging to a final class. Like static methods, it is in this case possible to map the method called to a single implementation that can be reviewed. Regardless of the origin of the object, the code being executed is known to come from the declared class. For example, because the `String` class is final, code that uses strings can rely on `String` objects to fulfill their specified contracts; it does not need to defend against some maliciously-defined object that impersonates a `String` but misbehaves in a devious way to violate security or privacy.

   Instance methods from non-final classes are trickier. In general, it is not possible to guarantee behavioral properties of methods belonging to such a class C, as one could be dealing with an arbitrary subclass which may fail to meet the documented semantics of the original declarer of the method. In order to avoid this risk, it is necessary to prevent arbitrary subclassing of C. One way to achieve this in Java is to define C and its subtypes to have only package-scope constructors, but no public constructors. To allow instantiation by code outside the package, these classes can provide public factory methods. This ensures that C can only be subclassed by the bounded set of classes in its own package, permitting reasoning about the behavior of objects with declared type C, even if their origin is not trusted.

If the programmer adopts a particular style of programming, called *capability discipline*, Joe-E supports reasoning about the privileges granted by an object based upon that object's declared type. Capability discipline proposes that the documentation for each type should specify the authority that may be granted by instances of that type. For instance, Joe-E's `File` object conveys authority to a single file on the filesystem (or, in the case of directories, a subtree of the directory hierarchy); passing a `File` to another party will enable them to access the specified file, but not (say) send network packets or erase the entire hard drive. When a type T is non-final, the documentation for the type T should specify an upper bound on the authority granted by instances of T or any of its subtypes. If code reviewers check that subclasses of T never yield more authority than this, then we can use the type system to upper-bound the authority passed across an interface: if a method $m()$ accepts a parameter of declared type T, we can conclude that this parameter will not yield more authority than that specified in T's documentation. Similarly, if a method has return type T, we can conclude that this method's return value will not yield more authority than that specified in T's documentation. We follow this pattern in the Waterken server and have found that it is helpful for reasoning about the authority that a type can convey.

2.5.3 Defensive Consistency

Reasoning about the security of a program is difficult if understanding its security properties requires comprehending the entire program all at once. The task is greatly simplified if it is possible to analyze the program in a modular fashion, one piece at a time. The easiest way to do this is to decompose the program into a number of trust domains, and for each domain determine what invariants it aims to maintain, and which invariants it relies on from other classes. In Joe-E, a trust domain would normally correspond to a single object, or perhaps a small collection of objects. Normally, domains interact following a client-server metaphor: domain D might provide service to clients $C_1, \ldots, C_n$. The standard approach to modular analysis in the program verification literature suggests we verify that (1) D provides correct service to its clients, assuming that all its clients meet D's documented preconditions; (2) each client C_i establishes D's documented preconditions. This allows us to analyze the code of D on its own, then separately analyze the code of each client C_i on its own, without having to mentally consider all possible interactions between them. However, this approach requires us to verify that every client C_i meets D's preconditions, which may not be possible in an open world or where some clients may be malicious.

Defensive consistency is a relaxation of this concept [43, §5.6]. To show that D is defensively consistent, we must show that D provides correct service to every client that meets D's documented preconditions. Note that if one of D's clients, say C_1, fails to meet D's preconditions, then D is under no obligation to provide correct or useful service to C_1, but D must still provide correct and consistent service to its other clients $C_2, \ldots, C_n$ (assuming they do meet D's preconditions). Thus, D must maintain its own invariants, even if one of its clients behaves maliciously. A defensively consistent domain can be safely used in contexts where some of its clients may be malicious: its non-malicious clients will be protected from the misbehavior of malicious clients.

Defensive consistency confines the malign influence that a single malicious or compromised component can have. Without defensive consistency, verifying security becomes harder: if domain C acts as a client of a non-defensively consistent abstraction A, then verifying the correctness of C requires us to verify that no other client of A is malicious, which may be difficult and may require reviewing a great deal of additional code. Thus, defensively consistent components support least privilege and reasoning about security.

2.5.4 Immutability

Joe-E's support for immutable types (§ 2.4.2) facilitates defensively consistent programming. When immutable objects are passed between trust domains, immutability provides guarantees both to the sender and recipient domains. The sender is assured that the recipient cannot modify the passed object, and thus the sender can continue to use the same object internally without having to make a defensive copy to guard against corruption of its internal state. Also, passing an immutable object conveys no capabilities aside from the data contained in the passed object, which helps the sender avoid inadvertent capability leakage.

```
class C {
  private Object signers[];
  public Object[] getSigners() {
    return signers;
  }
}
```

```
class C {
  private ImmutableArray<Object> signers;
  public ImmutableArray<Object> getSigners() {
    return signers;
  }
}
```

Figure 2.8: At left, an example of a classic Java vulnerability: a malicious caller to `getSigners()` could mutate the internal state of the class, due to the failure to make a defensive copy of its `signers` array. At right, a natural way to write this code in Joe-E is secure without defensive copies, thanks to the use of immutable types.

The recipient is also protected from unexpected mutation: it can store the immutable object as part of its internal state without fear of interference from modifications performed by any other code that has access to the same object. Thus, Joe-E's immutable types eliminate the need for defensive copying at the sender or the receiver. For instance, Fig. 2.8 shows a classic Java vulnerability and how Joe-E's immutable types eliminate the vulnerability pattern.

2.5.5 Attenuation of Authority

In order to achieve least privilege, it is helpful to be able to easily attenuate the authority provided by a capability. This refers to being able to take a capability to a resource and derive from it a less-powerful capability to the resource that has only a subset of the privileges of the initial capability. One example of this would be a new object that wraps the old object and acts as a reference monitor on operations performed on the encapsulated object. While this is supported in a general-purpose and flexible way by defining classes that act as reference monitors, we suggest that class libraries and type hierarchies be designed to facilitate easier use of common attenuation patterns.

For example, in Joe-E a file object represents the ability to access a particular file, or if it is a directory, any of its subdirectories and their files. Joe-E directory objects provide a method to obtain a capability to any of the files or directories contained within them. This allows one to create an attenuated capability that allows access to a smaller part of the filesystem; a program can be given a capability to a large directory, but have the ability to delegate only a portion of this authority to other, less trusted parts of the program. This makes it easy to follow the principle of least privilege. An important requirement to correctly implementing attenuable authority in tree structures like the file system is to avoid methods that retrieve the parent of a node, as such methods would make any node actually give the authority to access the entire tree.

2.5.6 Facets

A client can always create an attenuated version of a capability by defining a wrapper object; however, this places an implementation burden on the author of the client code that discourages the practice of the principle of least privilege. Where the author of an interface can anticipate a useful attenuation of authority, providing it as part of the interface encourages better capability hygiene by all clients.

For instance, Fig. 2.9 shows a typical Java queue interface, followed by a Joe-E queue interface that predefines the attenuated authority to add elements to the queue. The implementation technique for this attenuated authority is called a "facet". A facet defines an additional interface for manipulating state that can also be manipulated via another interface. Whereas a typical object has a single public interface that governs access to its state, an object with facets has many such interfaces. Each of these facets is designed to provide a least privilege interface for a particular kind of client. In this case, the enqueue facet provides permission to add elements to the queue, without the permission to remove elements or to access elements added by other clients of the queue.

Using the facet technique, the author of an object can implement an attenuated capability more economically than a client could, since the state protected by the facet is already within the lexical scope where the facet is defined. This economy of expression makes the facet technique useful even in cases where the attenuation is only of use to one client.

2.6 Implementation

We implemented a source-code verifier for Joe-E as a plugin for Eclipse 3.x. The plug-in supports the development of Joe-E code alongside the use of unrestricted Java. A Java package annotation `@IsJoeE` is used to indicate that a package is written in Joe-E. The plug-in checks every class belonging to such packages and flags any violations of Joe-E restrictions in a manner similar to compilation errors. This package annotation, which is retained at runtime, allows our system to recognize Joe-E code during verification and at runtime via the reflection API.

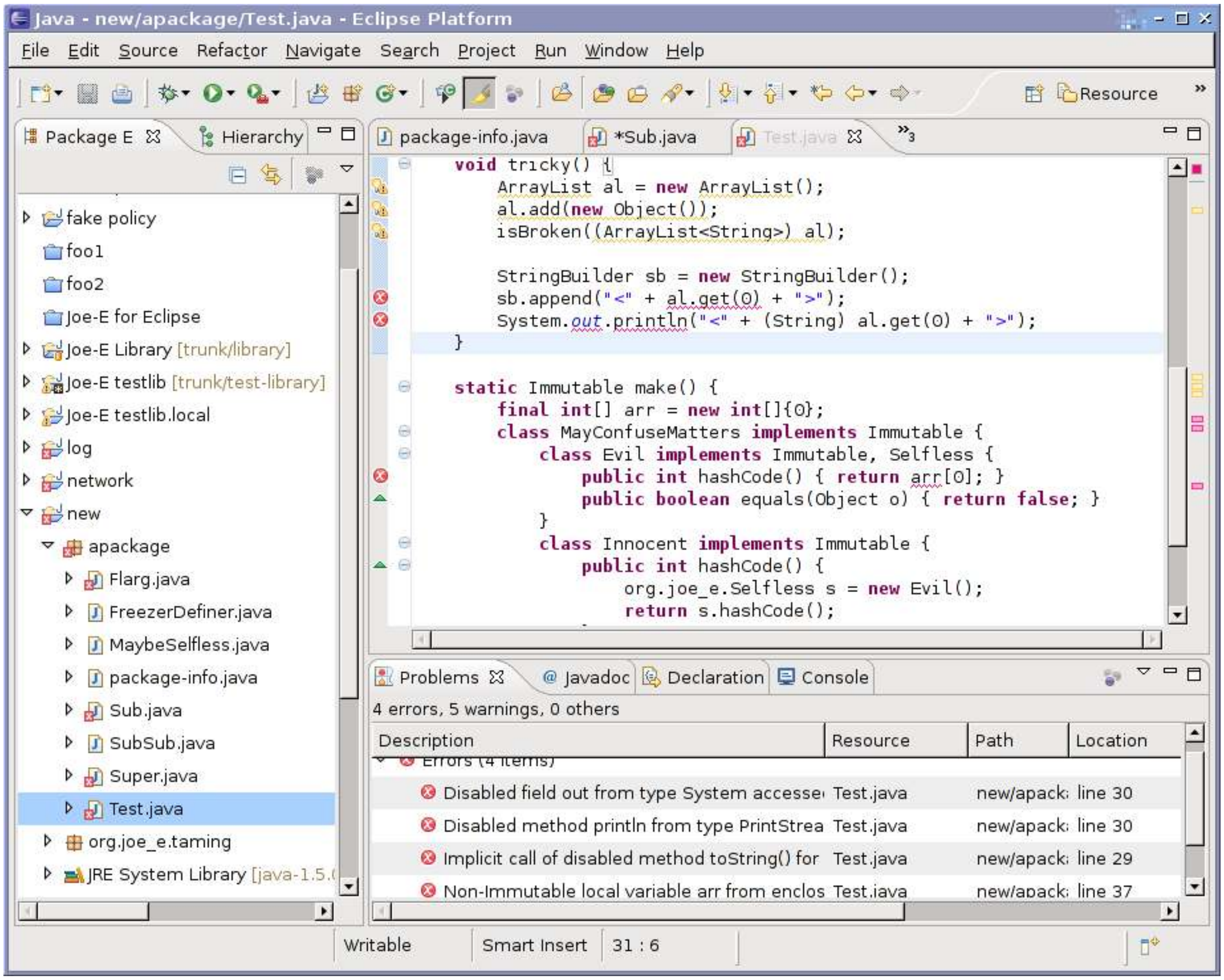
We perform checks on the Java source code rather than on Java class files since the Java runtime subjects bytecode to only a limited set of validation checks, allowing bytecode to do a number of things that Java programs cannot. The expanded semantics afforded to bytecode but not specified by the Java language are unfamiliar and not clearly defined, and thus much harder for a programmer or auditor to reason about.

Working with source code has disadvantages. Since Java source code is higher level than Java bytecode, the verifier must correctly handle a larger number of features, raising the likelihood that an implementation bug in the Joe-E verifier could allow an attacker to sneak something by the verifier. For example, the Joe-E verifier must reason about code implicitly generated by the Java compiler, such as default constructors, string conversions, and enhanced for loops. Our verifier infers the presence of these implicit calls, and checks that only permitted methods and constructors are called. Another complication is that generic type parameters are not type-safe. This complicates inference of which `toString()` method will be invoked by implicit string conversions. While the Joe-E language permits full use of Java generics, our verifier implements a more conservative type check than the Java

```
class Queue {
  public Object dequeue() {
      ...
  }
  public void enqueue(Object o) {
      ...
  }
}

class Queue {
  public Object dequeue() {
      ...
  }
  public void enqueue(Object o) {
      ...
  }
  public Receiver enqueuer() {
      return new Receiver() {
          public void receive(Object x) {
              enqueue(x);
          }
      };
  }
}
```

Figure 2.9: Above, an example of a typical Java queue interface. Below, a Joe-E queue interface that defines an attenuated facet that only supports adding elements to the queue. Easy access to this facet encourages clients to practice the principle of least privilege by delegating only the permission to enqueue, not the permission to dequeue, to those objects that do not need full access to the queue.



```
    void tricky() {
        ArrayList al = new ArrayList();
        al.add(new Object());
        isBroken((ArrayList<String>) al);

        StringBuilder sb = new StringBuilder();
        sb.append("<" + al.get(0) + ">");
        System.out.println("<" + (String) al.get(0) + ">");
    }

    static Immutable make() {
        final int[] arr = new int[]{0};
        class MayConfuseMatters implements Immutable {
            class Evil implements Immutable, Selfless {
                public int hashCode() { return arr[0]; }
                public boolean equals(Object o) { return false; }
            }
            class Innocent implements Immutable {
                public int hashCode() {
                    org.joe_e.Selfless s = new Evil();
                    return s.hashCode();
```

4 errors, 5 warnings, 0 others

Description	Resource	Path	Location
Errors (4 items)			
Disabled field out from type System accesse	Test.java	new/apack:	line 30
Disabled method println from type PrintStrea	Test.java	new/apack:	line 30
Implicit call of disabled method toString() for	Test.java	new/apack:	line 29
Non-Immutable local variable arr from enclos	Test.java	new/apack:	line 37

Writable | Smart Insert | 31 : 6

Figure 2.10: The Joe-E Verifier for Eclipse

compiler to ensure that tamed-away `toString()` methods will not be invoked [39, § 4.10].

We have tamed a subset of the Java libraries that is small, but sufficient for writing useful programs. It would be useful to increase the set of tamed classes, as much of Java's utility derives from its rich class library. While it is possible for Joe-E application developers to add additional classes and enabled methods to the taming database, determining whether classes are capability-safe is unfortunately a high-risk process that requires careful attention and awareness of possible pitfalls. There is an opportunity for future work in tools that simplify and improve the safety of this process.

As mentioned above (§ 2.4.2), some important functionality cannot be made safe by taming alone. Joe-E provides safe wrappers for the filesystem, for Java reflection and proxying APIs, and for locale-independent character encoding and decoding routines.

The Java language only includes mutable arrays. Joe-E also provides read-only array types for use as collections of data: `ConstArray<T>` is a read-only array of possibly-mutable objects, and `ImmutableArray<T>` is a read-only array of immutable objects. We need multiple classes because generic type parameters are not type-safe in Java: for instance, an object of type `ConstArray<String>` isn't guaranteed to actually contain strings, and thus might not really be immutable. A runtime check at creation ensures that all elements in an `ImmutableArray` are in fact immutable. One alternative we considered was to use a type annotation to distinguish between mutable and immutable arrays in Joe-E source code, and enforce the annotation in the Joe-E verifier. While this approach might work for simple cases, the lack of runtime information would greatly complicate serialization and probably make reflection infeasible for immutable arrays.

The Joe-E verifier and libraries are released as an open-source project, available at `http://www.joe-e.org`. To increase our assurance in the implementation's correctness, we have implemented a suite of over 300 unit tests, including several for each of the Joe-E language restrictions, covering as many corner cases as we could devise.

2.7 Conclusions

Object capabilities are a promising approach to building software systems that provide reliable security properties and are easier to audit and safely extend. In this work, we have shown that the advantages of object-capability systems can be achieved with moderate changes to a popular type-safe object-oriented language. Defining a subset allows one to reason about sophisticated security properties of a program in a familiar language, obtaining the benefit of stronger guarantees about what a program is able to do while leveraging existing tools and programmer expertise. We anticipate that these techniques will be useful for developing security-critical and other software, as the industry moves beyond ad-hoc responses to specific attacks toward the construction of verifiably robust, trustworthy software.

Chapter 3

Verifiable Functional Purity in Joe-E

This chapter is based on a paper [21] coauthored with Matthew Finifter, Naveen Sastry, and David Wagner, presented at 15th ACM Conference on Computer and Communications Security in October 2008.

3.1 Introduction

Critical real-world programs often have high-level security and privacy requirements expressed in terms of reproducibility, invertibility, non-interference, or containment of untrusted code. We would like to verify these properties given the programs' source code, but this task is difficult in the languages commonly used to write real-world programs. These imperative languages permit side effects and data dependencies that are difficult to reason about. Purely functional languages, in which methods obey the semantics of mathematical functions, make reasoning about effects and information flow easier, but have not gained the popularity and code base of more traditional imperative languages. We present a technique for implementing verifiably *functionally pure* methods in imperative languages. To be functionally pure, a method must satisfy two critical properties[1]:

First, it must have no side effects. For a computational method to be free of side effects, its execution must not have any visible effect other than to generate a result. A method that modifies its arguments or global variables, or that causes an external effect like writing to disk or printing to the console, is not side-effect free.

The second property is functional determinism: the method's behavior must depend only on the arguments provided to the method. The method must return the same answer every time it is invoked on equivalent arguments, even across different executions of the program. A simple example would be a method to upper-case a string: every time it is given a string containing the word "foo", it will return a string containing "FOO". Many methods do not satisfy this criterion, including ones whose behavior depends on the time of day, the amount of free memory, or whether a specific flag was present on the command line.

[1] More formal definitions of these two properties are provided in Section 3.3.

Electronic voting machines are one important application with a number of security requirements amenable to enforcement using functional purity. These machines are single-purpose computers running custom software designed to allow the voter to select his or her preferred candidates and to record the selections. Given the importance of these machines to our democracy and concerns over their trustworthiness, it would be useful if we could prove aspects of their operation correct.

For example, we argue that voting machines should be designed to ensure that each voter's voting experience will be a deterministic function of the ballot definition and that voter's actions. For a particular set of voter actions, the system should always present the same screens and record the same selections, independent of previous voters' interactions with the voting machine. Leaking any information about previous sessions could violate earlier voters' privacy and could create a conduit for a malicious voter to interfere with subsequent voters. Also, voting sessions should have no side effects; their only legitimate effect should be to return the voted ballot. Functional purity can help verify these security properties.

As another example, voting machines must serialize and possibly encrypt the voter's selections when writing them to stable storage. This data will be read and tallied at a future date, likely on a different machine. In order for the voter's choices to be counted as they were cast, we must be certain that the reconstituted votes will match the originals. We propose a fail-stop check on the encoding process: the machine writing the data should decode the serialized output and verify that it matches the original vote selection data structure. If the decode method is deterministic, this check ensures that this data structure will be correctly reconstructed later when the votes are counted. If the serialization and deserialization routines are also side-effect free, they can be removed from the trusted computing base, as the check verifies their correctness as needed.

In general, verifying that a computation will be deterministic and free of side effects is a difficult task that typically requires careful examination of a program's entire source code. Verifying side-effect freeness requires verifying that the computation does not *modify* the state of any parameters or global state and does not affect the outside world in any observable way (e.g., writing to an I/O device). Verifying determinism requires ensuring that the method does not *read* any information that may differ between different calls. Checking the latter property first requires ensuring that anything that is read by the method isn't changed elsewhere in the program. Also, we must ensure that any value read by the method doesn't depend on environmental factors that could differ between executions of the program.

We can see that the concepts of determinism and side-effect freeness are related, in that they both restrict access to state created outside the method. We use a unified approach to achieving both goals, based on object capabilities [43]. Specifically, we introduce and define the concept of *deterministic object-capability languages*, in which the ability to cause side effects and to observe data that varies between executions is conveyed by explicit object references that are propagated only by explicit program statements.

A key advantage of our approach is that it supports modular reasoning about purity, side effects, and determinism. In particular, a programmer can tell whether a particular method is pure simply by looking at its type signature. In our system, if all parameter types are immutable, then the method can be guaranteed to be pure. This allows purity specifications to be part of the contract of a method and simplifies the task of reasoning about program behavior. The *body* of a pure method has no additional constraints, permitting wide flexibility in how it is implemented. In particular, pure methods can call impure methods, and vice versa. In short, pure and impure code can easily be mixed; the majority of a program can be imperative, with purity still being enforced where needed.

We briefly describe how the Joe-E subset of Java satisfies the requirements of a deterministic object-capability language, and how it can be used to write methods that can be easily recognized as verifiably pure. In order to evaluate our approach to verifiable purity, we ported three legacy libraries (an AES implementation, serialization logic from an experimental voting machine implementation, and an HTML parser) to the Joe-E subset, and refactored them so that their top-level methods could be verified as pure.

As Joe-E was not explicitly designed to ease migration of legacy code, we found that the task of modifying existing code to satisfy the Joe-E restrictions was at times difficult. Certain recurring patterns account for much of this difficulty; code that avoids these patterns is much easier to port. Refactoring methods so they could be verified pure was generally harder than just porting to the Joe-E subset, and sometimes required changes to data structures and interfaces. We therefore recommend our approach primarily for use with new code that is designed with this approach to purity in mind.

We view the contributions of this work as follows:

- We enumerate several applications where the ability to verify that particular blocks of code are pure makes it easy to verify interesting high-level application-specific properties.
- We describe a class of imperative programming languages in which it is easy to verify purity.
- We introduce Joe-E's enforcement of determinism and we show how this enables verifiable purity in Java.
- We share our experience refactoring legacy codebases so that they can be verified as pure, thus attaining useful security guarantees.
- Based on this practical experience, we identify programming patterns that are well-suited to writing verifiably pure systems as well as anti-patterns that make this task difficult.

3.2 Applications

We argue that functional purity has many applications in security and reliability. Purity is a helpful tool for building more modular programs that are easier to reason about, and this makes it easier to verify many kinds of security properties. Languages and programming idioms that make this property easy to achieve and verify may be of benefit to programmers, especially those aiming to write maintainable, auditable, and understandable code.

3.2.1 Reproducibility

Consider the following scenario, inspired by [2]: Mallory generates a PDF file containing a contract for Alice to electronically sign. Mallory constructs this PDF file so that its displayed content depends on the system date. When viewed in January, the contract says that Mallory will pay Alice $100; in any other month, the contract says that Alice will pay Mallory $1,000. Suppose Alice reads and electronically signs the contract on January 1, and returns the signed contract to Mallory. On February 1, Mallory presents the signed contract to a judge, and the judge orders Alice to pay Mallory $1,000.

The problem is that the computation that renders the text is not deterministic. The behavior of the PDF viewer depends on other factors aside from its input, the bits of the document file. This attack could not succeed if the PDF viewer's computation was a pure function of the input file. If we could verify the purity of the viewer, we would be assured that Mallory's attack will fail.

This is an example of a TOCTTOU vulnerability. Whenever we compute a result that is checked, and then recompute it later when it is used, we must be careful to ensure that the computation is reproducible. Pure functions are useful for this, because determinism ensures reproducibility and makes explicit the inputs a computation may depend upon.

Another application is in transactional systems. Suppose we take periodic checkpoints of an application and log all its inputs. If the application is deterministic, then we can recover from crashes: reincarnating the application and replaying from an old snapshot and input trace will always reproduce the same behavior that the previous incarnation of the application followed. This eliminates the need to checkpoint every intermediate state. It also allows a replicated system to transparently fail over to a backup system that is receiving the same stream of input events.

3.2.2 Invertibility

The serialization example given in the introduction is representative of a class of applications that have a matched pair of algorithms (Encode, Decode) for which it is intended that Decode is an inverse of Encode. Specifically, the *inverse property* should hold: for all x, Decode(Encode(x)) should yield some output x' that is functionally equivalent to x. To ensure the original x will be recoverable in the future, this has to hold even if the invocation of Decode takes place at some later time on a different machine.

Purity helps support *fail-stop* enforcement of this property, in which errors are detected at runtime but before any harmful consequences have taken place. One can test Encode(x) at runtime to ensure that it will be decoded correctly by Decode:

```
y := Encode(x)
abort if x != Decode(y)
```

If Decode is purely functional, its determinism ensures that the check can be performed at any time and will accurately reflect whether the message can be correctly decoded in the future. Also, if Decode is side-effect free, adding this check to existing code won't break the program.

This approach applies to, e.g., serialization and deserialization, encryption and decryption, and compression and decompression. In many such applications, it is better to fail and warn the user than it is to proceed and lose data. If this pattern is used to ensure that all data that is encoded can be recovered, neither the encoder nor decoder need to be trusted correct in order to establish the property that data is never lost or corrupted.

Formally verifying the correctness of serialization and deserialization with static analysis is a difficult task. Serialization and deserialization typically involve walking a (potentially cyclic) object graph, and thus inevitably implicate complex aliasing issues, which is known to make static analysis difficult. Therefore, purity seems better-suited to this task than classical approaches.

Deterministic functions can also be used for enforcement of more complex functional relations than invertibility. The exokernel Xok's stable storage system uses what the authors call a UDF (untrusted deterministic function) for each type of metadata disk block (e.g., inodes) to translate the set of blocks referenced by the metadata into a form recognized by the kernel [30]. The determinism of this function allows Xok to verify that metadata can only claim ownership of the correct set of disk blocks. This is done by verifying, when the metadata is updated, that the set of blocks claimed by the new metadata is the same as the set claimed by the old metadata with the intended change applied. This mechanism is only sound if the metadata decoding function is known to be deterministic.

3.2.3 Untrusted code execution

Purity gives us a way to execute untrusted code safely: we first verify that the untrusted code is pure, and then many useful privacy and security properties will follow. In particular, the lack of side effects means that the pure, untrusted computation cannot violate the integrity of the rest of the program it interacts with[2], so pure code inherently executes in a sandbox.

Purity can also be used to structure programs in a way that reduces our reliance upon the correctness of some subset of the code. If we use a pure method to process (possibly malicious) data from an untrusted source, and if the output from the pure method is no more trusted than its input, the method doesn't need to be trusted to defend itself from malicious data successfully. Even if a malicious input is able to somehow subvert the proper operation of that method, at worst it can only influence the result of the pure computation; it cannot harm the proper operation of the rest of the program.

Bernstein's discussion of address-extraction code in `sendmail` [6] illustrates these ideas well. The address-extraction code is responsible for parsing an email message and extracting an email address from a particular header. At one point, this code contained a remotely

[2] Untrusted code can still deplete resources or fail to terminate. Limits on resource usage or looping would be needed if denial of service is a concern [57], but that is beyond the scope of this work.

exploitable vulnerability that allowed an attacker to gain root by taking control of `sendmail`. Bernstein proposed an alternate architecture:

> Suppose that the same address-extraction code is run under an interpreter enforcing two simple data-flow rules:
>
> - the only way that the code can see the rest of the system is by reading this mail message;
> - the only way that the code can affect the rest of the system is by printing one string determined by that mail message.
>
> The code is then incapable of violating the user's security requirements. An attacker who supplies a message that seizes complete control of the code can control the address printed by the code—but the attacker could have done this anyway without exploiting any bugs in the code.

We note that Bernstein's two conditions are exactly determinism and side-effect-freeness, so implementing the address-extraction code as a pure method would provide the desired security benefits.

Determinism allows us to bound what information a pure method can read—in particular, the method can only observe the value of objects that are reachable from one of its arguments, but cannot gain any information about any other data in the program. Moreover, deterministic code cannot listen on covert channels: for instance, any differences in behavior due to timing information or resource limits would violate the determinism properties. This ensures that the untrusted method cannot spy on any sensitive program state that was not explicitly provided to it.

Purity also limits the untrusted code's ability to leak sensitive information to others through overt channels. It can communicate to others only through its return value (or thrown exceptions) and its resource consumption. However, it can transmit over a timing- or resource-based covert channel to a receiver that is not pure. For instance, we might download an untrusted tax calculator and verify that it is pure before executing it. Then even if we type our salary into it, it cannot leak our salary to others directly, though it may be able to leak our salary through a covert channel.

Purity may also be useful for application extensions and plugins. For example, consider an image viewer that, out of the box, supports only a handful of image formats. It might allow installation of a plugin for viewing images in a different format only if that plugin is written as a verifiably pure function that, given the contents of an image file, returns a bitmap to be displayed by the image viewer. Once verified as pure, any such plug-in could be downloaded and executed safely; it cannot gain any information about other private information stored on the system, nor can it corrupt the state of any other part of the program.[3]

[3] It should be noted that purity does not eliminate all threats that the plugin could pose to the program. The invoker of a pure plugin method must still ensure that only appropriate data is passed to the plugin and defend against unexpected return values from the plugin.

3.2.4 Building robust systems

Pure methods are also helpful for writing trustworthy security-critical code that mediates between untrusted components.

For the purposes of preserving application integrity, pure methods are always safe to expose to untrusted code. Their functionality could always be duplicated by the untrusted code itself, so they cannot pose an additional threat. Pure methods may still be part of the TCB, but only if their behavior is trusted for semantic correctness, not because the method is granted privileged access to program internal state. This is a consequence of the lack of side effects. It is possible, however, that specific instances of immutable objects, and thus their associated pure methods, might convey confidential or malicious information. One must still be careful about data flow.

A pure method is automatically "defensively consistent" [43, § 5.6], provided that in the absence of malice it provides correct service to individual clients and provided that each invocation of the pure method processes information from only a single client. (An object that serves multiple clients with independent interests is defensively consistent if, even when one client violates its preconditions, it continues to comply with its specification for other clients that satisfy its preconditions.) Defensive consistency ensures that one malicious client cannot attack other clients who may rely upon the same pure method.

Bernstein presents an example in which a pure `jpegtopnm` converter receives a JPEG image from the network, decompresses it, and outputs a raw bitmap of the image to a user [6]. The "client" here is the sender of the image, who as an arbitrary remote party, is untrustworthy and may wish to corrupt or disrupt processing of images from other sources. A defensively consistent implementation would thwart such attacks by continuing to provide correct conversion of all images originating from other senders, even if one sender sends malformed data with the intent to exploit the converter. The purity of the converter ensures defensive consistency because it allows one to know that each image is converted independently from any others, preventing a malformed image from affecting the processing of other images.

3.2.5 Bug reduction

Pure functions can help us eliminate certain classes of bugs. Of course, anything that reduces the number of bugs in security-critical code helps security.

A pure computation is automatically thread-safe, requiring no locks, and can always be run in parallel with other computations without risk of interference. Determinism guarantees that concurrent operations cannot disrupt its correct execution, and the lack of side effects means that it cannot disrupt other computations.

Reproducibility is particularly useful when debugging and testing applications. It is often the case with modern applications that bugs are discovered in the wild rather than during testing, due to novel configurations that were not considered during testing. In many cases, it can be difficult to reproduce the bug as there are a number of hidden variables that cause the behavior of a program to differ between runs. If a method is pure, any failure of the method will be reproducible given the same well-defined, bounded set of inputs. This known set of data can be collected and used to reproduce the bug for the developer, who can then fix the program.

Deterministic functions can also make testing more effective. If the computation is deterministic, we only need to cover any particular input once; on the other hand, if the computation is nondeterministic, it may conceal bugs that trigger nondeterministically, so it is difficult to know whether we have tested all possible behaviors. For instance, Bernstein cites dealing with nondeterministically triggered error cases as one challenge in testing qmail, and proposes that testing would have been easier if the code had been structured as a purely functional computation plus a simple wrapper that interacts with the environment (so that the wrapper can be easily mocked in testing) [6]. Verifiable purity would enable developers to check that this discipline was followed correctly and preserve it under maintenance.

3.2.6 Assertions and Specifications

It is widely accepted that assertions should be side-effect free. If evaluating the assertion condition causes no side effects, a program that always satisfies the assertion will behave the same way whether the assertion is enabled or disabled. This restriction could be checked by a lint-type tool that would warn about potentially impure assertions.

In applications where assertions are used for debugging, it is also helpful to know that the assertion condition is deterministic. If a deterministic assertion succeeds, we know that it will not fail on another run of the program due to dependence on seemingly unrelated state or nondeterministic behavior of the underlying platform. Sometimes, programmers use assertions specifically to check for and abort in the face of incompatible platform configurations. In deterministic languages like Joe-E, however, platform-specific behavior is mostly hidden from the program, which would reduce the need for this pattern.

Some specification languages allow methods to have pre- and post-conditions that are defined using the same language as the code, and these conditions may call other methods. For instance, in JML, a specification language for Java, specifications are only supposed to call methods that are “pure.” JML’s notion of purity forbids side effects but does not require determinism and places no restrictions on what state the method’s behavior may depend upon [34]. Since JML specifications can be compiled to assertions and checked at runtime [13], the purity requirement is intended to ensure that these assertions do not change the program’s semantics.

We argue that pre-conditions, post-conditions, and object invariants should be deterministic as well as side-effect free. When methods are used in specifications, the specification cannot be considered fully defined unless the method is deterministic. In particular, if the requirements on the method are predicated on external state whose value changes from invocation to invocation, it will not be possible to statically verify that the method satisfies its contract. While JML’s restrictions on side effects in specifications may suffice to prevent runtime enforcement from changing the semantics of the program, static checking is more difficult than it would be if specifications were functionally pure.

We have not implemented a tool for checking the purity of assertions, but verifying the purity of some assertions would be a straightforward extension of our techniques for methods. There are some common patterns for assertions and specifications (such as those that require executing methods on non-immutable objects) for which our approach may not be applicable.

3.3 Definitions

Our definition of functional purity derives from the concept of a mathematical function, a well-defined one-to-one mapping from inputs to outputs. We consider a method in a program to be functionally pure if and only if it is both side-effect free and deterministic.

3.3.1 Side-effect freeness

A method is *side-effect free* if the only objects that the method ever modifies are created as part of the execution of the method. This definition permits the method to create and modify new objects, any subset of which may be reachable from its return value, but does not permit it to make any change that would be observable from outside the method.

In addition to this linguistic notion of side-effect freeness, we also require that a pure method not cause any side effects outside the language environment, with the exception of resource consumption (memory and CPU cycles). For example, it must not write to files, communicate over the network, or print to the console. This is necessary to soundly constrain the effects of untrusted code.

3.3.2 Determinism

A mathematical relation is considered a function if each distinct input is associated with a single specific output. Any two evaluations of a mathematical function with the same inputs will give the same result. This result depends deterministically on the inputs, and nothing else. Our determinism requirement for functions in a program is analogous. We want any two calls with equivalent arguments to a pure function to give the same result. The result must depend only on the arguments and not on other global or thread-local state such as the current time or the stack trace.

For numbers and other mathematical constructs, there is a well-accepted canonical definition of what it means for two sets of function inputs to be the same, namely mathematical equality for each argument. We need a similarly precise definition for equivalence of arguments in programs. Element-by-element equality works for value types, but pointers and references raise questions. Should two calls of a method be equivalent if their arguments have the same numeric values but different aliasing relationships? If they have the same aliasing but reside at different addresses in memory?

There is not a single obviously right answer to these questions. Determinism is thus a *parameterized* property: given a definition of what it means for arguments to be equivalent, a method is deterministic if all calls with equivalent arguments return results that are indistinguishable from within the language. The determinism guarantee is only useful for calls whose arguments are equivalent according to the definition of equivalence. The definition should make semantically equivalent invocations (those that look equivalent to the programmer or auditor) have equivalent arguments.

If the criteria for equivalence include memory layout information such as the concrete addresses of pointers, invocations will essentially never be equivalent, and the determinism guarantee will be meaningless. For types that serve purely as collections of immutable data, we can avoid addresses completely by comparing purely by value, not by reference. This prevents pointer aliasing from causing seemingly equivalent invocations to be distinct. For example, consider a Java method that concatenates two strings. If aliasing information is included, `concat(str, str)` will not be equivalent to `concat(str, str.clone())` because in the first case both arguments refer to the same object, and in the second case they refer to different objects. For other types (e.g., graph nodes) that are generally compared by reference, aliasing information may be important to include. Our equality definition (§3.4.1) excludes memory layout details but includes aliasing relationships for types where identity provides the semantic notion of equality.

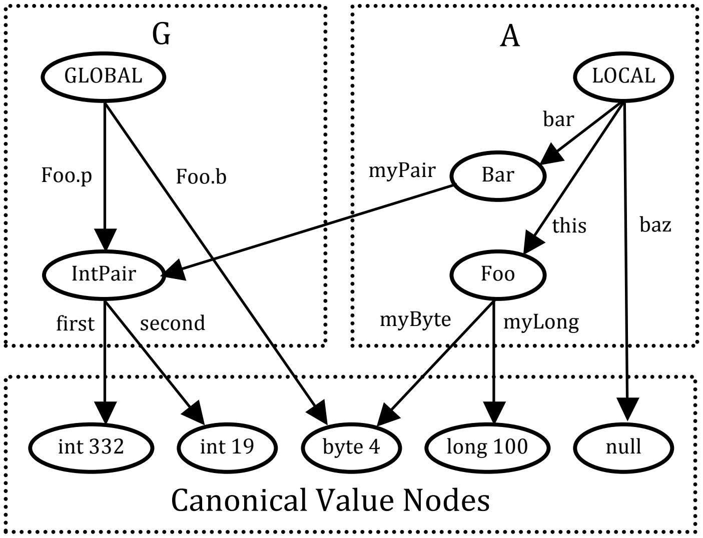
3.4 Approach

Our approach to purity is based on leveraging the properties of a *deterministic object-capability language*, i.e., an object-capability language that (a) has no nondeterministic language primitives and (b) requires a restricted capability for any access to nondeterminism.

An object-capability language [43] is one with the following properties:

- all state that can be communicated between methods is stored in objects
- all objects can only be accessed by references
- references can only propagate by being passed as arguments or being stored in a shared object
- references are unforgeable (for instance, the language must be memory-safe, and it must not permit unsafe casts)
- access to references is strictly limited by lexical scoping of variables and transitive reachability of references

In such a language, references serve as capabilities, and capabilities can be granted only by explicitly passing references. For these properties to be effective in restricting code's effects, the global scope must not contain any capabilities to affect program state or the outside world. In other words, using methods and objects in the global scope it must not be possible to have any effect on mutable program state or external effects aside from resource consumption.



```
public class Foo {
  public static byte b;
  public static IntPair p;

  private long myLong;
  private byte myByte;

  public void foo(Bar bar, Baz baz) {
    // ...
  }
}

public class Bar {
  private IntPair myPair;
  // ...
}

public class IntPair {
  public int first, second;
}
```

Figure 3.1: An example of an object graph and corresponding Java class definitions.

In a deterministic object-capability language, the observable global state must never change and must be the same on every execution. Any data returned from global methods is considered to be part of the observable global state. This means, for example, that no globally accessible methods can provide the time of day or state of the filesystem, even though this data is not explicitly stored anywhere in the global scope. A method's view of global state will thus be the same every time it is invoked, so globals can effectively hold only compile-time constants. Then, since the only variables in a method's scope are globals and arguments, any variation in the method's behavior can always be attributed to differences in its arguments.

From §3.3.2, we must specify when function arguments are equivalent. We do so in the following section, in which the concrete arguments to a method invocation are considered a set of named references. For an instance method, the implicit reference to the target object is treated as an argument named "this".

3.4.1 Equivalence of reference lists

At a high level, we consider two sets of named references equivalent if their reachable object graphs (including values, types, and aliasing relationships) are isomorphic. Figure 3.1 gives an example; the rules for its construction follow.

Many object-oriented languages include both *reference types* and *value types*. Objects of reference type have an identity distinct from their value. The language distinguishes between references that point to the same object and those pointing to different objects with identical contents. Any type with mutable fields is by necessity a reference type, as changes to one instance will not affect another, but immutable objects can also be reference types if the language provides a way to test for object identity (such as Java's `==` operator). In contrast, value types can be compared only by value; there is no other notion of identity for these types. In Java, the primitive types (`boolean`, `char`, and the integer and floating point types) are value types, whereas `Object` and all its subtypes (including arrays) are reference types.

For simplicity in the following formal definition, values are represented in the object graph as references to canonical instances. One such instance exists for each distinct value of each value type. Similarly, we treat null pointers as references to a single canonical null object that belongs to every non-primitive type.

Let G be the set of named global (static) variables in the program. Let A be a set of named object references (such as the arguments to a method). We then define the *reachable object set* $A*$ corresponding to the set A as the transitive closure of objects reachable by following references from all fields of the objects pointed to by $A \cup G$. (The global state of the program will be the same across all executions, so the set of objects reachable from G won't change. This portion of the graph is only included in order to represent all observable aliasing relationships.)

The *object graph* for the reference set A is then constructed as follows: We create special nodes labeled *Global* and *Local*. We construct a canonical node for each primitive value in the program, in addition to a canonical node for `null`. For each reference-type object in $A*$, we construct a node labeled with its concrete type. For variables in G or A, we add edges originating in *Global* and *Local* respectively. These edges point to the node representing the object pointed to by the variable and are labeled with the variable's name. For each field of each object in $A*$, we draw a directed edge from the node holding the reference to the node representing the referenced object (or canonical value), labeled with the field name.

Two sets of object references are considered equivalent if they result in identical object graphs (nodes and edges with the same labels). Note that the object graph reflects all aliasing relationships between non-value objects reachable from the set A and from global variables.

3.4.2 Immutability

We also need the language to provide support for types that are verifiably immutable. An object is immutable if its state, and the state of all objects reachable from it, can never change during its lifetime. Immutability is transitive: all objects reachable from an immutable object must themselves be immutable. To statically verify purity, we need the language to provide some way to verify that a type `T` is immutable, i.e., that every instance of `T` will be immutable.

Objects that represent a capability to affect or observe external state must not be considered immutable, even if their explicit state in the language is immutable.

3.5 Pure methods

Languages that meet our requirements (§ 3.4) make verification of purity easy. The condition is simple: If all parameters to a method (including the implicit `this` parameter) are statically declared to be of an immutable type, then the method is pure.

A pure method cannot cause any side effects, because it will never be able to obtain a reference to a shared mutable object. The only mutable objects it can access are those that it creates itself, but any changes it makes to them are not side effects since they will not be visible outside the method's execution. The method is also prevented from causing any effects external to the language runtime, as all such effects would require an appropriate capability. Such capabilities are not considered immutable and thus are unavailable to methods with only immutable arguments.

The only state observable to the method is that in the global scope and reachable from its arguments. Since the global scope is constant (essentially determined at compile time), the only varying state it can observe is from its arguments. As immutable arguments cannot provide a view of external state, the only observable state that immutable arguments can provide is captured in the definition of reference-list equivalence above (§ 3.4.1). For a language without any inherently nondeterministic constructs, ensuring that two invocations of a method start in indistinguishable states is sufficient to ensure that they will terminate identically.

In our approach, purity is part of the contract of a method: we can verify that a method is pure simply by examining its type signature. This is powerful, because it means that we do not need to inspect the implementation of the method or of other code that it might call. Internally, it may make use of mutating operations and impure methods. Any operation expressible in the language, no matter how much internal mutation it contains, can be verifiably pure if a wrapper method is written with a pure interface. We do not need an automated tool to identify for us which methods are pure; instead, programmers can recognize pure methods from their type signatures. An annotation may be used on such methods in order to ensure that they remain pure under maintenance, i.e., that their argument types remain immutable in later versions of the program.

3.6 Implementation

Joe-E is a subset of Java; the Joe-E verifier, implemented as an Eclipse 3.2 plug-in, checks Java source code to confirm that it falls within the Joe-E subset. Any Java program that is accepted by the verifier is also a Joe-E program with the same semantics, but not all Java programs pass the verifier.

```
Integer.getInteger(SYSTEM_PROPERTY)
System.identityHashCode(new String())
string.intern() == string
Integer.newInteger(integer) == integer
```

Figure 3.2: Several methods from the Java library are nondeterministic and callable by all Java code. These are not exposed to Joe-E code.

3.6.1 Side effects and Nondeterminism

The restrictions on globally-available side effects and nondeterminism are accomplished by exposing only a subset of the fields and methods defined in the Java libraries to Joe-E code. The Joe-E language defines a whitelist of fields and methods from the Java libraries that Joe-E code is allowed to use; the Joe-E verifier will reject programs that make reference to any field or method not on the list. We use this mechanism to prevent Joe-E code from calling any method that exposes the ability to observe or modify the environment outside the JVM, provides access to nondeterminism, or allows reading or writing of global mutable state. See Figure 3.2 for examples.

Object identity

In Java, objects have identity: conceptually, they have an "address", and we can compare whether two object references point to the same "address" using the `==` operator. This notion of object identity can expose nondeterminism. As depicted in Figure 3.3, `Object.hashCode()` exposes nondeterminism, which is incompatible with reasoning about the purity of a method solely by examining its type signature. We have the Joe-E verifier forbid calls to methods such as `hashCode()` that expose nondeterministic representations of object identity.

Also, `String.intern()`'s static object cache contains global mutable state that is visible to Java code as a result of object identity (see Figure 3.4). For some types, such as `String`, we sidestep the problems caused by object identity and simplify reasoning by making the type a value type in Joe-E. This is accomplished by prohibiting Java's object identity comparisons on these types: e.g., Joe-E code is not allowed to use the `==` operator on `String`s or other value types. This makes our determinism guarantees more meaningful and more useful, as the programmer does not need to worry about aliasing relationships between variables of these types. Otherwise, a pure method could act differently on two invocations that the programmer might see as identical, but that differed only in aliasing relationships. For example, a string upper-casing method could behave differently if its argument aliases a global variable. In Joe-E, most immutable types are also value types, allowing more intuitive reasoning about equivalence of invocations.

```
boolean randomBit() {
  return (new Object().hashCode() % 2) == 1;
}
```

Figure 3.3: This Java method is not deterministic. It is not legal Joe-E code, because Joe-E forbids calls to `Object.hashCode()`.

```
boolean previouslyInterned(String s) {
  String t = new String(s);
  return t.intern() != t;
}
```

Figure 3.4: This Java method's return value is not a deterministic function of its input. `previouslyInterned("foo")` returns `true` iff some other code has previously called `s.intern()` on a string `s` such that `s.equals("foo")`. This is not legal Joe-E code, because Joe-E does not allow using `!=` on `String`s.

Exceptions

We treat a thrown exception as a form of return value from a method, since the caller can catch and obtain a reference to the thrown object. This makes it challenging to reason about determinism in Java, for two reasons.

First, every `Throwable` object contains a stack trace generated when the throwable is constructed. This operation is not deterministic in the arguments to the throwable's constructor, because the stack trace stored in the throwable depends on the calling context, which is not a function of the constructor's arguments. Any construction of an exception (which can happen implicitly) thus results in an object which in Java would be a source of nondeterminism. We solve this problem by preventing programmatic access to the stack trace. Specifically, Joe-E code is not allowed to call the `getStackTrace()` and `printStackTrace()` methods.

Second, the Java Virtual Machine exposes true nondeterminism when it encounters a condition that causes a `VirtualMachineError` to be thrown. This occurs under a number of exceptional conditions, some of which (like running out of memory) can be triggered by the application. Consequently, virtual machine errors are a source of nondeterministic behavior; see, e.g., Figure 3.5.

This limits the guarantees we can provide: we cannot promise that whether or not a method terminates successfully will be a deterministic function of its arguments, since other conditions (e.g., the amount of free memory) might influence whether it aborts with a `VirtualMachineError`. Instead, we provide the following guarantee: any two calls to a method with equivalent sets of arguments will yield equivalent results, as long as neither method aborts with a `VirtualMachineError`. On the other hand, if one or both calls throw a `VirtualMachineError`, then we promise nothing.

```
int estimateAvailStackSpace() {
  try { return estimateAvailStackSpace() + 1; }
  catch (VirtualMachineError e) { return 1; }
}
```

Figure 3.5: The JVM throws a `StackOverflowError` when the available stack space is exhausted, so this recursive method's return value is nondeterministic.

```
class IntException extends Exception {
    final int data;
    IntException(int data) { this.data = data; }
}

int nondet() {
    try { freemem(); return 0; }
    catch (IntException ie) { return ie.data; }
}

void freemem() throws IntException {
    int shift;
    try {
        for (shift = 0; ; ++shift) {
            new double[1 << shift];
        }
    } finally { throw new IntException(shift); }
}
```

Figure 3.6: `finally` clauses expose nondeterminism.

We mitigate this shortcoming by ensuring that the program will terminate immediately when the JVM throws a `VirtualMachineError`—the error will propagate to the top level and no Joe-E code will execute after the error occurs. To enforce this property, the Joe-E verifier prohibits catching `Error` or any subtype of `Error`. In addition, Joe-E must also prohibit the use of `finally` clauses, as they allow code to execute after (and in response to) a `VirtualMachineError`.

Figure 3.6 shows how to return a nondeterministic value without explicitly catching an `Error` by using a `finally` clause instead. The `freemem()` method tries to allocate larger and larger arrays of `double`s until triggering an `OutOfMemoryError`. This causes the `finally` clause to execute, which then throws an `IntException` that hides the pending `Error`. The `IntException` contains nondeterministic state (how many arrays could be allocated before running out of memory), which is extracted from the exception and returned by `nondet()`.

Fortunately, Joe-E's prohibition of the use of `finally` does not reduce expressivity: Joe-E code can explicitly catch `Exception`, which allows the catching and appropriate handling of any non-`Error` throwable in the Java library.

3.6.2 Immutability

To support reasoning about purity, we want to allow the programmer to write user-defined classes that are verifiably immutable. The programmer can communicate to the Joe-E verifier that class `C` is intended to be immutable by declaring it to implement the interface `org.joe_e.Immutable`. The verifier then confirms that `C` truly is immutable by checking that all its fields are `final` and have a static type that is a primitive or immutable.

Unfortunately, this is not quite sufficient. In Java, code that has access to a partially constructed object O can read final fields of O before they have been initialized, so reading the same object's fields twice might yield two different answers. To prevent this anomaly, Joe-E places several restrictions on all constructors to ensure that a reference to the object being constructed cannot escape from the constructor [39]. The most relevant is that constructors are prohibited from calling instance methods on the object being constructed.

Some reference types in the Java library, such as strings, are observationally immutable but are not declared to implement the `Immutable` interface. The Joe-E verifier handles these classes specially, treating them as if they did implement `Immutable`.

Java arrays are mutable, but are often used in situations where their mutability is not useful or desirable, as they are the simplest representation of a collection of objects. Therefore, Joe-E introduces a class, `ImmutableArray`, for storing an immutable sequence of immutable objects. Specialized subtypes are provided for holding primitive values without having to "box" them into objects; e.g., `ByteArray` holds an immutable sequence of bytes.

3.6.3 Verifying Purity

With our extensions, Joe-E makes it easy to reason about purity. In particular, in a Joe-E program, if all of the parameters to a method (including the implicit `this` parameter) are immutable, then the method is pure. For instance, suppose that we are decompressing a compressed file from an untrusted source. We might design the decompression interface as follows:

```
ByteArray decompress(ByteArray compressed)
```

This function will be pure, so even if the decompression code is buggy or insecure, a malicious compressed file cannot cause it to corrupt other application data structures. This allows us to contain the effect of any security holes in the decompression code.

As another example, suppose that we are building election tabulation software, which reads the contents of memory cards from the voting machines in the field, parses that data, accumulates the votes, and produces a report summarizing the tallies and winners. The parsing, accumulation, and report-generation code might be implemented following this interface:

```
String tabulate(ImmutableArray<CardData> cards)
```

If `CardData` is an immutable data structure holding the data read from a single memory card, this function will be pure. Hence we can be confident tabulation will be deterministically

	Source lines of code		**Num. classes**		**Num. methods**	
	Before	After	Before	After	Before	After
AES	319	276	1	1	9	9
Voting	688	692	25	25	80	79
HTML	12,652	10,848	94	99	965	947

Table 3.1: Basic code metrics for the three libraries used for evaluation, as measured both before and after refactoring.

repeatable, and that the tabulation operation cannot (even if it is buggy or insecure) corrupt other election data.

Some caveats apply. The soundness of our determinism guarantee depends on consistent behavior from the portion of the Java libraries that Joe-E programs are allowed to call. While it is straightforward to block truly nondeterministic library methods, the semantics of some methods differs between Java library releases, including useful methods that are fully deterministic within a particular version. For example, the behavior of many string and character routines has changed between versions to reflect characters added to the Unicode specification. For this reason, our implementation only guarantees reproducibility between executions using the same library version. Since Joe-E does not currently require all assertions to be pure, our determinism guarantee is also predicated on whether or not assertions are enabled. In Java, results of floating point operations may differ between JVMs, which may cause code to have platform-dependent behavior.

3.7 Evaluation and Experience

Our approach is intended primarily for programmers developing new code in Joe-E with verifiable purity in mind. Since Joe-E is intended to be as familiar as possible to Java programmers, we wanted to understand to what extent our approach would require Java programmers to change the coding style they are used to. We chose three Java libraries and retrofitted them (a) to pass the Joe-E verifier and (b) to have verifiably pure methods and resulting security properties. The refactoring was performed by a programmer who had no prior experience using Joe-E or any other object-capability language.

We give a detailed account of our experience, for three purposes: (1) to give the reader a sense of the type and magnitude of changes that were necessary, (2) to understand the programming patterns that could potentially act as a barrier to the adoption of our system, and (3) to evaluate the strengths and limitations of our approach to verifiable purity. See Table 3.1 for the three applications we analyze. (We used the Eclipse Metrics Plugin [63] for all code metrics.)

3.7.1 AES library

Motivation

We started with an open-source AES implementation written in Java [10]. We sought to prove that the `encrypt` and `decrypt` methods are pure. This would then enable us to check at runtime that these methods satisfy the inverse property, as described in Section 3.2.2.

Changes to the codebase

First, we refactored the code to pass the Joe-E verifier. The AES library initially contained mutable static state: it used static variables of array type to hold the S-box tables. We replaced these with `ImmutableArray`s, to meet Joe-E's requirement that all static variables be immutable.

Second, we refactored the class to provide verifiably pure methods. Originally, the AES library's interface had this type signature:

```
public AES()
public void setKey(byte[] key)
public byte[] encrypt(byte[] plain)
public byte[] decrypt(byte[] cipher)
```

After refactoring, the signatures for the relevant methods and constructors became:

```
public AES(ByteArray key)
public ByteArray encrypt(ByteArray plain)
public ByteArray decrypt(ByteArray cipher)
```

Method signatures for `encrypt` and `decrypt` were changed so that all parameters would have an immutable type, thus making the methods verifiably pure. This was accomplished by replacing each `byte[]` array with a `ByteArray`. Also, because `encrypt` and `decrypt` are instance methods on the `AES` class, we had to make the `AES` class immutable. As an immutable class, it can no longer have its key specified using a setter method that mutates its state. Instead, the key is specified as an argument to the constructor. Immutability of `AES` also required making all instance variables `final` and immutable. To accomplish this, we had to remove debugging trace information from the class. In a case where preserving such information is important, a suitable solution would be to return from the top-level methods an object containing both the original return value and the debugging trace for that method call.

Notice that we also changed the return type of the `encrypt` and `decrypt` methods to an immutable type. This was not strictly necessary for verifying the purity of the AES library, but it helps clients of the AES library write their own pure methods that manipulate data returned from the AES library. In general, returning an immutable data structure helps verify purity of other parts of the code.

After our refactoring, clients of the AES library are able to check that decryption is the inverse of encryption by inserting

```
\texttt{k.check(x);
```

before every call to `k.encrypt(x)`, where instance `k` is of type `AES`. The `check` method can be defined as follows:

```
public void check(ByteArray x) {
  assert(decrypt(encrypt(x)).equals(x));
}
```

Since this method is pure, inserting the call to `check` cannot change the program's behavior. Moreover, this call ensures that `encrypt` and `decrypt` satisfy the inverse property for every value `x` that is ever encrypted by any client of the AES library.

3.7.2 Voting machine

Motivation

Next, we examined the serialization and deserialization code of an experimental voting machine implementation [62]. We refactored the code to make serialization and deserialization pure. Our goal was to confirm at runtime that deserialization is the inverse of serialization, following the pattern described in Section 3.2.2. This ensures that all votes that are successfully recorded will be read back correctly during vote tallying.

Changes to the codebase

Nearly all changes made were simply replacing standard Java arrays with Joe-E immutable arrays. Another common change was adding the `Immutable` interface to classes that were already observationally immutable. (This required nothing more than adding "`implements Immutable`" to the class declaration.)

Another modification involved the use of a monotonically increasing serial number to filter out duplicate ballots. The last received serial number was stored as a static field inside the
`BallotMessage` class. Inside the deserialization method, the serial number of the ballot was compared with the static value of the most recently received serial number; if the serial number was already received, the deserialization method would return null. With detection of duplicate ballots written in this way, the deserialization is not deterministic. To fix this, we separated the deserialization functionality from duplicate ballot suppression.

The only other significant change necessary was to require that a `Ballot` received all of its `Races` upon construction. Prior to refactoring, the `Ballot` class exposed a method `addRace(Race r)`. This method had to be removed in order to make the `Ballot` class immutable.

The method that we wished to make verifiably pure serves to check the serialization. The method is called after the object has been serialized to an array of bytes, and tests that the serialized form deserializes to match the original ballot. Its signature was initially:

```
public static boolean deserializesTo (byte[] serialized, BallotMessage bm)
```

After refactoring, the method was changed to use a `ByteArray` instead of `byte[]`. The actual deserialization, which is performed by a constructor that takes a `ByteArray`, is also verifiably pure.

3.7.3 HTML parser

Our third application, an HTML parser [50], was a much larger and more instructive undertaking. Since our modifications to this library were significant, we ensured that it retained its functionality by verifying that our modified version and the original version produced the same results when run on a corpus of HTML test cases [26].

Motivation

Our primary goal was to refactor the code to make the top-level `parse` method pure. From a security perspective, a pure parse method is valuable for any system in which parses need to be performed on behalf of different users or using data from different sources. An example of this is on a web forum, where posts to the forum must be sanitized to prevent cross-site scripting attacks. A pure parse method together with a pure sanitization routine ensures that there can be no accidental data contamination between different posts, and that no private information about a user can be accidentally leaked into another post or to another user. Additionally, a pure parse method guarantees that a given parse is reproducible on any machine under virtually any circumstances.

Before refactoring, the top-level method signature, which resides in the `Parser` class, was the following:

```
public NodeList parse (NodeFilter filter)
    throws ParserException
```

Originally, neither the `Parser` class nor the `NodeFilter` class was immutable, and hence this method was not verifiably pure.

Mutable static state

We removed several instances of mutable static state from the HTML library, so that the code would pass the Joe-E verifier. For example, originally the only way to pass options to the parser was to set a global flag, parse, and then restore the flag, as follows:

```
boolean oldValue = SomeClass.SOMEFLAG;
SomeClass.SOMEFLAG = true;
try { parser.parse(); }
finally { SomeClass.SOMEFLAG = oldValue; }
```

```
String html = getHtmlStringFromSomewhere();
Parser p = new Parser(html);
NodeList list = p.parse(null); // null NodeFilter
// do something with the parse "tree" in list
```

Figure 3.7: A typical use of the `Parser` class. The HTML document is supplied to the constructor as a string. Then, the `parse` method is called with a `NodeFilter` as a parameter. A `NodeList` is returned, which contains a list of the top-level nodes from the HTML document.

This pattern seems to have been used to avoid propagating a configuration parameter through several levels in the call hierarchy. However, this use of global variables makes it harder to see how the flag is specified, renders the code thread-unsafe, and violates Joe-E's prohibition on shared mutable state.

We eliminated this pattern by augmenting the API with a top-level `parse` method that takes an extra argument and passes it as necessary to other parts of the program. The original top-level `parse` method remains, using a default value for the flag.

The original codebase also violated Joe-E restrictions by printing to `System.out` for debugging and reading the default locale using `java.util.Locale.getDefault()`. Using `Locale` objects in Joe-E code is not problematic, but the *default* locale is system-dependent and therefore non-deterministic. We instead modified the API to require that a `Locale` be passed as a parameter to objects needing access to the locale.

Instance method calls in constructors

We found many constructors that called other instance methods during their execution. As discussed earlier, Joe-E prohibits this (see § 3.6.2), so we had to eliminate all calls to instance methods from within constructors. This was bothersome—it was one of the few changes we had to make that did not reflect poor or nonstandard style in the original code—but fortunately we were able to work around the problem in every case by inlining the instance method, replacing the instance method with a static method, or using a factory method instead of a constructor. Nonetheless, this experience suggests that the restriction on calling instance methods from constructors may place an undue burden on Joe-E programmers. We are currently considering less restrictive alternatives for future Joe-E releases.

Immutable classes

Once the HTML parser's code passed the Joe-E verifier, we refactored the `Parser` and `NodeFilter` classes to be immutable.

The original `Parser` class contained a `Lexer` as an instance variable. In order to make the `Parser` class immutable, this instance variable had to be removed due to the fact that a `Lexer` is inherently mutable. We refactored the code to construct and use a `Lexer` inside the top-level `parse` method. A typical use of the `Parser` class can be seen in Figure 3.7.

Making the classes that implement the NodeFilter interface immutable was straightforward, except for the IsEqualFilter class. This required significant effort due to the fact that this class, which tests whether two nodes are equivalent to each other, contained an instance variable of type Node. As a result, all classes that implemented the Node interface had to be made immutable, which necessitated removing all setter methods from any Node subclass and requiring that all fields were set upon construction of any subclass of Node.

Refactoring the Node subclasses to be immutable proved difficult due to a nonstandard construction pattern. The library used a prototype construction pattern to support the creation of custom parsers that recognize varying sets of HTML tags[4]: before parsing, the caller could register a set of Node prototypes. When the Lexer needed to construct a new Node, it would clone a prototype and then overwrite the relevant fields of the clone using setter methods.

We refactored the code to use a more standard construction pattern in which Nodes are constructed using a constructor that takes an argument for each instance field that needs to be set. Minor functionality was lost with this change, as it is no longer possible to create a custom NodeFactory (without creating a custom class implementing the NodeFactory interface) to recognize a different set of nodes.

Also, to make Nodes immutable, we had to split the Page class into two classes. Before refactoring, the Page class conflated two distinct purposes. It was used by both the Lexer and by the Node classes. The Lexer used a Page instance during lexing to maintain information about the current position of the cursor in the page and to get and unget characters. This inherently requires a mutable class. On the other hand, the Node classes only used the Page object for finding out the line and column numbers for characters in the page. This information is fixed and will never change after construction of a Node. To reflect this fact, we created an immutable class called PageInfo to hold this information and extracted it from the mutable Page class. Now, when the Lexer creates a Node, it obtains the PageInfo from the Page and passes it to the Node's constructor.

As illustrated above, immutability is a property that necessarily spreads through related classes. We noticed similarities between these immutable data structures and those used in functional programming. For instance, immutable data structures must be constructed from the bottom up and hence are necessarily acyclic.

We also refactored the parse method to make the parse tree it returns be immutable. This is not necessary for the purity of the parse method, but it aids the creation of other pure methods that use the data structure returned by the parse method, since callers of the parse method can directly pass the parse tree it returns as a parameter to other pure methods.

[4] For example, one could create a parser that recognizes only img tags, and treats all other tags as generic tags with no hierarchical structure.

	Pure	**Total**	**% pure**
Methods	89	524	17%
Constructors	37	128	29%

Table 3.2: The number of pure and impure methods and constructors in the Waterken Server.

3.7.4 Summary of patterns

Using a strictly-functional style throughout a program is the most reliable pattern for attaining verifiable purity, as it ensures that every method will be pure. Such a strict approach is generally not necessary to achieve useful purity guarantees. None of the three applications that we refactored were written in an exclusively functional style, either before or after our modifications. Our approach to purity requires only that immutable types (and thus functional programming style) be used for the interface of a pure function, allowing its internal algorithms to be written in an imperative fashion if the programmer so desires.

Objects that have cycles (for example, a tree with parent pointers or a doubly-linked list) pose a challenge for our approach. A cyclic object graph, even if it is observationally immutable once fully constructed, cannot be statically verified as immutable in our system. We may therefore be unable to verify purity for methods that use such objects.

Joe-E required us to eliminate the use of mutable static state and pass parameters explicitly as arguments instead of using mutable global variables. We found that this brought our code closer to a functional style and had benefits of its own.

We believe that new code can take better advantage of Joe-E's guarantees if the class hierarchy is designed with immutability in mind. If part of a class is immutable but the rest of the class is not, the entire class must be treated as mutable. Consequently, if a concept has separable mutable and immutable aspects, it may be helpful to represent it as two separate classes.

3.7.5 Waterken Server

The Waterken server is an extensible web server designed for building distributed web services [14]. Waterken is implemented in a mixture of Joe-E and Java. The Joe-E code was not retrofitted from Java to Joe-E, as in our previous examples, but was designed and implemented following object-capability principles. The Joe-E portion is substantial, comprising 8,246 source lines of code and 132 classes.

We counted the number of pure methods in the Waterken Joe-E code. (See Table 3.2.) Our results are somewhat surprising: a large fraction of methods (17%) and an even larger fraction of constructors (29%) are verifiably pure. While the code was written in an object-capability style, verifiable purity was not an explicit goal. This suggests that verifiable purity can (and does) occur as a natural consequence of object-capability discipline.

3.8 Discussion

One advantage of our approach is that it can facilitate reasoning about side-effects and data dependencies for methods even if they do not strictly meet our requirements to be functionally pure. Since the accessible data and possible effects of a method are limited to objects reachable from its arguments, these effects are still bounded even if some arguments are mutable. In particular, the method can only mutate objects that are reachable from its non-immutable arguments. Typing and capability reasoning can limit this set to a small portion of the in-memory objects in the program, e.g., the values in a single array of ints, or the private instance fields of an object.

One can sometimes use these bounded effects to achieve purity properties from methods that are not individually pure. For instance, consider the following set of operations on a non-immutable object o:

```
T o = new T(a);
o.f(b); o.g(c);
... // do something with o
```

If the constructor is pure and the arguments a, b, and c are all immutable, then the state of o after this sequence of operations will be a deterministic function of a, b, and c and no other side effects will occur. We will refer to a sequence of invocations with this property as a *functionally pure sequence.*

If all of the inputs are known in advance, the sequence above can be written as a single verifiably pure function; for this example, we would have:

```
T pure(A a, B b, C c) {
   T o = new T(a);
   o.f(b); o.g(c);
   return o;
}
```

The more interesting case is where the inputs are not known in advance, such as if some of them come from interactions with a user. In this case, some of the inputs depend on information received from the program, e.g., return values from invocations on o. This case can be expressed as a sequence of verifiably pure method calls if it is refactored to use a purely functional style. Specifically, we would refactor T to be immutable and replace each mutating instance method of T with one that returns both the original return value and a new object that has the modifications applied.

For cases in which making T immutable is impractical or cumbersome, we need a new set of rules sufficient to verify such a sequence is functionally pure. It is safe to add return values to the first scenario above, as long as they do not enable modifications to the object's internal state. This limitation can easily be verified by requiring the return values to be immutable. (Thrown exceptions would also be a concern, but Joe-E already requires all throwables to be immutable). This set of restrictions is not as trivial to check as the ones needed for individual methods to be verifiably pure, but it allows for reasoning about useful properties of non-immutable objects.

The pattern allows for purity to be demonstrated in event-based interactive systems, such as a voting machine. Each voter's actions constitute a stream of events that should be interpreted as they arrive to produce the voted ballot. Pure sequences can allow us to verify that the voted ballot and behavior of the voting machine are a deterministic function of the sequence of input events.

Functionally pure sequences also occur in Waterken's implementation of deterministic server processes that react to input events. Each event causes mutations in the internal state of a handler object dedicated to that event's connection. Because these mutations are local to the per-connection object, the behavior of each server process is a deterministic function of the sequence of input events it receives, even though each individual call to the event-processing method is not verifiably pure.

3.9 Conclusions

Verifiable purity is useful for verifying many kinds of high-level security properties. A language with appropriate characteristics can greatly simplify the task of writing verifiably pure code. By combining determinism with object-capabilities, we describe a new class of languages that allow purity to be achieved in largely imperative programs. As such a language, Joe-E allows programmers to flexibly leverage verifiable purity while still using imperative algorithms in a familiar language.

Chapter 4

Joe-E's Overlay Type System and Marker Interfaces

This chapter includes material from a short paper [40] coauthored with David Wagner, presented at ACM SIGPLAN's 5th Workshop on Programming Languages and Analysis for Security in June 2010.

4.1 Introduction

In designing code for security review, it can be helpful to be able to characterize the properties of objects in a program. A sufficiently rich type system can help with this goal: programmers can document the properties of objects through their declared type, and an appropriately constructed static code verifier can check these properties for all instances of a particular type. In this way, implementing one of these types serves as an annotation to indicate that certain properties of the class hold. Once the desired property has been statically verified for all instances of a particular type, a code reviewer can use this property in reasoning about the structure or behavior of the associated type.

Static verification cannot be separated from types: the static verification process must take into account the structure of the type system. If the reviewer sees a variable in the program and wants to reason about the properties of objects stored in that variable, just checking what verified annotations exist on the declared type of the variable is insufficient. In languages supporting subtyping, the concrete type of the object stored in the variable may not match the declared type of the variable; it may be of a subtype instead. One approach to addressing this risk would be to locate all subclasses of the annotated class and check them when the annotated class is verified; but this approach fails if the source code of some subtypes is not available, perhaps because they have not yet been written.

The alternative is to run the verifier on all code, and to verify properties of classes when any of their supertypes has an annotation declaring that property. In other words, the annotation used to declare and verify properties of a class must be inherited by subclasses.

In Java, this can be accomplished by using interface types as annotations. We define a *marker interface* for each semantic property we want to verify. Then, a class may implement a marker interface to declare that it satisfies the property associated with that interface, and the verifier checks that the associated properties hold for every class that implements the marker interface.[1]

In Joe-E, we define a standard set of marker interfaces that serve to document that a class has specific properties, such as immutability. Our verifier uses these interfaces to verify these properties: each marker interface is associated with a static analysis to check that its properties hold for the classes that implement it.

A challenge with this technique is that pre-existing classes, such as those in the Java standard library, do not implement any marker interfaces. These classes may satisfy the properties associated with a marker interface, but cannot be declared to implement it without modifying the library. If we were to modify the library in this way, Joe-E programs would no longer be compatible with standard Java distributions that include unmodified libraries. Instead, we chose to leave the library unmodified and maintain a separate, auxiliary set of *honorary* marker-interface subtyping relationships. For instance, we want to treat `java.lang.String` as though it implemented the immutable marker interface, even though it is not declared that way. When these additional subtyping relationships are added to the base type system as defined by Java, the result is a combined *overlay type system* used by our verifier. This extended type system is used when checking the restrictions our verifier places on code to ensure that it falls within our language subset. Joe-E programs are also able to test relationships in the overlay type system through library methods we add for this purpose.

Thus, Joe-E programs are simultaneously well-typed under two different (but related) type systems. Every Joe-E program is a valid Java program, and thus is well-typed under the base Java type system. Moreover, our static verifier ensures that every Joe-E program is also well-typed under Joe-E's overlay type system. Joe-E programs can dynamically query typing relationships in both type systems. Of course, to ensure that this is sound, there must be a strong relationship between the two type systems: the overlay type system must be a refinement of the base type system. We discuss how this is ensured in Section 4.2.

In Joe-E, we use marker interfaces to verify a number of interesting class-based properties, generally relating to immutability and object identity. We have identified simple source code analyses that can be used to verify these properties statically. Of particular interest are the techniques used to ensure immutability, as Java has a number of corner cases that must be considered to verify this property reliably.

In the remainder of this chapter, we formally introduce the notion of an overlay type system and identify conditions that suffice to ensure that the resulting type system will be complete and consistent. We then describe how we verify class immutability and the additional restrictions we must impose on Joe-E code to ensure that our immutability guarantees

[1] Using actual interfaces rather than Java's inherited annotations provides a number of benefits, as described in Section 4.2.1.

are reliably enforced. Finally, we describe several other marker interfaces provided by Joe-E, including their associated class properties and how they are useful for writing reviewably secure code.

The rest of this chapter is organized as follows:

- We introduce and formalize the concept of an *overlay type system*. This concept is useful for a new programming language that is defined as an extension or subset of some existing language that already has its own type system; it enables us to extend or refine the base type system provided by the pre-existing language.

- For Java, we identify conditions under which extensions to the base Java type system yield an overlay type system that is sound and consistent.

- We present the design of a mechanism for statically verifying immutability in the class granularity in Java. Compared to prior work, our approach is relatively simple and predictable for programmers, and yet is sufficiently expressive for construction of new code written with our mechanism in mind.

- We describe several other semantic properties of object-oriented classes relating to object identity. We specify how to verify that Joe-E code satisfies these properties.

4.2 Overlay Type System

We introduce the notion of an overlay type system, consisting of additional subtyping relationships layered atop a base type system.

We formalize a type system I as a partial function that generates a subtyping relation R from a set of typing facts Γ. The resulting subtyping relation R is a transitive binary "is-a" relation on types, i.e. a set of subtyping judgements of the form $Type \times Type$. The typing facts Γ represent typing-relevant metadata for a particular language, e.g. the declared superclass for user-defined classes. These can take different forms for different language's type systems (or possibly even different representations of the inputs for the same type system). For some inputs, a type system can fail to generate a relationship between types. This occurs when the set of facts fails some correctness criterion, e.g. cycle-freeness.

An overlay type system for a base type system I_B is a second type system I_O that, when invoked on the input of I_B with additional facts, gives an augmented version of its subtyping relation. We use the following terminology: Γ_B is a set of base typing facts; denote $I_B(\Gamma_B)$ as T_B. Let $\Gamma_O = \Gamma_B + H$; then let R_O be $I_O(\Gamma_O)$.

An overlay type system is then defined as a type system with the following two properties, for every Γ_B and H such that $I_O(\Gamma_B \cup H)$ is defined:

1. Every typing judgement valid in the base type system also holds in the overlay type system, i.e. $R_B \subseteq R_O$

2. The typing relationships extant in the overlay type system correspond to a possible set of relationships derivable in the base system, i.e. there exists a transformation T such that $I_B(T(\Gamma_O)) = I_O(\Gamma_O)$.

4.2.1 Marker Interfaces

As mentioned earlier, Joe-E uses marker interfaces to document the security properties of code. Because interfaces are part of the Java type system, one can declare fields, methods, and return values to have the marker interface's type. This allows one to declare that a field or argument can hold arbitrary objects that implement the marker interface, or that a method only returns objects that implement the interface.

Marker interfaces used to indicate class-based properties need to satisfy two properties for our analysis to be sound, due to the fact that a Java object belongs to its runtime type as well as any supertypes, which may implement fewer marker interfaces.

- Implementing a marker interface must only add, and may not remove, restrictions on the *structure and behavior* of a class. This ensures that all classes that implement a marker interface can be relied upon to have the associated guarantees, even if a subclass implements additional marker interfaces.

- Implementing a marker interface must only add, and may not remove, ways that such a class may be validly *used*. This ensures that one cannot circumvent restrictions on how an object can be used by upcasting it to a supertype.

For Joe-E, the *base type system* is defined by the Java language; it defines certain subtyping relationships. Because standard library classes are not defined to implement our interfaces, and we did not want to replace the standard library, Joe-E provides a way to declare a Java library class to *honorarily implement* a marker interface. These additional implementation relationships, added to the base subtype relations in the Java type system, define an augmented *overlay* subtype relation used by the Joe-E verifier.

All type checking performed as part of the standard Java compilation process and JVM runtime enforcement uses the base subtyping relation, as required to preserve Java semantics. However, Joe-E's additional restrictions are defined in terms of the overlay subtype relation. The verifier thus includes the additional, honorary implementation relationships when checking Joe-E language restrictions, including properties of code that implements marker interfaces. Classes that honorarily implement the marker interfaces are generally not Joe-E code, and so are not subject to the same checks. Instead, we manually review these classes and ensure that their exposed functionality is consistent with the marker interfaces we have them honorarily implement. We refer to this process as "deeming" the classes to satisfy the interface's restrictions.

The overlay subtype relation is made visible to user code at runtime by means of library methods that query its runtime representation. We provide analogues to the `instanceof` keyword and the `Class.isAssignableFrom()` method that reflect the same subtyping relation used by the verifier.

4.2.2 Properties

We wish to ensure that the overlay type system is well-defined (e.g., free of cycles) and consistent in structure with the Java base type system (i.e. has the same subtyping relationships as the base type system would if the honorary relationships were added to their respective classes in the base type system).

We use the following notation:

$$\begin{aligned}
\mathit{csuper}(\tau,\tau') &\equiv \tau \text{ is a class explicitly declared to have superclass } \tau' \\
\mathit{cimpl}(\tau,\tau') &\equiv \tau \text{ is a class explicitly declared to implement the interface } \tau' \\
\mathit{isuper}(\tau,\tau') &\equiv \tau \text{ is an interface that directly extends the interface } \tau' \\
\Gamma &\equiv \text{a set of facts of the form } \mathit{csuper}(\tau,\tau'),\ \mathit{cimpl}(\tau,\tau'), \\
&\quad\ \ \mathit{isuper}(\tau,\tau') \\
\Gamma \vdash \tau \sqsubset \tau' &\equiv \text{From } \Gamma\text{, it is possible to derive that } \tau \text{ is a strict subtype of} \\
&\quad\ \ \tau'\ (\tau \neq \tau')
\end{aligned}$$

In our overlay type system, we augment the Java typing rules with additional interface implementation relationships, *hcimpl* and *hisuper*, with the following meanings:

$$\begin{aligned}
\mathit{hcimpl}(\tau,\tau') &\equiv \tau \text{ is a class that honorarily implements the interface } \tau' \\
\mathit{hisuper}(\tau,\tau') &\equiv \tau \text{ is an interface that honorarily extends the interface } \tau'
\end{aligned}$$

The overlay type system adds only new subtyping relationships; it does not add any new types. We indicate that τ is a strict subtype of τ' in the overlay type system (for $\tau \neq \tau'$) by $\tau < \tau'$.

4.2.3 Formalizations

The simplest formalization for a Java overlay type system with added honorary implementation relationships is given in Figure 4.1. In this formulation, it is easy to see that the two correctness criteria for an overlay type system will be satisfied. Given T that leaves the base facts unchanged and simply maps the honorary facts $\mathit{hcimpl}(\tau,\tau')$ and $\mathit{hisuper}(\tau,\tau')$ to $\mathit{cimpl}(\tau,\tau')$ and $\mathit{isuper}(\tau,\tau')$, all previously existing relationships will be derivable identially to in the base type system. For the second correctness criterion, there is a direct isomorphism between the derivation in the overlay type system and the derivation with the honorary facts added via T, provided that the added overlay facts would not result in rejection by the base type system.

In our actual implemenation, we do not perform the full inference implied by the rules given for the overlay type system in this figure. Instead, we use a simplified set of inference rules to determine whether a type implements one of our marker interfaces. We then restrict the kinds of honorary implementation relationships we add in order to ensure that these weaker inference rules still result in a valid overlay type system.

Rules	Java	Overlay
Axioms	$\dfrac{\mathit{csuper}(\tau, \tau') \in \Gamma}{\Gamma \vdash \tau \sqsubset \tau'}$ $\dfrac{\mathit{cimpl}(\tau, \tau') \in \Gamma}{\Gamma \vdash \tau \sqsubset \tau'}$ $\dfrac{\mathit{isuper}(\tau, \tau') \in \Gamma}{\Gamma \vdash \tau \sqsubset \tau'}$	$\dfrac{\mathit{csuper}(\tau, \tau') \in \Gamma}{\Gamma \vdash \tau < \tau'}$ $\dfrac{\mathit{cimpl}(\tau, \tau') \in \Gamma}{\Gamma \vdash \tau < \tau'}$ $\dfrac{\mathit{isuper}(\tau, \tau') \in \Gamma}{\Gamma \vdash \tau < \tau'}$ $\dfrac{\mathit{hcimpl}(\tau, \tau') \in \Gamma}{\Gamma \vdash \tau < \tau'}$ $\dfrac{\mathit{hisuper}(\tau, \tau') \in \Gamma}{\Gamma \vdash \tau < \tau'}$
Transitivity	$\dfrac{\Gamma \vdash \tau \sqsubset \tau' \quad \Gamma \vdash \tau' \sqsubset \tau''}{\Gamma \vdash \tau \sqsubset \tau''}$	$\dfrac{\Gamma \vdash \tau < \tau' \quad \Gamma \vdash \tau' < \tau''}{\Gamma \vdash \tau < \tau''}$
Object	$\dfrac{\tau \neq \texttt{Object}}{\Gamma \vdash \tau \sqsubset \texttt{Object}}$	$\dfrac{\tau \neq \texttt{Object}}{\Gamma \vdash \tau < \texttt{Object}}$
Array	$\dfrac{\Gamma \vdash \tau \sqsubset \tau'}{\Gamma \vdash \tau[\,] \sqsubset \tau'[\,]}$	$\dfrac{\Gamma \vdash \tau < \tau'}{\Gamma \vdash \tau[\,] < \tau'[\,]}$

Figure 4.1: Simple rules for the Java type system's (strict) subtype relation $\sqsubset$ and a small change that adds additional interface relationships to generate an overlay type system subtype relation $<$.

Alternate typing rules for Java. We provide an equivalent formulation of Java's typing rules below, to facilitate a correctness proof of our modified overlay type system inference rules. For a set of facts Γ, the subtyping relation $\tau \sqsubset \tau'$ is the least relation satisfying the following inference rules (where τ, τ', τ'' range over all valid reference types[2]):

$$\frac{\mathit{csuper}(\tau, \tau') \in \Gamma}{\Gamma \vdash \tau \sqsubset_1 \tau'}\mathit{Axiom}$$

$$\frac{\mathit{cimpl}(\tau, \tau') \in \Gamma}{\Gamma \vdash \tau \sqsubset_1 \tau'}\mathit{Axiom}$$

$$\frac{\mathit{isuper}(\tau, \tau') \in \Gamma}{\Gamma \vdash \tau \sqsubset_1 \tau'}\mathit{Axiom}$$

$$\frac{\Gamma \vdash \tau \sqsubset_1 \tau'}{\Gamma \vdash \tau \sqsubset \tau'}\mathit{Promote}$$

$$\frac{\Gamma \vdash \tau \sqsubset_1 \tau' \quad \Gamma \vdash \tau' \sqsubset \tau''}{\Gamma \vdash \tau \sqsubset \tau''}\mathit{Trans}$$

$$\frac{\tau \neq \texttt{Object}}{\Gamma \vdash \tau \sqsubset \texttt{Object}}\mathit{Object}$$

$$\frac{\Gamma \vdash \tau \sqsubset \tau'}{\Gamma \vdash \tau[\,] \sqsubset \tau'[\,]}\mathit{Array}$$

Our overlay type system. In our implementation, the relation $<$ is defined as the least relation satisfying the following inference rules:

$$\frac{\mathit{hcimpl}(\tau, \tau') \in \Gamma}{\Gamma \vdash \tau < \tau'}\mathit{Axiom}$$

$$\frac{\mathit{hisuper}(\tau, \tau') \in \Gamma}{\Gamma \vdash \tau < \tau'}\mathit{Axiom}$$

$$\frac{\Gamma \vdash \tau \sqsubset \tau'}{\Gamma \vdash \tau < \tau'}\mathit{Incl}$$

$$\frac{\Gamma \vdash \mathit{hcimpl}(\tau, \tau') \quad \tau' \sqsubset \tau''}{\Gamma \vdash \tau < \tau''}\mathit{Trans}$$

$$\frac{\Gamma \vdash \mathit{hisuper}(\tau, \tau') \quad \tau' \sqsubset \tau''}{\Gamma \vdash \tau < \tau''}\mathit{Trans}$$

$$\frac{\Gamma \vdash \tau < \tau'}{\Gamma \vdash \tau[\,] < \tau''[\,]}\mathit{Array}$$

[2] This model excludes primitive types, such as `int` and `boolean`, as they are unrelated to reference types.

Constraints on honorary relationships. We impose the following constraints on the honorary typing relationships *hcimpl*, *hisuper*:

1. $(\tau \sqsubset_1 \tau') \wedge \mathit{hcimpl}(\tau', \tau'') \Rightarrow (\tau \sqsubset \tau'') \vee \mathit{hcimpl}(\tau, \tau'')$
2. $(\tau \sqsubset_1 \tau') \wedge \mathit{hisuper}(\tau', \tau'') \Rightarrow (\tau \sqsubset \tau'') \vee \mathit{hisuper}(\tau, \tau'') \vee \mathit{hcimpl}(\tau, \tau'')$
3. $(\mathit{hcimpl}(\tau, \tau') \vee \mathit{hisuper}(\tau, \tau')) \wedge \tau' < \tau'' \Rightarrow \tau' \sqsubset \tau''$
4. $\mathit{hcimpl}(\tau, \tau') \Rightarrow \tau \neq \texttt{Object}$

The first two constraints are motivated by a desire to simplify the use of the overlay type system. They are the result of two different requirements, one for newly written Joe-E classes, and one for the types of honorary relationships we add to pre-existing classes.

It is beneficial for as many classes as possible to implement a marker interface in the base type system, as opposed to exclusively in the overlay type system. If the subtype relationship is known to the Java compiler, it is possible to assign the implementing type to a variable whose declared type is the interface. For this reason, we require new classes to explicitly implement any interfaces that are honorarily implemented by their superclasses and superinterfaces. For Joe-E code, this rule is enforced by the Joe-E verifier.

In order to facilitate documentation, we require all existing classes to explicitly honorarily implement any marker interfaces that they would otherwise implicitly inherit from their supertypes. Our tool that assembles the database of honorary typing relationships verifies that this is the case.

Restrictions 1 and 2 also have the benefit of allowing us to simplify the logic we use to determine subtyping relationships in the overlay type system. The typing rules for the overlay type system above reflect this simplification in that there are only overlay typing inference rules for scalar types τ that are the subtype in honorary typing facts. (If we did not have these simplifications, we could more directly translate the rules of the base type system into the overlay type system, e.g. by simply replacing all $\sqsubset$ with $<$ and adding the additional *Axiom* rules. The complexity of the proof for the completeness theorem largely derives from proving that the simplified rules remain complete due to the invariants above.)

The third rule indicates that none of the marker interfaces themselves have any honorary supertypes. As all marker interfaces are new code, there is no need to make use of the honorary mechanism when they can extend the supertype in the base type system.

The fourth rule states that we are not so foolish as to add honorary implementation relationships to the class `Object`, as this would inevitably result in circularity in the type system.

Results. With these definitions, we will prove that properties 1 and 2 still hold, i.e. that the simplified formulation remains a valid overlay type system.

The proof of Property 1 is trivial: Any subtype relation present in the original Java type system holds in the overlay type system by applying of the *Incl* rule.

Demonstrating Property 2 is more difficult. To accomplish this we must specify a transformation T resulting in a set of typing relationships equivalent to those derived for our rules for I_O. We establish this property by defining a function T and verifying the following three facts:

1. Assuming that Γ_O satisfies the restrictions 1-4 and that $T(\Gamma_O)$ is a valid input to I_B, then $I_O(\Gamma_O) \subseteq I_B(T(\Gamma_O))$.
2. Assuming that Γ_O satisfies the restrictions 1-4 and that $T(\Gamma_O)$ is a valid input to I_B, then $I_B(T(\Gamma_O)) \subseteq I_O(\Gamma_O)$.
3. If Γ_O satisfies the restrictions 1-4, then $T(\Gamma_O)$ will be a valid input to I_B.

Let T denote the translation from honorary implementation facts into their equivalents in the base type system by replacing all instances of *hcimpl* with *cimpl* and *hisuper* with *isuper*. Given a set of base typing facts Γ_B and additional honorary typing facts H, we wish to prove that every relationship derivable for $<$ using the facts in $\Gamma_B \cup H$ is also derivable for $\sqsubset$ using the facts in $T(\Gamma_B \cup H)$ and vice versa.

Soundness Theorem. Given a set of base typing facts Γ_B and additional honorary typing facts H that satisfy Restrictions 1–4 and such that $T(\Gamma_B \cup H)$ is a valid input to I_B, $\Gamma_B \cup H \vdash \tau < \tau' \Rightarrow T(\Gamma_B \cup H) \vdash \tau \sqsubset \tau'$.

Completeness Theorem. Given a set of base typing facts Γ_B and additional honorary typing facts H that satisfy Restrictions 1–4 and such that $T(\Gamma_B \cup H)$ is a valid input to I_B, $T(\Gamma_B \cup H) \vdash \tau \sqsubset \tau' \Rightarrow \Gamma_B \cup H \vdash \tau < \tau'$.

We are aware of three reasons why a set of Java classes could fail to compile due to the presence of additional superinterfaces. These are:

1. An added interface declares a method that is not implemented by a class that newly implements it
2. A class or interface comes to have the same parameterized superinterface in two different ways with different parameterizations (JLS § 8.1.5)
3. A circularity in the resulting subtype relation (JLS § 8.1.4)

The first problem above is avoided by only adding as honoraries interfaces that declare no methods, or only declare methods that are present on `Object` and thus implemented by all classes. The second is avoided by not honorarily adding implementations of parameterized interface types.

The way that the overlay type sytem is constructed ensures that it has no cycles, i.e. $\nexists \tau . \tau < \tau$, assuming that the base type system is free of such cycles.

Non-circularity Theorem. Given a set of base subtyping facts A and a set of honorary subtyping facts H that satisfy Restrictions 1–4, $\nexists \tau . A \vdash \tau \sqsubset \tau \Rightarrow \nexists \tau' . A \cup H \vdash \tau' < \tau'$.

Proofs of these theorems may be found in Section 4.7.

4.3 Immutability

Joe-E provides support for verification of class immutability: all instances an immutable class are guaranteed to be immutable. By immutable, we mean that no state reachable from an immutable object can be observed to change. Any two reads of a field transitively reachable from such an object will return the same value. Our immutability requirement is stricter than those previously studied in that we do not exempt a partially-constructed object that escapes its constructor from the need to satisfy observational immutability. A client of such an object may be unaware that it is only partially constructed, and thus will observe a change of its fields if they are later initialized to a non-null, non-zero value.

Immutability is helpful for reasoning about the correctness and robustness of code, for a number of well-known reasons. One particular reason motivates our strong immutability requirement: if an object is immutable, code that makes use of it does not need to defend against modifications to that object by other code in the system. For instance, this eliminates the possibility of time-of-check-to-time-of-use attacks on those values. Also, immutability facilitates verification of functional purity (determinism and side-effect freeness) of methods, as described in Chapter 3.

In contrast with other type systems for immutability such as Immutability Generic Java [79], in which references to read-only or immutable objects are subjected to additional type checks, in Joe-E we only support class immutability, i.e., classes with no mutable state. This is less expressive, but we have found it to be sufficient for new code. Class types that implement the `Immutable` marker interface are verified to be immutable, and those that do not are not.

Joe-E's static verifier checks that each class that implements the `Immutable` interface is indeed immutable by verifying that the class C meets all of the following requirements:

1. Every instance field of C must be both declared `final` and of a primitive or immutable type. No such field may be declared `transient`. ("Every instance field of C" includes fields of all superclasses, whether accessible to C or not; this includes, for instance, all private fields of all superclasses.)

2. If C or any of its superclasses is a non-static inner class, every enclosing class must be immutable.

3. If C is a local class (an inner class defined within a method), all local variables defined outside C that are observable by C must be immutable.

Any violation of these requirements is a verification-time error.

The third requirement is necessary because local classes in Java can make use of `final` local variables in the scope in which they are defined. The Java compiler analyzes the code of each local class L to see which local variables are used by L. When looking for uses of local variables, it also scans any related inner classes that L constructs, or that transitively are constructible via a chain of such class constructions. Any local variable used by L, or by any related class it could construct, is implicitly included as a field of L.

Our verifier duplicates the calculation made by the compiler to determine which local variables will implicitly be included in each inner class, and uses them when verifying that such a class is immutable.

Note that immutability of a class C does not place any restrictions on any local variables of its methods, or imply that classes defined within C must themselves be immutable. It also says nothing about the mutability of the arguments passed to C's methods. The checks performed on C serve only to ensure the immutability of all objects reachable from all of C's fields, including implicit fields inserted by the compiler.

4.3.1 Ensuring Final Means Final

In Java, the `final` keyword does not guarantee that a field's value will never change. If a reference to an object escapes before it has been fully initialized, code might observe one of its fields once at its default value, before the field is initialized, and later observe the field at a different value, after initialization. This would allow an otherwise-immutable object to appear to change its value.

To address this problem and ensure that `Immutable` objects are truly immutable, Joe-E prevents Joe-E code from reading a field before it has been initialized. We ensure this by preventing the `this` pointer from escaping from any constructor[3], enforced as follows:

1. Instance initialization must not call any instance methods on the object being constructed (including supermethod invocations like `super.m()`).

2. Initialization of a class C must not call the constructor of any non-static inner class of C, i.e., any anonymous class or non-static member class that is defined within C or any of C's superclasses. (Non-static inner class instances have a reference to their containing object and thus its fields; this restriction ensures that no code from such an inner class executes during construction.)

3. Initialization must not reference the `this` pointer corresponding to the object being constructed, except as a way to name fields (e.g., a use or definition of the field `f` using the expression `this.f` is permitted). This restriction ensures that `this` cannot become aliased. For inner classes, references to enclosing objects' `this` pointers are unrestricted.

For legacy Java code, these restrictions would be too restrictive: a non-trivial amount of existing code might violate these rules. However, for new code written in Joe-E, we have found these restrictions to be tolerable.

Java has a similar weakness with static final fields: if there is a circular dependency in classes' static initializer logic, it is possible for code executing during static initialization of

[3] At present we do this for all classes, not just those that are immutable, as they may have an immutable subclass. This is the simplest approach, but is stricter than necessary, as some classes may inherently preclude an immutable subclass, e.g., by being final or declaring non-immutable fields.

```
public final class LockedBox<T> {
    private final Token key;
    private final T content;

    public LockedBox(Token key, T content) {
        this.key = key;
        this.content = content;
    }

    public T getContent(Token key) {
        if (key == this.key) {
            return content;
        } else {
            throw new IllegalArgumentException();
        }
    }
}
```

Figure 4.2: A locked box class. Once the content is stored in the box, it can only be retrieved again given the key object.

these classes to see uninitialized values for their fields. Joe-E does not currently address this problem, which does not technically violate our object immutability guarantees, as it only affects static fields. However, if an immutable object reads a value from a static field when an instance method is invoked, it may return different results on different invocations, which appears the same as a change in state, so we would still like to close this loophole.

4.4 Identity-based Authority

One basic pattern of object-capability based reasoning is to consider the evolution of the object graph as a program executes. This graph has a node for each active stack frame and in-memory object and directed edges connecting each reference-typed local variable and field to the object it points to. Given any snapshot of this graph, the set of objects reachable from each node bounds the authority available to the corresponding object. It is also possible to bound the possible future propagation of references between objects in the graph.

The coarsest bound on this propagation is bidirectional reachability on the object graph. A reference can potentially propagate from any node that has a reference to it to any adjacent node. This simple bound is too imprecise; all live objects will fall into the same component and thus we would be able to conclude nothing.

We can reason more precisely if we verify and rely on certain behaviors of shared objects in the reference graph. Consider the locked box class presented in Figure 4.2. (This is an adaptation of a construction [45, §6] that dates back to 1973.) The box's constructor accepts an object to store in the box and a key object. The key must be presented to extract the object from the box. The class `Token` is an empty class used here solely for pointer comparison.

If a locked box is passed to another entity, the recipient cannot retrieve the box's contents unless it also obtains access to the key used when the box was constructed. The content object can only be extracted from the box if some object has a reference to both the box and the key at the same time. The key acts as an authentication token to permit a holder of the box to retrieve its contents. In Java, every object is more than its contents; it also has an *identity* in that it can be distinguished from other objects of identical type and contents by the use of the `==` or `!=` operators. `LockedBox` uses this identity as an unforgeable credential, as Joe-E's memory-safety ensures that there is no way to create an alias of an existing object from scratch.

Despite having no state and no methods, a token object used as a key conveys authority to objects that have a reference to it. Without the token, an object with a reference to a locked box cannot open it; with the token, it can. This pattern of programmatically presenting a credential to an object to enable additional functionality is known as *rights amplification.*

There are a number of alternate patterns that can be used for rights amplification. Instead of using an object reference as a token, the box could store a password in a private field, and only divulge its contents when a lexically-matching password is given as an argument. It could issue instances of a privately-constructable inner class for use as credentials, using their unforgeable type to authenticate instead of identity comparison. Using tokens, however, has the advantage of being based on a simple, fundamental property of object-capability languages (and of memory-safe languages like Java), unforgeability of object references.

4.4.1 Power and Tokens

Joe-E specifically supports reasoning about rights amplification using unforgeable token objects. In Joe-E, other ways of implementing rights amplification may still be effective, but are not provided the same language support. We provide a `Token` class in the Joe-E standard library for this purpose, and recommend that Joe-E applications use this `Token` class in places where their security relies upon unforgeable object identity. In this way, the `Token` class explicitly documents which objects are used for rights amplification on the basis of their identity. If this idiom is followed, the only objects that will convey authority solely by their object identity are instances of `Token` and its subtypes (collectively called tokens).

As another example, consider again the currency system of Figure 2.3. Each `Currency` object corresponds to a different currency, and is used to ensure that money is not accidentally transmuted from one currency to another. Additionally, it gives its holder the ability to mint virtual coins in that currency, even though the class itself has no fields or methods. A `Purse` object is used by a client, such as an object representing a player of an online game, to hold a number of units of a particular currency. If a client wants to transfer some money to another object, he first constructs a new, empty purse of the same currency using the unary constructor, then transfers some money into this new purse from his primary purse using the `takeFrom` method. The new purse can then be passed to the recipient, who can add the balance from it to her own main purse also using `takeFrom`. The `Purse` class can be reviewed to verify that currency cannot be created without the use of the `Currency` token; this reduces the portion of the program that would have to be reviewed to ensure that there are no bugs that might allow money to be created from nothing.

Clearly, it would be a problem if the `Currency` object is passed around indiscriminately; despite its lack of fields, it is important to audit everywhere in the program that it might be used. The fact that all tokens extend the same base class makes it easier to identify every place in the program where these objects might be used.

If developers write their code such that `Tokens` are the only objects that represent privilege by their object identity, then objects that contain no `Tokens` cannot convey privileges in this way. This makes it easier to reason about places that may perform rights amplification and supports the following pattern of security reasoning:

- Conservatively assume that any authority made available by the object identity of non-tokens is available everywhere in the program.
- Conservatively assume that all secret data can be guessed or leaked, and are available everywhere in the program.
- Check that the code never relies upon the unforgeability of object identity of non-tokens, or the unguessability of data, for authentication or security.
- Perform local checks of all uses of tokens to ensure rights amplification is implemented properly.

Without this refinement of basic object-capability reasoning, we might conclude that a `Currency` object yields no authority and that a `Purse` object might potentially provide the authority to mint new money; the first conclusion is wrong, and the second is too conservative.

To encourage using tokens solely for their object identity, and not as containers for other capabilities, the class `Token` is declared to implement `Immutable`. Thus, tokens (including subtypes of `Token`) cannot contain mutable objects. `Token` also implements `Equatable` so tokens can be compared for identity (see Section 4.5).

4.4.2 Powerless

Joe-E introduces the notion of a *powerless* type. Objects belonging to such types are immutable and do not contain any tokens, i.e., no tokens are transitively reachable by following a powerless object's field pointers.

A powerless object conveys no inherent or identity-based authority and thus can be excluded from the object reference graph entirely without loss of soundness. Due to its immutability, it cannot serve as a channel for propagating references, and because it is both immutable and token-free, it cannot contain any capabilities of concern for the reachability analysis.

Any authority granted to the holder of a powerless object is solely a product of the data it contains; this authority could be "forged" by anyone with knowledge of this data and thus does not reflect a type of capability that can be guarded by our system. (Note that cryptographic keys fall into this category; our system is not able to reason about cryptography, because Joe-E does not provide any provision for reasoning about knowledge or the flow of information—it supports reasoning only about the flow of references.) Any authority vested in the object identity of a non-`Token` object is not modeled in our view of authority and is conservatively assumed to be available to everyone.

An immutable object conveys no authority except for the unforgeable identity of any tokens it may contain. This potential form of authority distinguishes a powerless object from one that is merely immutable. (By definition, all powerless objects are also immutable.) In practice, most immutable objects are likely to also be powerless.

If a class C implements `Powerless` in the overlay type system, the Joe-E static verifier checks that C satisfies all of the following restrictions:

1. Every instance field of C must be both declared `final` and of a primitive or powerless type. No such field may be declared `transient`.
2. If C or any of its superclasses is a non-static inner class, every enclosing class must be powerless.
3. If C is a local class, all local variables defined outside C that are observable by C must be powerless.
4. C must not be a subclass of `Token`.

Any violation of these requirements is a verification-time error.

In Joe-E, in order to achieve the principle of least authority, we restrict global (static) fields to only contain `Powerless` objects. In addition to ensuring that the global scope does not contain any mutable state, this ensures authority-bearing tokens are not made globally available. A bigger concern is accidental escalation or malicious transmission of privileges due to an exception being thrown and caught. As exceptions are often hard to predict and reason about when performing a code review, in Joe-E we require all subtypes of `java.lang.Throwable` to be powerless in order to ensure that they cannot be used to transmit authority in unexpected ways 2.4.3. This allows us to limit our examination to the more explicit mechanisms for reference propagation when reasoning about how objects can communicate with the rest of a program.

```
public class Buggy {
    public static boolean isYes(String answer) {
        return answer == "yes" || answer == "Yes"
            || answer == "y" || answer == "Y";
    }
}
```

Figure 4.3: A method that violates the intuitive expectation that `String` is a value type. This code is probably a bug (and would not be allowed in Joe-E).

4.5 Selfless and Equatable

Java contains a number of library classes that, intuitively, are intended to act as value types: types where equality is determined by the the object's contents and whose object identity should be irrelevant. For instance, two different `String`s with the same contents compare equal, using the `equals()` method, and intuitively should be essentially interchangeable. However, Java exposes the object identity of `String`s: a client can distinguish two `String`s with the same contents using `==` or `!=`. Exposing the object identity of value types is usually undesirable.

Consider, for instance, Figure 4.3, where a buggy method compares two strings using `==` instead of `equals()`. This breaks the abstraction that strings should be value types. Java has no way to enforce that code treats `String` as a value type; in contrast, Joe-E hides the object identity of `String`s and similar library classes from Joe-E code, making these library classes true value types. Such abstraction-violating bugs can sometimes be tricky to detect. For example, in Figure 4.4, testing code that makes use of literal (and thus automatically-interned) strings will fail to replicate the incorrect behavior of the `isYes()` method when called with non-interned strings from user input that have the same contents.

Joe-E also enables programmers to define additional value types, with assurance that their object identity will be hidden from other Joe-E code. The programmer is responsible for writing correct `equals()` and `hashCode()` methods for each class that is intended to be a value type. Joe-E ensures that clients possessing references to instances of these classes cannot observe object identity, by prohibiting use of the `==` and `!=` operators on these classes. In addition, Joe-E helps the programmer of these classes avoid inadvertently revealing object identity (e.g., by calling `super.equals()` or `super.hashCode()` and leaking their return value to clients).

To achieve these goals, we introduce the notion of *equatable*, *selfless*, and *deep selfless* types. In Joe-E, the `==` and `!=` operators can only be used on equatable types. It is a verification error for any type to be simultaneously equatable and selfless; thus, the `==` and `!=` operators cannot be applied to selfless types. Furthermore, Joe-E ensures that selfless types do not reveal object identity in other ways. A type T is deep selfless if every object (transitively) reachable from an instance of T (by following fields) is selfless. These notions

```
public class Tester {
    public static void test() {
        if (!Buggy.isYes("yes"))
            fail();
    }
}
public class Client {
    void processInput(StreamTokenizer st) {
        // read a parameter using the stream tokenizer
        st.nextToken();
        if (Buggy.isYes(st.sval))
            doSomething();
    }
}
```

Figure 4.4: The bug in Figure 4.3 might not be detected by testing (due to automatic interning of string literals), but might trigger in practice.

mean that object identity is optional for each type in Joe-E, unlike in Java where all reference types have object identity.

A class may be marked as being *equatable* by implementing the `Equatable` marker interface. Joe-E code is allowed to compare two references using `==` and `!=` if at least one of the references' declared types is equatable, or if either reference is `null`. Specifically, we ensure that, for every use of the binary operators `==` and `!=`, the type resolved for at least one of the two operands must be either the null type, a primitive type, or a reference type that implements `Equatable` in the overlay type relation.[4] All other uses of `==` and `!=` are prohibited.

For instance, the buggy code in Figure 4.3 would not be allowed in Joe-E, because `String` is not an equatable type, and thus the use of `==` found there would result in a verification-time error.

A class C is *selfless* if and only if it implements the `Selfless` interface in the overlay type system. The Joe-E static verifier checks that each such class C obeys the following restrictions:

1. All instance fields of C must be `final` and may not be `transient`.

2. C must not be equatable.

[4] Two objects of distinct runtime types are never identical. Joe-E allows programs to determine and compare the concrete type of objects, and thus does not hide this fact. The result of `==` only reveals additional information if its operands are of the same concrete type; therefore it suffices to check that either operand is equatable.

3. The object identity of instances of the class must not be visible. This can be satisfied by one of:

    (a) C's superclass is a selfless type, or

    (b) C's superclass is `java.lang.Object`, C overrides `equals()` and `hashCode()`, and C doesn't call `super.equals()`. (No Joe-E class is allowed to call `Object`'s version of `hashCode()`, even on itself, due to its exposure of nondeterminism.)

Joe-E provides several guarantees about selfless objects:

- The identity of selfless objects is not exposed, even indirectly. Selfless types must override `Object`'s default implementation of `equals()`, which is equivalent to `==`. Neither a selfless type's version of `equals()`, nor any other method it defines, can use `==` or `!=` on itself or call `Object`'s identity-exposing versions of `equals()` or `hashCode()` on itself.
- The hash returned by a selfless object's `hashCode()` method will be a deterministic function of its contents. This follows because a selfless object must provide its own implementation of `hashCode()`, and the object's contents are the only objects observable by this code; the object's own identity is not visible, not even to its code. As a result, we can store arbitrary selfless objects in a hash table, without fear that their `hashCode()` method will expose nondeterminism.

Selfless classes are useful for constructing serialization code. To serialize a selfless object, we only need to serialize its contents (and possibly their identity); we do not need to record its own identity. This makes it easier to ensure that the result of serializing and deserializing an object is indistinguishable from the original. For instance, the Waterken server 3.7.5 uses Joe-E's `Selfless` interface to ensure the correctness of a performance optimization: when serializing a non-selfless type, we must maintain a unique serialized version per instance, whereas for selfless types, it is safe to make multiple copies of a single instance if that improves performance. f A *deep selfless* class C must satisfy the requirements for a selfless class. Also, all instance fields, all local variables of enclosing scopes observable by C, and all enclosing classes (if C or any superclass is a non-static inner class) must be deep selfless. Joe-E does not yet implement deep selfless, but it would be a straightforward addition. A deep selfless class is also powerless (but not necessarily vice versa).

One feature of Joe-E is that it makes it easy to verify when methods are definitely deterministic (see § 3.3.2). By deterministic, we mean that two successful invocations of the method with equivalent arguments will always yield equivalent results. The notion of "equivalence" depends upon the type of the object. If any arguments are not deep selfless, equivalence must take into account object identity: for instance, two immutable objects are equivalent if they have equivalent contents as well as the same object identity (or more generally, the same set of aliasing relationships to other objects in the method's scope, including other arguments and global variables). Put another way, if all we know about the method's arguments is that they are immutable, then we cannot rule out the possibility that the method's behavior and return value might depend upon the identity of its arguments. This notion of determinism is often weaker than we might prefer.

Selfless types enable us to strengthen the notion of determinism to exclude the possibility that the method might depend upon the identity of its arguments. When dealing with selfless arguments, we can refine the notion of equivalence: two selfless objects are equivalent if they have equivalent contents (regardless of their identity). When all arguments are deep selfless, then we can rule out the possibility that the method might depend upon their object identity.

For instance, if we have a method whose type signature is

```
public static boolean isYes(String s);
```

then (in Joe-E) we can conclude that this method's behavior and result will depend deterministically only upon the value of the string `s`, but not on `s`'s object identity.

Value types in Joe-E also make it possible to verify the correctness of memoization. Suppose we have a method whose arguments and return value are of value types: their types are deep selfless, and moreover are known (somehow) to have a correct implementation of `equals()`. Then this method can be transparently memoized. We can maintain a hashtable that maps argument lists to results; before invoking the method on some argument list, we look up the argument list in the hashtable. If an entry is found, we return the cached result without invoking the underlying method; otherwise, we invoke the method and add its result to the hashtable. Thanks to the property of deep selfless types, the memoized version will be indistinguishable from the original.

We have only limited experience with selfless and deep selfless types. We initially implemented `Selfless` types primarily to support Waterken's serialization logic. In retrospect, deep selfless is probably a more useful concept, but is not currently implemented in Joe-E.

4.6 Conclusions

We have demonstrated a way to extend an existing object-oriented language with new subtyping relationships for the purpose of analysis, without making modifications of libraries provided with the base language. This is useful for verifying a number of class-based properties, as it allows the use of existing library classes without modifying the libraries or reimplementing library functionality. We have identified a set of conditions under which this augmentation results in a cycle-free type system with all relationships that would exist if the library classes had been modified to add the new subtyping relationships.

4.7 Appendix: Proofs of Theorems

4.7.1 Completeness

Lemma. Given a set of base typing facts Γ_B and additional honorary typing facts H, if $\Gamma_B \cup T(H) \vdash \tau \sqsubset_1 \tau'$, either $\Gamma_B \vdash \tau \sqsubset_1 \tau'$, or else one of $\mathit{hcimpl}(\tau, \tau')$ or $\mathit{hsuper}(\tau, \tau')$ must be a fact in H.

Proof. A derivation for $\sqsubset_1$ given $\Gamma_B \cup T(H)$ can only be generated by a single application of one of the base *Axiom* rules for a fact in $\Gamma_B \cup T(H)$. If the fact used is in Γ_B, then $\Gamma_B \vdash \tau \sqsubset_1 \tau'$, as the same derivation holds without using any facts in $T(H)$. If the fact used is from $T(H)$, then the corresponding fact $\mathit{hcimpl}(\tau, \tau')$ or $\mathit{hsuper}(\tau, \tau')$ must be in H.

Completeness Theorem. Given a set of base typing facts Γ_B and additional honorary typing facts H that satisfy Restrictions 1–4 and such that $T(\Gamma_B \cup H)$ is a valid input to I_B, $T(\Gamma_B \cup H) \vdash \tau \sqsubset \tau' \Rightarrow \Gamma_B \cup H \vdash \tau < \tau'$.

Proof. by induction on the structure of the derivation of $T(\Gamma_B \cup H) \vdash \tau \sqsubset \tau'$. Four rules can give a derivation of this form:

1. **Promote.** If $\tau \sqsubset \tau'$ derives from a *Promote* rule, then $\Gamma_B \cup T(H) \tau \sqsubset_1 \tau'$ must hold. By Lemma 1, either $\Gamma_B \vdash \tau \sqsubset_1 \tau'$, $\mathit{hcimpl}(\tau, \tau') \in H$ or $\mathit{hsuper}(\tau, \tau') \in H$. In the first case, $\Gamma_B \cup H \vdash \tau \sqsubset_1 \tau'$ implies that $\Gamma_B \cup H \vdash \tau \sqsubset \tau'$ by rule *Promote*, and thus $\Gamma_B \cup H \vdash \tau < \tau'$ by rule *Trans*. In the other cases, application of the matching *Axiom* rule for $<$ immediately gives us $H \vdash \tau < \tau'$.

2. **Trans.** For a derivation of the *Trans* rule, we have τ, τ', τ'' such that $\Gamma_B \cup T(H) \vdash \tau \sqsubset_1 \tau'$ and $\Gamma_B \cup T(H) \vdash \tau' \sqsubset \tau''$. By our induction hypothesis, $\Gamma_B \cup H \vdash \tau' < \tau''$. By the lemma, either $\Gamma_B \vdash \tau \sqsubset_1 \tau'$, or else one of $\mathit{hcimpl}(\tau, \tau')$ or $\mathit{hsuper}(\tau, \tau')$ must be a fact in H.

   If $\Gamma_B \vdash \tau \sqsubset_1 \tau'$, consider the derivation of $\Gamma_B \cup H \vdash \tau' < \tau''$. There are four cases:

   (a) $\Gamma_B \cup H \vdash \tau' < \tau''$ is derived using the *Axiom* rule. Then by Property 1 or Property 2, we can derive that $\Gamma_B \cup H \vdash \tau < \tau''$ by the use of the *Axiom* rule or the *Incl* rule.

   (b) $\Gamma_B \cup H \vdash \tau' < \tau''$ is derived from the *Incl* rule. Then $\Gamma_B \vdash \tau' \sqsubset \tau''$, and by the base *Trans* rule, we get $\tau \sqsubset \tau''$; the *Incl* rule can be used to give $\Gamma_B \tau \sqsubset \tau''$.

   (c) $\Gamma_B \cup H \vdash \tau' < \tau''$ is derived from one of the *Trans* rules. Then there exists some type $\tau^* \sqsubset \tau''$ such that either $\mathit{hcimpl}(\tau', \tau^*)$ or $\mathit{hisuper}(\tau', \tau^*)$. By property 1 or 2 respectively, either (a) $\tau \sqsubset \tau^*$ in which case $\tau \sqsubset \tau''$ can be derived from the base type system's *Trans* rule, and then $\tau < \tau''$ follows from *Impl*, or (b) $\mathit{hcimpl}(\tau, \tau^*)$ or $\mathit{hisuper}(\tau, \tau^*)$, and the overlay type system's *Trans* rule implies that $\tau < \tau''$.

(d) $\Gamma_B \cup H \vdash \tau' < \tau''$ is derived from the *Array* rule. This is not possible because if this were the case, τ' would be an array type, and array types are never the supertype of a class or interface.

Otherwise, one of $hcimpl(\tau, \tau')$ or $hsuper(\tau, \tau')$ must be a fact in H. Then, by one of the overlay $Trans$ rules, we directly get $\Gamma_B \cup H \vdash \tau < \tau''$.

3. **Object.** The *Object* rule is true for every set of facts, so $\Gamma_B \cup H \vdash \tau <$ `Object` and the *Incl* rule can be applied.

4. **Array.** By our induction hypothesis, if the last rule is of the form $\Gamma_B \cup T(H) \vdash \tau[\,] \sqsubset \tau'[\,]$, we know that $\Gamma_B \cup H \vdash \tau < \tau'$. Given this, we can apply the *Array* rule for $<$, giving our goal of $\Gamma_B \cup H \vdash \tau[\,] < \tau'[\,]$.

Soundness Theorem. Given a set of base typing facts Γ_B and additional honorary typing facts H that satisfy Restrictions 1–4 and such that $T(\Gamma_B \cup H)$ is a valid input to I_B, $\Gamma_B \cup H \vdash \tau < \tau' \Rightarrow T(\Gamma_B \cup H) \vdash \tau \sqsubset \tau'$.
Proof. by induction on the structure of the derivation of $\Gamma_B \cup H \vdash \tau < \tau'$. Four types of rules can give a derivation of this form:

1. **Axiom.** If $\tau < \tau'$ follows directly from an *Axiom* rule, the associated *hcimpl* or *hisuper* fact will be converted to a *cimpl* or *isuper* fact by T. Applying *Axiom* and then the *Promote* rule to this fact gives $T(\Gamma_B \cup H) \vdash \sqsubset < \tau'$.

2. **Incl.** If $\tau < \tau'$ follows by application of the *Incl* rule, then $\tau \sqsubset \tau'$ in the base type system by the same derivation, as no rules for $\sqsubset$ make use of honorary facts.

3. **Trans.** Either $Trans$ rule in the overlay type system can be replaced with the single $Trans$ rule in the base type system. For the left subtree, the *hcimpl* or *hisuper* fact is converted to a *cimpl* or *isuper* fact by T. The base type system *Axiom* rule then yields $T(\Gamma_B \cup H) \vdash \tau \sqsubset_1 < \tau'$. The right-hand subtree is a derivation in the base type system and is unchanged.

4. **Array.** By our induction hypothesis, if the last rule is of the form $\Gamma_B \cup H \vdash \tau[\,] \sqsubset \tau'[\,]$, we know that $T(\Gamma_B \cup H) \vdash \tau < \tau'$. Given this, we can apply the *Array* rule for $\sqsubset$, giving our goal of $T(\Gamma_B \cup H)\tau[\,] < \tau'[\,]$.

4.7.2 Non-circularity

Lemma. For all Γ, if $\Gamma \vdash \tau[\,] < \tau[\,]$, then there exists an non-array type $\tau*$ such that $\Gamma \vdash \tau^* < \tau^*$.
Proof. Induction on the structure of the derivation of $\Gamma \vdash \tau[\,] < \tau[\,]$. Since an array type $\tau[\,]$ cannot be the subtype of a *hcimpl* or *hisuper* fact, the only rules that can give rise to the result $\Gamma \vdash \tau[\,] < \tau[\,]$ are the *Incl* rule and the *Array* rule. If the derivation ends with the

Incl rule, then $\Gamma \vdash \tau[\,] \sqsubset \tau[\,]$, implying a cycle in the base type system, which we assume is not the case. If the derivation ends with the *Array* rule, then it must be the case that $\Gamma \vdash \tau < \tau$. If τ is not an array type, then let $\tau^* = \tau$. If τ is an array type, then we can apply our induction hypothesis on υ with $\upsilon[\,] = \tau$, to get a non-array type τ^* such that $\Gamma \vdash \tau^* < \tau^*$.

Non-circularity Theorem. Given a set of base subtyping facts Γ_B and a set of honorary subtyping facts H that satisfy Restrictions 1–4, $\nexists \tau . \Gamma_B \vdash \tau \sqsubset \tau \Rightarrow \nexists \tau' . \Gamma_B \cup H \vdash \tau' < \tau'$.
Proof. By the lemma, it is sufficient to restrict our consideration to non-array reference types, i.e. classes and interfaces. The proof is easily accomplished by dividing the set of reference types into three kinds:

1. the type `Object`

2. interfaces that are the supertype in any honorary relationships and their supertypes aside from `Object` (which we will collectively refer to as "the marker interfaces")

3. all other classes and interfaces.

We can then characterize the types of subtyping relationships within and between these three kinds, and eliminate all possible types of subtyping cycles:

- **No cycle includes kind 1.** The type `Object` has no supertypes in the base type system, and by Property 4, we do not add any in the overlay type system. It thus cannot be involved in any subtype cycle.

- **No cycle includes both kinds 2 and 3.** The only subtyping relationships between kind 2 and kind 3 are ones in which a type of kind 3 is the subtype. These include subtype relations in the base type system Γ_B and also all honorary facts H. As all such relationships only go one way, there can be no subtype cycles involving members of both kinds 2 and 3.

- **No cycle includes only kind 2.** By property 3, no element of kind 2 is the subtype in any honorary typing fact. This means that there are no honorary relationships between elements of kind 2. There can thus be no cycles within kind 2, as the only typing relationships between its members are in the base type system, and we assume that the base type system is non-circular.

- **No cycle includes only kind 3.** By definition, kind 3 does not contain any types that are the supertype of any honorary typing relationship. Therefore, all typing facts regarding solely types of kind 3 are present in the base type system. The honorary typing relationships can therefore introduce no cycles within kind 3.

Chapter 5

Applications of Joe-E

The greatest challenge in using Joe-E is that attaining many of the security benefits requires architecting systems following capability design principles, which are unlikely to be familiar to most programmers. Consequently, using Joe-E effectively will likely require training in capability concepts. Where it is not practical for every programmer to have these skills, it may be possible for someone with such expertise to carefully architect a system being designed as a collection of modules whose interfaces enforce least privilege and thus minimize trust in the modules. Modules that are no longer critical to the application's security properties can then be implemented by programmers with less specialized training, who must just ensure that their code passes the verifier. For those familiar with capability design principles, Joe-E appears to be usable; we did not find the restrictions that Joe-E imposes a serious problem in the programs we have implemented.

Joe-E has been used to build a number of interesting applications. Here, we describe two systems designed to leverage Joe-E's support of isolation and least privilege to improve the security of web applications. First is the Waterken server, which provides secure isolation and cooperation between mutually distrustful web applications written in Joe-E. Second is Capsules, a security-enhanced version of the familiar Java servlet architecture that uses Joe-E to provide isolation between components and users, granting each component only the privileges it needs to perform its function.

5.1 Waterken

In its standard distribution, Joe-E supports design and review of code that is single-threaded, transient and local. The Waterken Server [14] extends this scope to code that is multi-threaded, persistent and networked. Waterken follows the asynchronously communicating event loop model [43]. An *event loop* is a loop that repeatedly extracts the next event off a queue and processes it. Each application object is created within a single event loop, which services all invocations on the object. An event loop and the collection of objects it services is called a *vat*. The vat is the unit of concurrency in Waterken: separate vats may process their events concurrently, but each vat is single-threaded, so two events handled by the same vat cannot be processed concurrently. The vat is also the unit of persistence: after processing of an event has completed, all changes to the vat's objects are written to persistent storage. Vats may communicate through exported references. When a new vat is created, a reference to one of its objects is exported. The object that created the new vat receives the exported reference, enabling it to send asynchronous invocations to the referenced object. An asynchronous invocation is processed by the referenced object's vat as an event, and the return value sent to the caller's vat as an event. The invocation event and return event may also transport exported references, introducing the callee or caller to objects in either's vat, or in another vat.

An instance of the Waterken server can host many vats within a single JVM. An application running on the Waterken software consists of application-specific Joe-E code, running in one or more vats, which may be hosted on a single Waterken server instance or spread across multiple Waterken instances. The implementation of the Waterken server is crafted to ensure that security review techniques for single-threaded, transient, local Joe-E code remain sound when applied to Joe-E application code running on the Waterken platform. To assist verification of its implementation, the Waterken software itself uses Joe-E to prove certain properties of its own implementation and to ensure that assumptions about hosted application code are valid. The following sections examine some of these properties to highlight different "design for review" techniques enabled by Joe-E. The Waterken server comprises about 13K SLOC of Joe-E code and 4K SLOC of Java (excluding blank lines and comments).

5.1.1 Consistent Persistence

Processing of an event by a vat should be like processing of a transaction by a database: either the vat is transitioned to a new consistent state, or reverted to its state prior to processing. Put another way, either all mutations that occur during handling of an event must be persisted, or none of them must be. This consistency is crucial for preserving the security of Joe-E applications hosted on Waterken. For example, in the currency example from Fig. 2.3, if some mutations were not persisted, a malicious client could generate money from nothing by invoking the `takeFrom()` method during processing of an event that did not persist changes to the source `Purse`. Waterken's persistence engine is designed to prevent such violations of consistency. After an event is processed, the persistence engine traverses the graph of all objects that were accessible during processing. Any modified objects are written to persistent storage. If the modifications can not all be committed, an exception is raised and processing of a subsequent event begins by reloading the vat's state from its prior persistent state.

The correctness of the persistence engine depends upon its ability to find all modifications made during processing of an event. This goal is met by requiring Waterken applications to be written in Joe-E. As a result, application code is restricted as follows: `static` variables cannot be mutated or refer to mutable objects; Java APIs that provide access to external resources, such as `java.io.File` constructors, are not accessible, and thus cannot be used to cause unmonitored side effects; and application code is not able to break the encapsulation of objects that implement the persistence engine. These restrictions make it easier to review the persistence engine.

The correctness of the persistence engine also depends upon certain restrictions upon the code that invokes it. The persistent state of each vat is encapsulated in an object of type `Database`. An event, or transaction, is an invocation of the `Database`'s `enter()` method, which takes an argument of type `Transaction`. The `enter()` method provides the `Transaction` object access to the vat's objects and returns an object of the `Transaction`'s choosing. A faulty `Transaction` object could violate consistency by storing a reference to a vat object and modifying it after completion of the `enter()` invocation, or during a subsequent invocation. The persistence engine would then fail to detect the modification since it didn't expect the late modification, or didn't know that a subsequent event had access to the object. A reference to a vat object could similarly escape if used as the return value from a transaction. We use Joe-E to prevent the escape of mutable vat objects by declaring both the `Transaction` type and the return type of `enter()` to implement Joe-E's `Immutable` marker interface. The Joe-E verifier can thus be used to ensure that clients of the persistence engine do not have these faults. All clients of the persistence engine in the Waterken software pass the Joe-E verifier.

In defensive programming, an object implementation normally has sole responsibility for maintaining its invariants. The object's clients are assumed to be buggy or even malicious. In the above example, Joe-E's `Immutable` interface is used to relax this constraint, enabling the `Database` object to depend upon particular client behavior that the Joe-E verifier automatically enforces. Through clever use of a Joe-E-verified property, a design which previously required review of all client code can instead be made defensively consistent, so that we don't need to review the client code.

5.1.2 Cache Coherence

Exported references are accessed remotely using HTTP. An HTTP GET request results in an invocation of a getter method on an object in some vat. The request response contains a representation of the return value from the getter method. To support caching, the Waterken server includes an ETag header in the response. The value of the ETag header is a secure hash of the current application code and all vat state accessed during invocation of the getter method. All GET requests are processed in a Waterken transactional event that aborts if any modifications are made, ensuring that there are no side effects and that the request can be served from cache.

Cache coherence is crucial to application correctness for the same reasons as persistence consistency: either may break code or invalidate security reasoning due to the use of partially stale state. For caching of Waterken server responses to be coherent, the ETag value must fully identify the response text: two responses with the same ETag must yield the same text. For performance reasons, it is best to avoid generating the response text at all when there is a valid cached version. Consequently, the ETag is not simply a hash of the response text.

Instead, the Waterken server leverages Joe-E's support for determinism [21] and so uses the hash of the inputs to the response text generator. Since Joe-E prevents application code from accessing sources of non-determinism, the Waterken server can track all state accessed by the application and thus ensure that any return value is identical to that produced if the same state is used again.

The ability to track all mutable state, together with the prohibition against reading sources of non-determinism, makes any Joe-E computation cacheable and ensures that caches can be made coherent. In the absence of such a systematic approach, caching is often implemented in an ad-hoc fashion specific to a given request. For example, any use of HTTP caching in a standard Java servlet environment requires careful code review for potential cache coherence issues. The difficulty of performing this analysis sometimes results in disabling of caching. Joe-E enables the Waterken server to reliably ensure cache coherency, allowing caching to be enabled for every application.

5.1.3 Remote capabilities

Many of the security arguments that can be made about Joe-E code rely on the capability nature of local object references. Object state can only be accessed via a corresponding object reference. Since object references are unforgeable, encapsulation boundaries can be used to control access to references. Since a reference cannot be represented as data, clients cannot delegate a reference by invoking a method that accepts only data arguments, such as a `String`.

Supporting remote references necessarily requires a representation of an object reference that can be passed as data over a network. The Waterken software enables application code to use remote references that retain the security properties of local object references, by ensuring that data representation of such references is never visible to application code. When processing an invocation event, the Waterken server deserializes the received data into a target object reference, method name and argument list. Any remote reference representation contained in the received data is deserialized to a `java.lang.reflect.Proxy` instance that contains the remote reference data and a reference to the outbound message queue. An invocation on the proxy object enqueues an outbound message, using the remote reference data. The serialization of an invocation ensures a message will only contain a remote reference representation if a corresponding proxy is being sent as an argument. The Joe-E verifier ensures that a proxy object is afforded the same protections as any other object; so a client cannot access the contained remote reference data, nor the outbound message queue. Without access to the outbound message queue, an application cannot construct a working remote reference even if it somehow gained knowledge of the remote reference data. Consequently, a security review can reason about an application's access to remote objects using the same logic applied for local objects. In particular, an application can only access the network if it has received a remote reference; and then only to invoke methods on the referenced object.

By default, the permission tracking enabled by Joe-E does not support tracking of secret data, like strings or integers. The underlying data can be communicated via any method argument, unrestricted by the type system. By encapsulating each such secret in an object, we enable Joe-E style reasoning about the flow of such secrets in a program. In the above example, remote reference data is encapsulated in a `Proxy` object which allows Joe-E code to refer to the data and delegate it, without ever having direct access to the data. This pattern can extend the scope of Joe-E verification to a wide range of data-centric programs, such as those that perform cryptographic operations or generate markup, as the Waterken server does.

5.2 Capsules

The Capsules system [32] is an architecture that uses Joe-E to build a security-enhanced version of the Java Servlet API. It aims to enable programmers to build real-life web applications with high-level security properties that can be easily verified by security review. It aims to make it feasible to conclude with confidence that application security goals are met while providing a model in which programmers can efficiently build real applications.

Concretely, Capsules is designed to support the principle of least privilege for each servlet in the application, provide reliable isolation between user sessions and between servlets. These properties simplify reasoning about the security of Capsules systems and facilitate reviewing them for security.

Capsules relies on Joe-E to enforce isolation between web application components and grant each one a minimal set of privileges appropriate to its functionality. We prevent application components from interfering with each other and restrict the privileges granted to each one in accordance with the principle of least privilege [59]. These two properties serve to prevent vulnerabilities in a web application and limit the consequences of exploiting remaining ones.

5.2.1 Design

Joe-E can reliably isolate objects from each other, and Capsules uses this ability to separate web applications into components which can only interact in proscribed ways, achieving its component isolation property. Each application component forms a protection domain. The extent of a protection domain is specified by the set of capabilities assigned to that component, and by considering which capabilities are shared between domains, we can understand how components can communicate with each other. If two components have disjoint sets of capabilities, we immediately know that they are completely isolated.

In this approach, all application state, including capabilities to application-specific resources, is stored in a per-session data store. This architecture facilitates session isolation. The only way state can be shared between sessions is if shared capabilities are added to the session by an application's trusted session initialization code. This trusted code is a small fraction of the application that must be carefully reviewed. Each application component

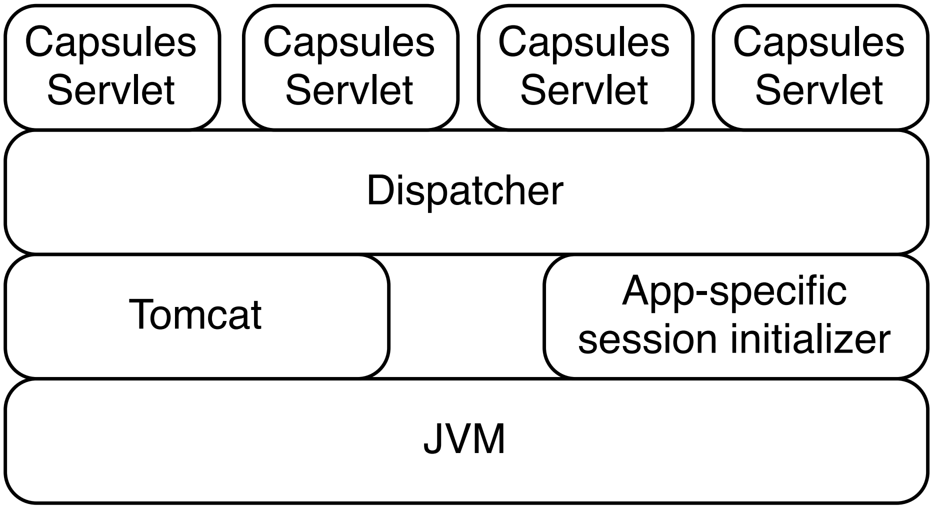


Figure 5.1: Overall architecture of the Capsules application. The dispatcher exposes a modified Servlet API to the application-level servlet instances. The dispatcher is part of the Capsules framework; the application developer writes the servlets and session initializer code. Typically, servlets are written in Joe-E, and the trusted session initializer is written in Java.

has a declarative policy governing its access to the per-session data store. This permits review of communication channels between components and limits how they can interfere with each other. Properly-defined access policies can achieve least privilege and thus facilitate a security review.

5.2.2 Implementation

We implemented a prototype web framework, called Capsules, atop the Java Servlet framework. Capsules introduces an additional layer, the *dispatcher*, between the servlet container and Capsules applications; see Figure 5.1. The dispatcher exposes a modified Servlet API to the application and controls communication between the user and the application, allowing it to provide additional security features not offered by traditional servlet containers.

Isolation

In the Servlet framework, a single instance of each servlet is created and shared across all sessions. The instance variables of servlets can be used to store servlet-local state that is shared across sessions (though this is not a recommended practice), and application developers may not realize that these shared variables can cause concurrency bugs or leak sensitive information between users. This violates our session isolation goal. To eliminate this communication channel, we require that servlets contain no mutable state, which we enforce by declaring the `JoeEServlet` class to implement Joe-E's `Immutable` marker interface (see § 4.3).

This restriction ensures that all application state is maintained in `HTTPSession` objects, which are made available to a servlet when it calls the `doGet` and `doPost` methods. Since servlets cannot maintain state on their own, any objects associated with users must be reachable from that user's `HTTPSession` object. Thus, we can achieve user isolation by ensuring that every object in the `HTTPSession` contains only data that should be accessible to the current user.

Restricted Views

The standard servlet model makes the entire session object and all cookies received available to every servlet. In this model, it is difficult to pinpoint where session members and cookies are used, and it is challenging to fully understand the consequences of modifying or misusing these objects. This lack of documentation complicates security reviews. Capsules provides restricted access to the HTTP session object and cookies in order to reduce privileges granted to components and facilitate review. Joe-E's support for strong encapsulation ensures that the session object can only be accessed through the methods provided in the view. Capsules `SessionView` and `CookieView` classes are automatically generated based on a configuration file, which can be easily audited to ensure that only appropriate session members are made available to each servlet.

5.2.3 Evaluation

To evaluate the effectiveness of our programming model for building secure web applications, we built a webmail application as a case study. While our application is simple, we are able to demonstrate high-level security properties. Our application is implemented as a collection of servlets, where each servlet corresponds to a specific application feature. We wrote 8 servlets for creating user accounts, authenticating and logging out of the application, and for reading, writing, and deleting emails. Each of these servlets defines a `SessionView` and a `CookieView` that we use to restrict the capabilities provided to that component. Users' mail is stored in the file system, in the form of a mailbox directory for each user following the Maildir specification. A top-level `users` directory contains all the user directories. We use Postfix to accept incoming email on port 25 and deliver it to the user's mailbox directory.

We identify two critical application-specific security properties that we want our webmail service to achieve:

1. **Integrity.** If Alice is a user of the webmail service, an attacker who does not know Alice's password is unable to use the webmail service to affect (directly or indirectly) the contents of Alice's mailbox, except by using the webmail service's defined interface to send Alice an email.

2. **Privacy.** If Alice is a user of the webmail service, an attacker who does not know Alice's password is unable to use the webmail service to gain any new information on the content of messages in Alice's mailbox through any overt channel.

We assume that the attacker controls a malicious web client. We allow for the possibility that the attacker might be colluding with the programmers who wrote the webmail servlets (so we do not exclude malicious code from our threat model). However, we assume that the system administrator and platform code (e.g., Tomcat, Postfix) are trusted and not malicious. The privacy property makes no promises about information that might be leaked through covert channels.

To evaluate how effectively Capsules facilitates security reviews, we conducted a security review of our webmail application to verify the two critical security properties listed above. Unlike many security reviews, which often consist of a best-effort search for bugs starting with the most likely places, we instead aimed to convince ourselves that our two critical security properties hold. We constructed an explicit argument that each property holds and then checked that the code satisfies each of the assumptions made by that argument. Due to the isolation and privilege separation properties of the framework, it sufficed to manually review only a portion of the code in order to check each property: we were able to identify a subset of the code that was critical to enforcing each property and then informally reason about these subset.

Verifying Integrity

To convince ourselves that our webmail application achieves the integrity property above, we first identified all application components that have the ability to directly modify the contents of Alice's mailbox by finding all components that can obtain a capability to any file in Alice's mail directory. We found that only the session initialization module and `DeleteServlet` can acquire a capability to modify any such file, the latter only when it executes within Alice's session. We also verify that each session is associated with at most one user of the webmail service and that the authentication logic prevents logging into a session as Alice without knowledge of Alice's password. Therefore, no one else can gain control of a session belonging to Alice and directly violate her integrity.

Next, we check that the attacker cannot indirectly affect the contents of Alice's mailbox. The only way this could happen is if some other session were able to influence the execution of code running in Alice's session, causing that code to modify Alice's mailbox in a way she did not request. Here our basic strategy is to show that the set of heap objects reachable from the attacker's session is disjoint from the set of heap objects reachable from Alice's sessions and then argue that this prevents an attacker from influencing the behavior of code running on Alice's behalf.

Session Initialization We reviewed the application's session initialization code and confirmed that it doesn't use unsafe Java features to violate the isolation properties that Joe-E guarantees for the application code. This enables us to soundly reason about the propagation of capabilities.

The session initialization module is the only application code that can create file capabilities from scratch. We verified that it only uses this power to construct an `Auth` capability, giving it a reference to the `users` directory. The `Auth` capability is then stored in the session object.

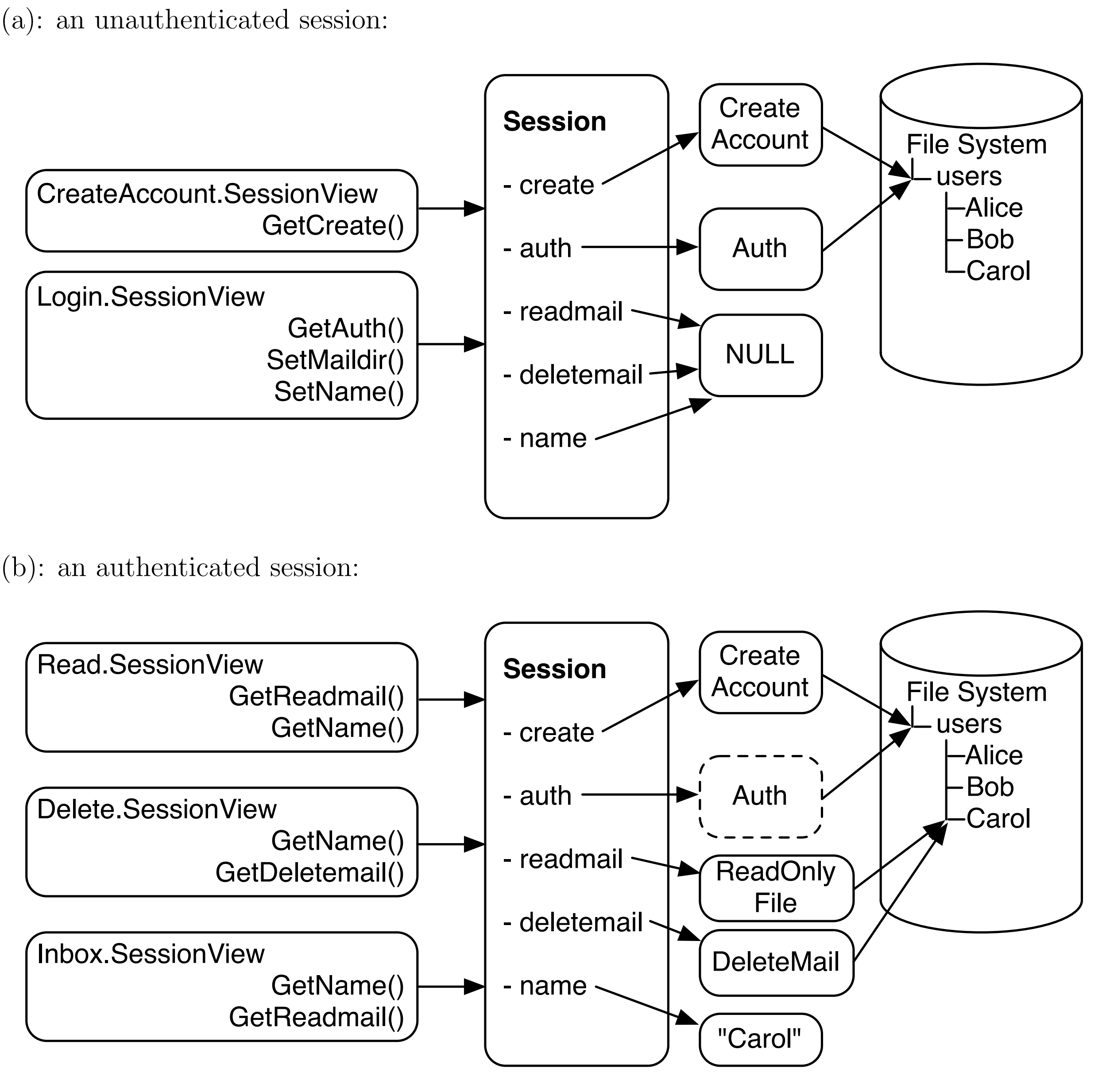
Authentication When a user successfully authenticates and logs into the webmail application, the application populates her session object (that of the session in which the successful authentication occurred) with capabilities to her mailbox files; before then, the session object does not contain any mailbox file capabilities. See Figure 5.2, which shows the application state before and after successfully authenticating a user.

In our framework, the only way capabilities can become available to application code processing a request is if they are stored in the session. Since application code cannot construct new capabilities to the outside world, all external capabilities must derive from the ones initially placed in the session by the session initializer. Therefore, the only way for code to access Alice's mailbox is if its session's initial capabilities gave access to her directory. Due to the immutability of session objects and Joe-E's prohibition on mutable static fields, there is no way to "smuggle" Alice's data or a capability to her mailbox from a previous session.

In our application, there are several capabilities initially stored in the session, but only two can be used to access the `users` directory and thus potentially read or modify Alice's mail: namely, a reference to the `Auth` class, which authenticates a user and retrieves her mailbox, and a reference to the `CreateAccount` class, which can be used to add new users to the system.

In reviewing the `CreateAccount` class, we confirmed it can only create and initialize an account that does not already exist at creation time. It appropriately restricts the user name via a whitelist of valid characters, chosen to avoid creation of multiple names that the Postfix mail transfer agent will map to the same user mailbox directory. It then invokes Postfix to create and initialize the new mailbox directory and updates the Postfix configuration to recognize this username. As a `CreateAccount` object cannot be used to gain access to an existing mailbox, we do not need to review it further to establish the integrity property.

The `Auth` class has a single method which accepts a username and password and returns the user's mailbox directory. We reviewed the implementation of this method and verified that it requires a correct password before returning a capability to the user's mailbox files. Before returning, the `Auth` capability invalidates itself by clearing a boolean flag. This operation is atomic because our framework serializes all requests per session by default. This means that each `Auth` object can only be used for at most one successful authentication. As the session initialization code adds only a single `Auth` object to the session, there is no way for that session to gain a reference to any other `Auth` object. The `Auth` object encapsulates a capability for the user's mailbox files and releases this capability only upon successful



(a): an unauthenticated session:

(b): an authenticated session:

Figure 5.2: Our authentication scheme. A fresh session is preloaded with capabilities that can be used to create a new user account or authenticate a user. However, once a user is authenticated, the `Auth` capability is invalidated.

authentication. There is no other way to gain a capability to a user's mailbox. Thus, during the lifetime of any one session, the session can contain capabilities for at most a single user's mailbox files.

It follows that a web client who does not know Alice's password cannot obtain a session with access to her mailbox.

User Isolation We verified a user isolation property: the execution of a request on behalf of one user (say, the attacker) cannot influence the execution of code running on Alice's behalf in a way that would affect the contents of Alice's mailbox, except for authorized transmission of e-mail messages to Alice. We verified this in two phases: first, we ruled out influence via overt channels by reasoning about all possible overt channels between two sessions; then, we ruled out unwanted influence via covert channels by examining the code running on Alice's behalf.

We eliminated the possibility of unwanted influence via overt channels by reasoning about the possible runtime heap graphs of our application. In particular, the heap graph separates into mostly-disjoint regions. Each region corresponds to a distinct user: the region contains all of the objects transitively reachable from any session associated with that user, except that we prune the transitive traversal at certain *bridge objects*. In addition, there is a set of shared resources that are not associated with any particular region. See Figure 5.3 for an example. Bridge objects are objects that have a capability to a shared resource, an external resource, or an external communication channel. Bridge objects enable a potential communication channel between regions and as such must be reviewed to verify that they do not violate user isolation.

In our application, the bridge objects are the `CreateAccount`, `Auth`, and `MTA` classes. We reviewed their code to confirm that they do not permit unintended communication. `CreateAccount` and `Auth` could in principle be used by servlets to communicate across sessions, but we verified that the servlets do not in fact use this potential channel to communicate in a way that would violate integrity. No servlet listens on this communication channel and uses the message received to influence its modifications to mailbox contents. `DeleteServlet` does not query the existence or non-existence of other accounts or allow their existence to affect the modifications it makes to Alice's mailbox. `LoginServlet` does not use its capability to the mailbox files to modify the contents of any mailbox. No other servlet ever receives a capability to modify the contents of any mailbox file.

Mail transport can also be used to communicate across sessions, but it cannnot violate the recipient's integrity because the `DeleteServlet` does not look at the contents of any mail message before deciding which one to delete.

Since servlets are immutable, they cannot be used as a communication channel between users. Therefore, they are not bridge objects and need not be reviewed when verifying user isolation. As part of the Framework, the dispatcher is assumed not to violate servlet isolation.

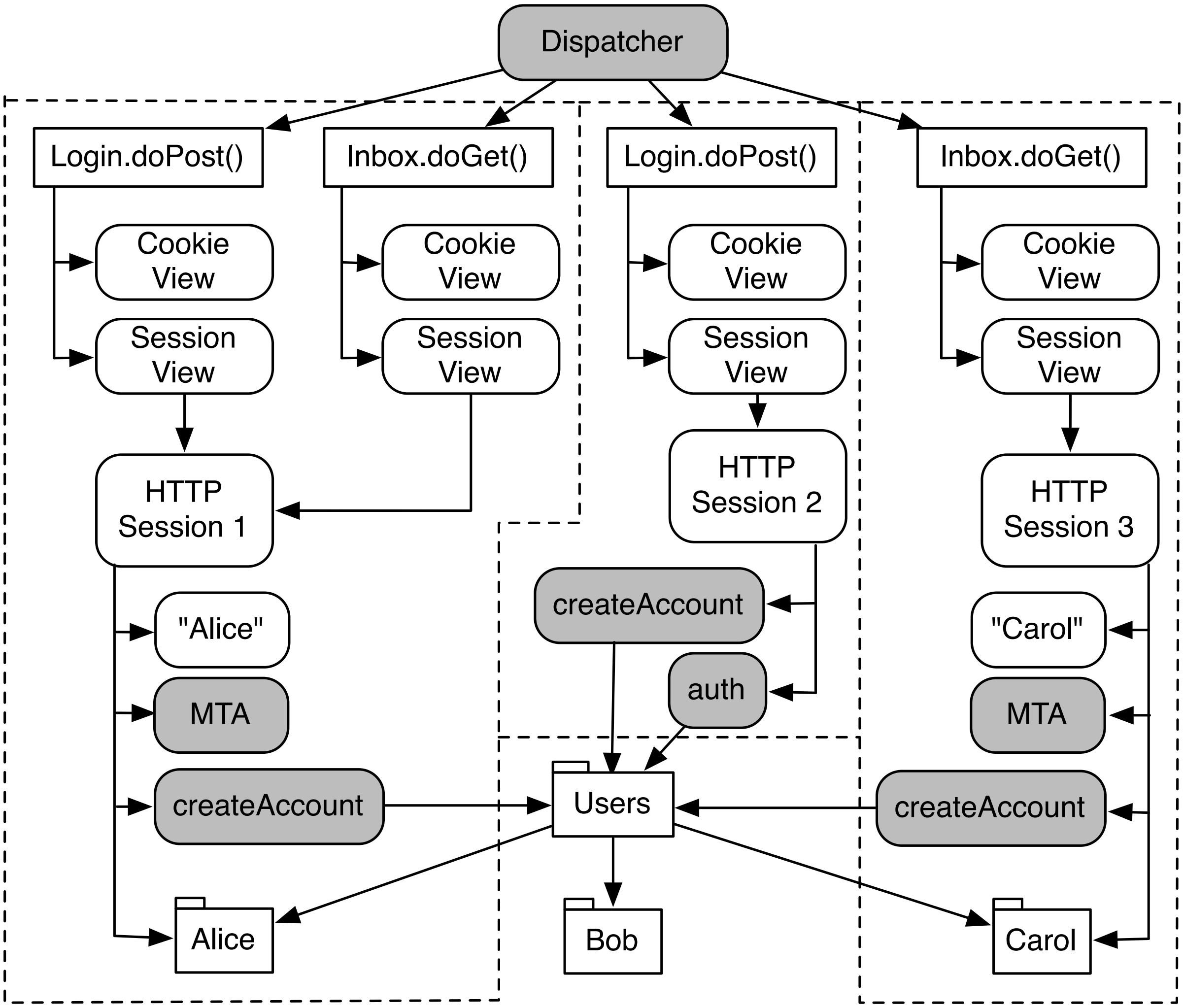


Figure 5.3: A graph of the heap of our running application. Isolation can be verified by examining a limited number of trusted components (shaded) that bridge isolation domains.

Verifying Privacy

To ensure that the privacy of Alice's messages is preserved, we reviewed all application code that can potentially read the contents of Alice's messages. For each such class, we verified that it does not send information about the messages to anyone other than Alice. We check that (a) it does not send this information to other connected web users or out via outgoing email, and (b) it does not communicate any of this sensitive information to other classes, e.g., by storing it in the session. If condition (b) did not hold, we would also have to review any additional code that could read that data.

Immediately after session initialization, with no user logged in, only the `CreateAccount` and `Auth` capabilities grant any access to the user mailboxes in the filesystem. The logic implementing the `CreateAccount` capability ensures that it does not leak sensitive information. It invokes the MTA to add a user for mail delivery and then sends a welcome mail to that user. It reads data from the filesystem only when it checks that a user does not already exist, so it does not reveal any information about the content of messages in Alice's mailbox. Use of this capability is guarded by a lock to eliminate possible race conditions from concurrent creations of the same user name. The `Auth` capability itself never looks at the contents of a user's directory, so it does not reveal any information about the content of the user's messages. Additionally, our prior review of the `Auth` capability showed that it returns a capability to a user mailbox only when supplied with the appropriate password, so an attacker who does not know Alice's password cannot use it to gain a capability to Alice's mailbox.

The only remaining way that Alice's privacy might be compromised is if servlets running on behalf of Alice read her email and leak information about it to the attacker. After authentication, the objects in the session that provide access to her mailbox include the `CreateAccount` capability and the `ReadOnlyFile` and `DeleteMail` wrappers pointing to her mailbox directory. Of these, only the `ReadOnlyFile` is actually able to read the contents of Alice's mail. We therefore need to review only the servlets that are allowed to get this capabiltity, as specified by the policy file.

The policy file specifies that only the `Read` and `Inbox` servlets have access to the `ReadOnlyFile` capability used to read Alice's mail. The `Inbox` reads in the file names of Alice's messages and reads the content of these files in order to get their subject lines. The file names are used for URLs that point to the `Read` servlet; as web browsers do not reliably protect the URLs of pages, these are potentially readable by a malicious website acting in concert with the attacker. While these file names contain timing information used to generate a unique identifier by the local mail transfer agent, they do not reveal any information about the content of the message. The subject line, on the other hand, is included only in the response to Alice as part of a text block in the HTML page sent to Alice. Unlike the URL of a page, the browser's same-origin policy is assumed to reliably protect the content of a page from being read by script originating from other domains. Our webmail application is hosted on its own server, so we do not need to verify any script belonging to other applications. In our current implementation, our application does not have any script content, so we do not need to worry about other servlets being able to read the content output by the `Inbox` servlet. If our application had used JavaScript, we would have to review the scripts to ensure that they do not allow other servlets to read sensitive information from the `Inbox` servlet.

The `Read` servlet reads the contents of a file from the user's mailbox directory, with the filename specified by a GET parameter, and simply outputs the file's contents as a single DOM text node in the response to be sent back to Alice. We assume that the browser does not leak this text to any other domain; that should be prevented by the same-origin policy. If we had JavaScript in our application, we would have to verify that it does not read and exfiltrate the private information output by this servlet, but at present our webmail application does not use any scripts. The `Read` servlet does not make any other overt use of the contents of the email message, and thus does not violate Alice's privacy.

Chapter 6

Related Work: Joe-E

6.1 Capabilities

Capabilities have a long history as an approach for securing systems [35]. Early multiuser capability systems were based upon hardware support for capabilities, where each capability indicated a resource and a set of access rights. These systems were sometimes criticized for the performance overhead imposed by the special hardware, and for the extra complexity of managing capabilities separately. Joe-E minimizes performance overhead by performing security checks at compile time as part of static verification, rather than at runtime. In Joe-E, references used to designate resources also carry the authorization to access those resources, eliminating the need to separately manage privileges.

While hardware support for capabilities is no longer commercially available, capability-based operating systems are still found in research and some commercially-available high-assurance systems, including the GNOSIS kernel from TymShare, KeyKOS [24], EROS [65], and derivatives. We share the view of capabilities as programmatically invokable references, but integrate them into the language.

6.1.1 Object-Capability Languages

There has been a great deal of work on object-capability languages. As far back as 1973, Morris described how a programming language can provide protection features that enable composition of code from multiple sources and support local reasoning about security [45]. W7 implemented these features in a Scheme environment and provided an early example of language support for capabilities [55]. Joe-E was heavily influenced and inspired by E, a seminal object-capability language [43]; Joe-E brings many of the novel security features of E to a modern language (Java) that might be more familiar to programmers, and shows how a static type system can support these security goals. We have also drawn on work in the E language community on recognizing and defining the object-capability approach and identifying patterns for secure programming. Our work is closely related to Oz-E [67], an object-capability variant of Oz, and Emily [68], an object-capability subset of OCaml concurrently developed with Joe-E that follows similar design principles.

Object-capability principles have also been applied to the web. The Caja project [44] provides a way to incorporate untrusted content into a web page, introducing an object-capability subset of Javascript called Cajita as well as support for legacy Javascript code by translating it to Cajita. ADsafe [15] is a more restrictive object-capability subset of JavaScript, designed to support advertisements whose security can be checked without requiring code rewriting. Emily, Cajita, ADsafe, and Joe-E can all be considered examples of semantically-enhanced library languages [69]: they subset a base language, then augment its functionality by adding libraries.

6.1.2 Privilege Separation

Privilege separation is the process of breaking a legacy application into two or more components that can execute at different levels of operating system privilege. A prototypical architecture involves a trusted, high-privilege master process that does most of its work via less-privileged slaves [52]. The privman library [31] factors out much of the logic of implementing a privilege-separated program. The Wedge toolkit [8] aims to facilitate the process of privilege separating legacy software by creating appropriate primitives and providing a runtime profiling tool that identifies the resources used by the components to be separated. We share the goal of architecting systems for security. However, operating system approaches seem best-suited to coarse-grained protection domains; Joe-E provides language support for separating an architecture into many fine-grained protection domains.

6.2 Security for Java and Related Languages

The Java language incorporates mechanisms for access control and protection, based on the security manager, which is invoked when sensitive operations are performed. It can make use of stack inspection and code source information to determine whether to allow such operations [22]. This mechanism provides central enforcement of a security policy, which is usually specified centrally and separately from the code to which it applies. In contrast, Joe-E enforces security polices implemented by the program itself in the form of capability delegation and reference monitors defined by encapsulating objects. This provides an expressive mechanism for supporting a wide variety of policies, including fine-grained and dynamic policies that may be difficult to enforce in Java. It also allows modularity and separation of concerns for policy enforcement, because each part of the security policy can be enforced at the point in the code where it is relevant. We expect Java's mechanisms to be better-suited to enforcing security on legacy code, but for new code, Joe-E may help enforce and verify richer, domain-specific security properties.

Scala [49] is an object-oriented language that compiles to Java bytecode and provides interoperability with Java. It offers better support for functional programming, supporting immutable data structures and event-based Actor concurrency. While we find some of the spirit of Scala in line with the patterns for effective Joe-E programming, it does not provide security properties comparable to Joe-E. Scala syntactically prohibits static fields and methods, replacing them with instance fields on singleton classes. While syntactically cleaner, this approach can still provide capabilities in the global scope.

Another way to enforce application-specific security properties is by restricting information flow between designated security principals or labels. The Asbestos [18] and HiStar [78] operating systems enforce information-flow policies at a per-process granularity.

Like Joe-E, Jif [46] is based upon Java, leveraging programmer familiarity with Java. Jif implements information flow restrictions at a finer granularity, enabling each variable to receive its own label and providing a way to check many of these restrictions statically at compile time. Variable declarations are annotated with labels that indicate an owning principal and a policy for data stored in the variable. The policy can specify (a) the principals whose data the variable may *depend on* and (b) those whose data are allowed to *affect* the information stored in the variable. These restrictions are enforced statically, in cases where it is possible to statically guarantee that the policy is followed, and dynamically, in cases where it is not. Information flow techniques seem most suitable when the security policy is concerned with the flow of data throughout the system; in contrast, capability languages seem most relevant when we are primarily concerned with controlling the side effects that a system can have.

As a special case, it is possible in Jif to specify data flow restrictions that ensure that a particular method is pure. In contrast, while our approach does not allow for the rich policies expressible in Jif, obtaining purity in Joe-E does not require the explicit specification of principals or policies.

6.3 Functional Purity

A number of previous languages, including object-capability languages, have addressed functional purity. Most of these languages are more functional in style than the Joe-E subset of Java, making it easier to limit side effects. In the E language, the `Functional` auditor examines an object to check that “every method on the object has no side effects and produces an immutable result depending solely on its arguments” [77]. The auditor verifies this property using runtime introspection on an object; in contrast, we verify it statically for individual methods. In this work, we specifically build upon Joe-E, but we expect that many of our techniques could be applied to many other object-capability languages as well.

We were inspired by the notion of “environment-freeness” [62], which is essentially the determinism portion of our notion of purity. Environment-freeness was used to verify determinism of a decoding operation and for fail-stop enforcement of the inverse property described in Section 3.2.2.

6.3.1 Side Effects

Most previous work on purity in imperative languages has focused on side-effect freeness and paid little attention to determinism. The definition of side-effect free used has generally been weaker than ours, as it has included only objects in memory and has excluded state external to the program. For legacy code, Rountev [56] and Salcianu and Rinard [58] provide pointer-based analyses that recognize side-effect free methods. Both address only in-language side effects; neither mentions any special treatment for native methods, which can cause external side effects. Analysis-based approaches have the advantage of being directly applicable to legacy code. Language-based approaches, on the other hand, provide more guidance for programmers in writing side-effect free methods.

In Joe-E, we make use of class immutability both in enforcement of determinism and for side-effect freeness. Specific classes in the standard Java type system are considered immutable; standard Java type safety and final field enforcement ensures that objects of such classes are never mutated after construction. An alternative is to use an extended type system that treats some references or instances as read-only while allowing others to be mutated. The C++ `const` qualifier for pointers is the most well-known example of this. It is a compilation error to assign the fields of, or invoke a non-`const` method on, a `const` reference. Its use in preventing side effects is limited because the restrictions are not transitive; it is possible to modify an object contained in a field reached via a `const` pointer. A transitive analogue of this was introduced by the KeyKOS operating system [24] as a "sensory key"; such a key prohibits writes and also causes all keys retrieved through it to be sensory. This concept is also found in the type system of a few programming languages to improve reasoning about immutability and side effects. Such types allow for documentation and modular checking of effect restrictions on a per-function basis.

Ieurusalimschy and Rodriguez [27] use such a qualified type to enforce side-effect freeness in the SmallTalk-like language School. Methods annotated to be side-effect free are type-checked with all arguments and the instance pointer implicitly marked with an `old` qualifier. This qualifier prevents writes to the fields of `old` objects. The type checker only allows invocations of side-effect free methods on `old` objects, and treats the return values from all such invocations as `old`. This ensures that the only non-`old` (and thus mutable) objects that can be used by the method are ones it creates itself. The paper makes no mention of rules for dealing with mutable objects in the global scope or external side effects; their emphasis on soundness would suggest that School has neither. The Javari [71] type system provides similar qualifiers for Java, but instead of having a side-effect free annotation, Javari uses explicit `readonly` qualifiers as a transitive, sound version of `C++`'s const. Like C++, Javari provides a way for fields of a class to be declared as exempt from the `readonly` restrictions.

In addition to a sound, transitive version of `const` with no escape clauses for mutable fields, the D language [9] provides an instance-immutability qualifier `invariant` that can be used to achieve functional purity. Functions marked with the `pure` keyword must have only `invariant` arguments, can only read `invariant` global state, and can only call other `pure` methods. Their compiler restricts `invariant` variables in the global scope to constant values that can be computed by the compiler[1], ensuring determinism. While this approach avoids the need to eliminate mutable state and determinism from the global scope, there is a substantial cost in expressivity as it prevents pure functions from making any use of impure functions and methods. The result is essentially of a partition of the program into imperative and purely functional portions, whereas our approach allows pure functions to make full use of the rest of the program, limited only by the references they hold.

The increased convenience of reference immutability (`const` or `readonly`) over class immutability is attractive, as one can just use the type qualifier with existing classes rather than modifying the type hierarchy. However, class or instance immutability is necessary to ensure determinism in a concurrent program, as otherwise a mutable alias can be used to concurrently modify the object. For non-concurrent programs, reference immutability would be adequate provided that the global scope can only store immutable references. As a general mechanism for defensive programming, reference immutability can only serve to protect the originator of a reference from unwanted modifications; the recipient of an immutable reference may still need to make a defensive copy.

Instance immutability, like provided in D, is an interesting alternative to class immutability that deserves further exploration. For Java, however, the lack of type safety for generics is likely to be an issue. For immutable classes, we perform runtime checks to ensure that the elements placed in an ImmutableArray are actually immutable; this would not be possible with a purely compile-time `invariant` type qualifier as would be required to preserve full compatibility with Java.

Spec# [5] and JML [11, 34] are extensions to C# and Java that allow the programmer to specify invariants on functions and classes. They follow Bertrand Meyer's suggestion that classes and methods should have a contract specified by invariants [34]. They support annotating methods with the `pure` attribute, but purity as defined for JML includes only side-effect freeness and not determinism.

6.3.2 Functionally Pure Languages

In strictly functional languages, like Haskell, nearly all functions are pure. Monads can be defined to allow writing in a more imperative style, in which each operation takes an input state and returns a monad instance that wraps the result along with auxiliary information such as side effects [72]. The monad type defines an operator for sequencing such invocations to obtain a final result; syntactic sugar makes this look like a sequence of imperative statements. While some monads provide a way to retrieve a sequence's final result integrated with any auxiliary information, other monads do not. They are "one-way": once a value is wrapped with the monad, it never comes out. The I/O monad is an example. All functions that potentially expose nondeterminism or cause external side effects use this monad, which allows them to be recognized as potentially nondeterministic. All functions

[1] The D compiler can perform a substantial amount of computation to determine these values, unlike Java's, which only pre-assigns literal constants.

whose return type does not mention the `IO` monad are functionally pure. While monads provide a means to use effects in Haskell, the language is primarily oriented at the functional style. In contrast to a mechanism for imperative patterns in a functional language, our approach is focused on being able to recognize pure methods in an otherwise imperative language. This reduces the changes needed to existing code and programming patterns.

Other systems have mixed imperative and functional programming styles to varying degrees. The Eiffel language [42] separates what it calls *commands* and *queries*. Commands may have side effects, while queries are supposed to be side-effect free. This is, however, only a convention; it is not enforced in any way. Similarly, both Euclid [33] and Spark [4] define two distinct constructs for routines: *procedures* can have side effects, while *functions* are only able to compute a value (and thus are guaranteed to be free of side effects). In Euclid, functions can only import variables read-only, which prevents side effects and ensures determinism if the imported variables (which may be modified elsewhere) are treated as additional arguments. In Spark, annotations on procedures specify exactly which variables can be modified by the procedure, and which variables their modifications are derived from. This and other information flow policies are verified by the Spark Verifier.

6.4 Overlay Type Systems

Our use of an augmented overlay type system follows E [43], which also provides a mechanism for indicating that Java classes “honorarily” satisfy similar object properties, though unlike our approach, in E these properties are not represented by types in the Java type system. The work that most closely resembles our use of marker interfaces is the Auditors framework for E, which uses runtime introspection of an object’s AST to verify annotated properties such as immutability and selflessness [77]. In contrast to that work, we verify similar semantic properties on a per-class basis in a class-based language.

In Joe-E, marker interfaces’ semantic restrictions apply on a per-type basis. For example, specific classes in the Java type system are considered immutable; standard Java type safety and final field enforcement ensures that objects of such classes are never mutated after construction. An alternative, at least for immutability, is to use an extended type system that treats some references or instances as read-only while allowing others to be mutated. The C++ `const` qualifier for pointers is the most well-known example of this. It is a compilation error to assign the fields of, or invoke a non-`const` method on, a `const` reference. Its use in preventing side effects is limited because the restrictions are not transitive; it is possible to modify an object contained in a field reached via a `const` pointer. A transitive analogue of this was introduced by the KeyKOS operating system [24] as a “sensory key”; such a key prohibits writes and also causes all keys retrieved through it to be sensory. This concept is also found in the type system of a few programming languages to improve reasoning about immutability and side effects. Such types allow for documentation and modular checking of effect restrictions on a per-function basis.

The Javari [71] type system provides similar qualifiers for Java; its `readonly` qualifier serves as a transitive, sound version of C++'s const. Like C++, Javari provides a way for fields of a class to be declared as exempt from the readonly restrictions. This is potentially problematic where the immutability of an object of untrusted type is necessary to guarantee a security property. In addition to a sound, transitive version of const with no escape clauses for mutable fields, the D language [9] provides an instance-immutability qualifier invariant that can be used to ensure that a specific instance of an object will never be modified. This has the benefit of allowing the same type to be used in immutable and non-immutable forms. Immutability-Generic Java [79] supports both transitive-const (ReadOnly) and instance-immutability (Immutable) annotations, as an extension to the Java type system. Like Javari, they provide a mechanism for a class to declare fields as exempt from the read-only restrictions, which we would consider a soundness hole. Also in contrast to our work, they do not consider the escape of a ReadOnly reference to a partially-constructed object to be a violation of their immutability property. Pluggable type system frameworks like JavaCop [1] and the Checker Framework [51] can also be used to define type checks that enforce semantic properties such as immutability.

Previous work has addressed the problem of partially-constructed objects in the context of non-null types for instance fields in object-oriented imperative languages such as Java and C#. A number of papers have presented type systems to properly type partially-constructed object instances. Raw types [19] indicate which levels of the type hierarchy have not yet performed their share of an object's initialization. Delayed types [20] are associated with a lexical scope in which the object is not fully initialized, but ensure that all associated objects are fully initialized before the scope is exited. Masked types [53] explicitly specify which fields of an object have not yet been initialized, and thus cannot yet be used. These approaches would provide a more precise way to address the use case of wanting to pass a reference to an object under construction to other objects, allowing for more complex initialization patterns, e.g., creation of circular verifiably-immutable data structures.

Chapter 7

Retrofitting Web Applications for Security Review of Cross-Site Scripting Resistance

7.1 Introduction

Web vulnerabilities such as cross-site scripting (XSS) are a matter of increasing concern, as web applications are adopting a more prominent role in our lives. Attacks that exploit these vulnerabilities can do significant harm to a site's users and reputation. Web frameworks are an increasingly popular tool for creating web applications. They have been successful in introducing abstractions that can cut development time as well as reducing the opportunity for bugs, with a corresponding increase in security. For example, object-relational mapping systems relieve the programmer from the need to write explicit SQL code, providing effective defense against SQL injection attacks. Framework support for automatically inserting and checking CSRF tokens protects applications from cross-site request forgery attacks.

A number of frameworks also incorporate defenses against cross-site scripting. Web template systems, which fill in templates with provided data, are a natural place for these defenses to be implemented. Automatic escaping (autoescape) implementations escape this data by default, in order to prevent inserted content from affecting document structure. However, traditional autoescape systems use a single escaping function for all data, which adequately protects data in some HTML contexts but is insufficient for others. Consequently, traditional autoescape systems are error-prone and require non-trivial manual effort to use securely [75].

Context-sensitive automatic escaping. Context-sensitive autoescape systems improve upon traditional autoescape systems by choosing an appropriate escaping function based on the HTML context where data is inserted into the HTML document. Context-sensitive autoescaping provides a more robust foundation for security, as it can reliably defend against cross-site scripting attacks arising from untrusted data, regardless of context. Context-sensitive autoescaping is a recent development, and has just started to find adoption in production template systems. Given its attractive security and usability properties, we anticipate that more frameworks will support it in the near future.

Retrofitting existing code. We are interested in the problem of retrofitting existing web applications to reliably protect them from cross-site scripting attacks by use of context-sensitive autoescaping. There is a substantial body of code written that already uses templates, and is thus a candidate for migration to context-sensitive automatic escaping. While in principle it may appear straightforward to adopt automatic escaping, in practice doing so is more difficult, even for code that was previously written to target a traditional (context-insensitive) legacy escaping system.

The primary technical challenge is to avoid breaking application functionality, while ensuring that every untrusted value is adequately escaped. If we were to modify the template engine to escape all data that is inserted into the output document, many unmodified legacy applications will break due to over-escaping. Due in part to the inflexibility of traditional (context-insensitive) escaping, legacy applications frequently use “opt-out” mechanisms to turn off autoescaping for some variables. In these cases, programmers use manual sanitization or knowledge about data provenance for protection from content injection. If we blindly apply context-sensitive autoescaping to all data, then these variables will be escaped (or double-escaped) unnecessarily, breaking the application’s functionality. On the other hand, exempting these variables from autoescaping and inserting them into the output as is would frequently result in vulnerabilities, since programmatic sanitization and use of opt-outs is often buggy. In short, we must find a way to identify which variables should be autoescaped and which ones should be exempted.

In this paper, we devise framework enhancements and refactoring tools to address these challenges. Our approach uses a combination of lightweight static analysis and runtime mechanisms to identify which variables do and don’t need to be autoescaped. We implement these strategies in a prototype context-sensitive escaping mode for Django, and evaluate their effectiveness in enhancing security assurance and reducing developer effort.

In this work, we make the following contributions:

- We propose an alternative approach to context-sensitive autoescaping that allows a basic level of cross-site scripting protection for many applications, without any code changes or annotations.
- To support trusted content stored in databases, we present a strategy for coordinating context-sensitive autoescaping with a framework’s object-relational mapper. This supports common web programming idioms without needing explicit trusted casts or other escape hatches.
- We design a mechanism for building up HTML code programmatically, with context-sensitive autoescaping applied. Our design makes it easy for an automated refactoring tool to rewrite existing code to use this mechanism. The resultant code gains the

security benefits of context-sensitive autoescaped templates.

- We include the above enhancements in a prototype implementation of context-sensitive autoescaping for the Django web framework.
- We use our implementation to retrofit eleven open-source Django web applications and demonstrate that the approach is effective at protecting against cross-site scripting attacks with modest developer effort, while simplifying security code review.

In the rest of this paper, we first provide some background on template systems and automatic escaping. We then identify a number of challenges to retrofitting existing applications to use autoescape and our approach toward addressing them. Next we describe our prototype implementation for the Django web framework, followed by an evaluation centered on the results of applying this system to open-source applications. Finally, we compare to related work and offer our conclusions.

7.2 Background

Template systems. A template system is an interpreter for a restricted language: the template language. The template engine interprets a template to generate an output document (in this paper, an HTML document). A template consists of snippets of HTML code alternating with various template directives. The most basic kind of template directive is variable interpolation, which inserts content from the template's arguments into the output document. Template directives also include control constructs such as conditionals, loops, and inclusions of other templates.

A web application is thus a combination of a set of templates (written in the template language) and some code (written in a general-purpose programming language). The code handles an incoming request by computing values and then passing them as arguments to a template that it selects and renders as a response.

Autoescaping. Variable interpolation is a vector for cross-site scripting attacks. Consequently, many template languages provide some kind of support for escaping the contents of a variable before it is output. To prevent the error of forgetting to escape variables, many template systems perform automatic escaping (autoescaping), automatically applying an escaping filter to all variables. This helps prevent XSS attacks. Filters generally strip certain characters with syntactic significance in HTML or replace them with harmless alternatives (e.g., `<` is converted to `&lt;`).

Filtering outputs correctly is nontrivial due to the broad variety of parse contexts in HTML [75]. There are several policies that are appropriate for escaping input strings in HTML, depending on the context in which they are found. Contexts include body text, attribute values, URLs and parts of URLs, and contexts within JavaScript and CSS. For example, a string starting with `javascript:` is harmless in the text of an HTML document, but if used in a URL can cause attacker-provided JavaScript code to execute.

Traditionally, frameworks that support autoescaping use one standard escaping function for all variable interpolations. Generally this function is sufficient for HTML text and quoted plain-text attribute values, but not for other contexts such as URLs or JavaScript code. This provides incomplete protection and can give web applications developers a false sense of security. A template author using untrusted content in a context not handled by the default escaping function needs to be aware of this limitation and must explicitly invoke a different escaping function. The author may also need to explicitly disable the default escaping function to avoid incompatible escaping.

More recently, a few frameworks have adopted context-sensitive escaping, where the template engine identifies the parse context where untrusted data is being inserted and applies an appropriate escaping function for that context. To determine the parsing contexts, such frameworks parse the document to identify where tags, and script and style blocks are. This is a significant gain for security, providing complete mediation for all HTML contexts. It is also good for developers, who no longer have to manually escape data for contexts not handled by default in a context-insensitive system.

Opt-outs. Autoescaping is only appropriate where the data being inserted is intended to act as plain text. Escaping functions, by their nature, disable all HTML markup. In real-world templates, many template inputs are not plain text, and are instead HTML markup that the developer wants to insert intact into the template. Automatic escaping of such data would break the application by over-escaping. For this reason, practical autoescaping frameworks provide mechanisms to disable the automatic escaping operation.

This opt-out can be specified in two different ways. The most direct way is to indicate in the template that automatic escaping should not be applied at a specific variable interpolation point. While convenient for the developer, this is error-prone as there may be multiple sources for the data appearing in a particular template, and some may be more trusted than others. To review such an annotation, it is necessary to identify all places that the template is invoked and verify that the inputs are trustworthy.

An alternative is to have a way of marking a string in application code to be “safe” from automatic escaping when it is used in templates. This is easier to review, as it allows the policy decision to trust the string to be made nearer to the code that justifies this trust (e.g. by the input’s provenance or manual sanitization performed on it). Older autoescape frameworks provide a single “safe” marker that disables automatic escaping, but does not indicate the HTML context that the value is intended for use in. This requires auditing not only that the trust designation is justified, but that marked data is subsequently used in an appropriate context. When context-sensitive autoescape frameworks provide similar functions, they are associated with specific HTML contexts. Once a local audit verifies that the marked string is safe for use in its assigned context, there is no need to check where it is used in templates, as the autoescape framework will detect mismatched contexts.

7.3 Problem

Goals. We aim to take existing programs written in a web framework by authors who are not expert in security and retrofit them for use in a template engine with context-sensitive autoescaping. This process should result in an application that is verifiably secure against cross-site scripting, without breaking or disabling application functionality.

We expect that typically, the developer will have a limited budget or patience for manually modifying code that is currently “working”. Therefore, we want to minimize the amount of manual effort and the number of manual changes that the developer must make; our goal is to automate the process as much as possible. Also, to make it easy for the developer to review and approve the changes recommended by the automated refactoring tool, we want these changes to be as small as possible.

In this work, we focus only on server-side cross-site scripting. We make no attempt to defend against client-side XSS; we leave that problem to others.

Assumptions. We assume the legacy web application makes use of a template system for the dynamic HTML pages that it serves. For strict mode (see Section 7.4.2), we additionally assume that an object-relational mapper is used for all database access. Our approach has no further dependencies on the choice of web framework used, and so is compatible with any framework that provides these features, such as Ruby on Rails, CakePHP, or Struts, and for extension code in sufficiently-featured standalone CMSes such as Joomla.

We assume that the developer is not malicious, but do not assume security expertise. Specifically, we assume that HTML markup found in templates and source code is benign, but we make no assumptions about the application’s ability to secure itself against malicious user input, as XSS bugs are prevalent in legacy applications.

Challenge: determining what to escape. If all templates took in solely untrusted, plain-text input, adopting context-sensitive autoescaping would be easy: we could simply instrument the existing template engine to keep track of the parse state of its output and perform appropriate escaping for the current parse state on each variable. Everything in the template itself would be trusted, all arguments passed in would be correctly escaped, and we would be done.

Unfortunately, it is not so easy. In practice, template arguments are not limited to plain text; some arguments contain HTML fragments that must be output unchanged to preserve application functionality. The code may have built up these HTML fragments programmatically in a way that is safe by construction, or they may have originated from a trusted source.

In templating languages without autoescaping, places where trusted HTML is used cannot be easily distinguished from untrusted inputs where the developer forgot to specify an escaping function. Additionally, even when an escaping or sanitization function is specified, it is possible that it is inappropriate or insufficient for the parse context.

The variable `entry.teaser` contains data parsed from an RSS feed.
Excerpt from the template:

```
{{ entry.teaser|removetags:"script"|safe }}
```

Figure 7.1: An incorrect opt-out allowing cross-site scripting in NiftyURLs. The `safe` annotation instructs Django to disable autoescaping for this value.

In autoescaping template systems, the problem of forgetting to escape inputs entirely is avoided, as inputs are escaped by default. However, when we identify a variable where the application has disabled autoescaping, it is not clear what to do: if we automatically escape it, we risk breaking application functionality, but if we don't escape it, we risk security problems.

Challenge: we cannot trust existing sanitization or opt-outs. In our experience examining legacy Django applications, we found that it is not unusual for them to contain insufficient sanitization. This complicates the retrofitting task. We list two representative examples below, found in the NiftyURLs and Pinax Forum Django applications.

In NiftyURLs, the application code attempts to sanitize content retrieved from an RSS feed by passing it through a special filter to remove all tags of type `<script>` (see Figure 7.1). This is insufficient to remove all scripts because JavaScript can also be invoked through event handler attributes and JavaScript URLs, and both of these would pass unchanged through this filter. After programmatically (and incorrectly) sanitizing the RSS content, NiftyURLs disables Django's context-insensitive autoescaping mechanism for this data. If our retrofitting tool preserved this opt-out, the vulnerability would remain present in the retrofitted code.

In Pinax Forum, we found a more subtle bug (Figure 7.2). The forum allows users to post messages using the Markdown library, which converts sequences such as "`one *fine* day`" to HTML like "`one <em>fine</em> day`". Hyperlinks can also be generated, with syntax like "`[Example](http://example.com)`". Because Markdown will only generate a specific set of HTML tags (and not, e.g., script tags), one might expect that the output of the Markdown processor will be safe for inclusion into the document and free of scripts.[1] However, this expectation is false: because the Markdown library does not limit the URLs that may appear in hyperlinks, the Pinax Forum code can be abused[2] to add a Javascript URL to the page, which will execute script if a visitor to the forum clicks on it. Thus, this opt-out is erroneous as well. If our retrofitting tool trusted this opt-out and preserved it, the vulnerability would remain.

When retrofitting legacy applications, we want to be sure to fix all such vulnerabilities, so that the retrofitted application will no longer be vulnerable.

[1] In its default mode, Markdown allows HTML tags present in its input to pass through unchanged. Pinax Forum HTML-escapes posts before passing them to Markdown, which serves to remove all such HTML.

[2] e.g., "`[Example](javascript:alert%28%22xss%22%29)`"

Excerpt from the Python code:

```
def save(self, force_insert=False, force_update=False):
    ...
    self.body_html = markdown(escape(self.body))
    ...
```

Excerpt from the template:

```
{{ post.body_html|safe }}
```

Figure 7.2: An incorrect opt-out allowing cross-site scripting in Pinax Forum. While one might expect that output from the Markdown library is safe for inclusion in the output document without further escaping, in fact there is a subtle bug.

As these examples illustrate, the not-infrequent need for opt-outs from traditional autoescaping systems leads non-expert developers to sometimes opt out from autoescaping inappropriately. Our evaluation includes seven Django applications that make use of untrusted inputs. Of those seven, at least two are vulnerable to cross-site scripting by way of template variables marked as exempt from automatic escaping (see Figures 7.1 and 7.2). We also saw several instances of opt-outs that appeared to be unnecessary, possibly indicating confusion as to where opt-outs are required or a willingness to add them preemptively.

For these reasons, we cannot trust existing opt-outs in the code we are retrofitting. Ideally, our retrofitted version would not require any trusted opt-outs from the new, context-sensitive autoescaping system. In cases where opt-outs cannot be eliminated, their number should be minimized. Any that remain would ideally be verifiable as correct with straightforward, local code review at a limited number of easily identified audit points. We wish to be able justify the safety of all HTML that is output, so that we can confidently declare the resulting application to be secure from cross-site scripting and other HTML injection attacks.

7.4 Approach

We propose two different ways to defend applications from content injection attacks, *mitigation mode* and *strict mode*. These two modes present a tradeoff between retrofit effort and the security guarantees provided.

Mitigation mode defends against cross-site scripting, but does not prevent other injection attacks, such as user-interface spoofing or site defacement. For many existing applications, however, it can defend against cross-site scripting without requiring any annotations or code changes.

Strict mode defends against content injection more generally, maintaining the guarantee of autoescape template systems that all HTML markup in the output derives from the template or another trustworthy source. This is sufficient to prevent spoofing of trusted UI and defacement attacks that add unauthorized HTML markup to a site. It may require additional annotation and rewriting effort to preserve application functionality while guaranteeing this safety property.

7.4.1 Mitigation Mode

Standard context-sensitive autoescaping not only protects against cross-site scripting, but also against injection of any HTML markup through untrusted channels. If we focus only on cross-site scripting, however, it is possible to be more permissive when handling untrusted template inputs that are marked as exempt from escaping. In a template for a non-autoescaping template language, all variables are effectively opted out from escaping. Instead of escaping all HTML, we can allow some markup while still blocking XSS. Such permissiveness cannot make the site any more vulnerable to content injection attacks than it was originally, as the input value was already opted out from escaping.

The benefit of being more permissive here is that the autoescape system does not need to be as precise in determining what inputs are trustworthy for use without escaping. With a permissive enough policy (provided it still defends against XSS), we have found that it is possible to ensure safety against XSS while preserving much or even all of sites' functionality with no code changes or annotations. A highly-permissive policy that is still believed to block cross-site scripting is the one recommended by OWASP for HTML sanitization. This consists of a whitelist of tags, attributes, and protocols that are generally believed to not execute JavaScript.

We implement this policy as a "mitigation mode" for legacy code that defends against cross-site scripting attacks with no retrofit effort. Do to its weaker security guarantees, we only suggest the use of this mode when protection is desired with no modifications made to application code. Mitigation mode can have false positives, breaking application functionality when template variables contain inline JavaScript, but these situations appear to be rare.

In mitigation mode, we preserve any escaping performed by the existing template engine on variable interpolations. For each template variable occurring in a context where HTML tags can appear, we parse its value at runtime and apply the OWASP sanitizer to any HTML it contains. Using this approach, non-script HTML content from template arguments will be preserved if it was allowed by the original template engine.

7.4.2 Strict Mode

In strict mode, we preserve the invariant that all markup displayed to the user is authorized by the site administrator, or constructed by trusted code written by the developer. This prevents a broad class of HTML content injection attacks including site defacement, content spoofing for phishing attacks, and link spamming.

Conceptually, strict mode identifies which variables can soundly be output without further escaping and which need to be escaped. It then escapes only the variables that need escaping. To identify the variables that should not be escaped, we make use of existing opt-out directives without placing trust in them. Opt-outs indicate locations where the programmer most likely intends for some non-plain-text content to be present, and where the existing code will preserve HTML markup. In the event that we can safely honor opt-outs, we should do so, as otherwise we are breaking the application for no security benefit. For templates in non-autoescaping template languages, all variables should be treated as opted-out, to be escaped only when it is determined necessary for security.

As we do not trust the existing opt-out annotations, we must make our own decisions regarding which inputs if any are trusted for outputting in the current context. In the cases where a value is opted-out for escaping, but we do not identify it as trustworthy for outputting in the current context, we modify the output in order to defend against content injection attacks. This is both the primary opportunity that the new template system has to improve the security of the web application, and also the most likely case for it to break intended application functionality.

When a variable's value must be modified to prevent a possible attack, there are often multiple ways to do so. Existing automatic escape systems operate by applying an escaping filter to all untrusted content. This runs the risk of double-escaping content, particularly if one is modifying the behavior of an existing application in a way that its developer did not anticipate.

For this reason, when escaping opted-out variables, we use an *idempotent* escaping function when one is available. (An easy way to build an idempotent escaping function is to first unescape the value, then escape it.) This will avoid double-escaping in the event that the input has already been escaped. In this way, idempotent escaping simultaneously preserves functionality and defends against injection attacks, whereas standard automatic escaping can only do one or the other.

Database support

In frameworks with an object-relational mapper (ORM) layer, we have a strategy that can help preserve functionality without trusting opt-outs in templates. One major source of opt-outs from automatic escaping is the retrieval of trusted data from a database. It is common for applications to maintain the invariant that particular columns of tables are trusted to contain HTML that is safe to display to the user. (Every application we studied in our evaluation that supported persistent formatted text stored trusted HTML markup in the database.) This invariant usually holds for one of two reasons: either (a) that column is only populated with trusted input, e.g., from the site administrator; or (b) its HTML content is programmatically built up by templates or trusted code. Provided that the invariant holds, data retrieved from such a column is safe to output as HTML without any further escaping or sanitization.

In web frameworks that include an ORM (object-relational mapper) layer, the schema for storing persistent data in a database is defined explicitly within the framework, e.g. with a model class declaration. Objects are used to represent instances of the model, corresponding

```
<html><head><title>{{post.title}}</title></head><body>
<h2>{{post.title}}</h2>
<p>{{post.contents|safe}}</p>
<ul>
{% for comment in post.comments %}
    <li><a href="{{comment.site}}">{{comment.name}}</a>:
        {{comment.body}}</li>
{% endfor %}
</ul>
</body></html>
```

Figure 7.3: A template for displaying a post in a simple Django blog application. Trusted posts and untrusted user comments are retrieved from the database.

to database rows. Fields on such objects generally correspond to database columns; certain fields can contain trusted contents.

We propose that trust annotations be associated with fields in the object-relational model. Applications' implicit trust relationship with database fields can thus be made explicit, where framework code can be aware of these trust relationships.

For example, in a blog, posts can be created and modified only by the site administrator. These posts can be stored in a database model that labels their content field as containing trusted HTML. When retrieved from the database, the content of this field can be automatically cast to a marker type that indicates that it should be trusted by the new autoescaper to contain HTML. This would be sufficient to avoid over-escaping in the simple blog application in Figure 7.3.

In order to maintain the invariant that all data in a particular field is in fact trustworthy HTML that is safe to display without escaping, we propose interceding on all writes to the database initiated via the ORM system. When persisting data for a trusted field, we verify that the data being written is appropriately trusted. In the blog example, any writes to a blog post must be verified to belong to the trusted HTML marker type. If an untrusted value is seen when persisting a trusted field, it will be escaped before being written to the database.

In order to preserve the ability of the administrator to insert trusted content into the database, we propose exempting any administrative interface provided by the application from the write-time check. Trusted content that is suitable for storage in such fields can also be generated as the output of a template. Enforcing this check on writes closes the loop and allows us to soundly treat output from trusted fields as HTML that can be trusted by the new template system.

Programmatic template construction

Even when written in a web framework, applications do not always use the template system when constructing HTML content. Using the template system is preferable for auditability as with contextual autoescaping it is secure by default against cross-site scripting. In the applications we looked at, however, string operations were used fairly frequently to construct HTML. Most such string-based HTML construction was fairly small and self-contained, essentially consisting of a “pseudo-template” where all HTML in the output is derived from string constants in the code.

To handle this case, literal string constants in code that contain trusted HTML can be rewritten to include a cast to the trusted HTML type used by the autoescaper. String operations can be provided as necessary for the building up of HTML in a “bottom-up” fashion. If properly implemented, such code changes can be automated by a static code rewriting tool. The particular set of operations necessary will depend on the language used and its idioms for building up strings. Python, for example, makes heavy use of the `%` operator, which performs `printf`-style interpolation of arguments into a format string. The result of the rewriting will be code that makes use of trustworthy library calls to build up HTML in a safe way. The output of these calls can be automatically cast to the appropriate trusted type used by the context-sensitive autoescaping code, preventing over-escaping.

7.5 Implementation

Our implementation operates on applications written for Django, a Python-based web application framework. Django’s standard template language provides context-insensitive, non-idempotent automatic escaping of variables in templates. Specifically, a conditional escaping function is called on the value of each variable in the template. The conditional escaping function checks to see if the value has been “marked safe”, and if it has not, it replaces the characters `<`, `>`, `'`, `"`, and `&` with encoded HTML entities. This function is sufficient to sanitize values inserted between HTML tags (except within a special tag such as `script` or `style`) or within quoted, plain-text attribute values (i.e., those without special semantics such as URLs, event handlers, or inline styles). The developer can disable Django’s autoescaping for a particular variable in a template by using the `safe` filter.

Our implementation combines runtime mechanisms (including typed anti-taint markers on trusted strings, context-sensitive autoescaping in the template engine, support for inline templates, and integration with Django’s ORM layer) with static analysis (tools to analyze and rewrite legacy code). These mechanisms are detailed below.

7.5.1 Context-sensitive autoescaping

We modify the Django template engine to apply context-sensitive autoescaping. Our template engine dynamically parses the output of a template as it is evaluated, keeping track of the current HTML context. When a variable is output, we apply an appropriate escaping rule, dependent upon the HTML parse context and how the variable’s value has been marked (see below).

7.5.2 Marked strings

Django uses runtime tracking to identify strings that should not be automatically escaped. Django considers a value “marked safe” if it belongs to a special marker class defined by the Django library (`SafeString` and `SafeUnicode`, both subclasses of `SafeData`). These classes are subtypes of the corresponding standard string classes and act like normal strings. Concatenating two objects of one such type yields a result of the same type. Otherwise, there is no special effort made to avoid loss of the “safe” designation. For instance, the marked-safe annotation is lost when a marked-safe string is indexed, joined or truncated using string library functions, or used in `printf`-style string-formatting expressions using Python’s `%` operator.

Marked-safe values can be created using the constructors for the corresponding classes or, more commonly, by calling the utility function `mark_safe` on an existing string. Django templates provide a standard filter `safe` that can be applied to variables, with the same effect as the `mark_safe` method. The Django template engine exempts values that have been marked safe from its automatic escaping.

Our scheme also uses runtime tracking to identify strings that may not need to be automatically escaped by the context-sensitive autoescaper. Since we do not trust the opt-outs in the legacy code, we cannot reuse Django’s marked-safe annotations; instead, we define our own marker classes. Our marker classes indicate the HTML parse context where a value can be safely inserted. The `HTMLString` class is for trusted HTML, and it has an attribute that includes the HTML parse context into which the HTML snippet can be safely inserted. The most common case is content that can be inserted in a PCDATA context, i.e., formatted textual content such as found between tags. We also provide a `HTMLAttribString` class for strings containing attribute=value pairs, as this is a data type we see commonly in Django code that programmatically constructs HTML markup.

Rarely, application code will rely on strings to satisfy application-specific security invariants that are stricter than ensuring that a string is safe for use in a particular HTML context. To facilitate security review of these applications, we provide an `AdminString` marker class for such trusted strings. The `AdminString` marker type indicates that the string’s value was input directly by the administrator or was explicitly cast to the type by trusted invocation of the `AdminString` constructor.

Our types are orthogonal to the Django’s marked-safe annotation: it is possible for content to be marked as safe but not trusted using our markers, or vice versa (e.g., a string might be marked using our `HTMLString` class, but not marked safe using Django’s `mark_safe`). This orthogonality facilitates preserving existing behavior when porting applications to our context-sensitive escaping version of Django’s template engine.

7.5.3 Escaping rules

When a value is interpolated into a template, it may be handled in a number of different ways, depending on its type:

- If it has not been marked safe, it is escaped with Django's standard escaping filter. If this filter is not sufficient, such as for a URL, additional escaping or sanitization is performed as needed.
- If the string is marked safe and is also of type `HTMLString` with a starting context compatible with the output context, it is output without any escaping.
- If it is marked safe, but is not of type `HTMLString`, the output is escaped with an idempotent escape function. This rule helps avoid over-escaping: if the string happens to be already escaped and thus safe for its context, this idempotent escaping does not result in double-escaping, unlike Django's default autoescaping.
- Regardless of type, data interpolated into a context that our prototype does not support is escaped as it would be in Django's unmodified template system and a warning is printed.

With these rules, we use the existing marked-safe annotation as a hint whether or not the programmer believes additional escaping is needed, but we don't trust it. If the programmer has marked a value with `mark_safe`, indicating that she believes no further escaping is needed, and if we can verify that the string is already adequately escaped (i.e., if our idempotent escaping function leaves this string unchanged, or if we have previously marked it with `HTMLString`), then we can use the string as-is, preserving application semantics and functionality.

7.5.4 Database integration

Django's interactions with databases are mediated by its sophisticated object-relational mapper (ORM) layer. The structure of database-persisted objects is defined by model classes written in Python, which specify the names and types of data stored in the model as well as the relationships between models. Instances of a model class represent persistent objects that can be read from and written to the database. The model class defines typed fields that hold persistent objects' attributes. The object-relational mapper layer translates models to database tables, fields to database columns, and model instance objects to database rows.

We extend Django's ORM so the developer can introduce trust annotations on data stored in the database. We define a custom field type, `HTMLField`, that is used to store data consisting of trusted HTML. By changing a field declaration in the model class from `TextField` to `HTMLField`, the developer can annotate that a field is used for storing trusted HTML content. We only support trusted HTML snippets that begin and end in the HTML PCDATA parse context; we have not seen need for anything more. An example of this annotation is provided in Figure 7.4.

When data is read from an `HTMLField`, our augmented ORM layer automatically casts it to the trusted `HTMLString` type in the PCDATA context. It also verifies that all data written to that field of a model instance is of the `HTMLString` type and is labeled to start in the PCDATA context; if it is not, it calls an idempotent escaping function on it first.

We wish to retain the ability for the administrator to change the value of HTML fields

Code defining the original model for blog posts:

```
class Post(Model):
    title = models.CharField(max_length=255)
    contents = models.TextField()
    comments = models.ManyToManyField(Comment)
```

Annotated version indicating that post contents contain trusted HTML:

```
class Post(Model):
    title = models.CharField(max_length=255)
    contents = HTMLField()
    comments = models.ManyToManyField(Comment)
```

Figure 7.4: Original and annotated model for posts of the blogging application example from Figure 7.3.

using Django's administrative interface. In order to achieve this, we apply a runtime patch to the administrative code that casts inputs to these fields to the annotated type before saving the field.

We provide similar support for fields that contain content marked as derived from administrator input. When data is read from an `AdminField`, we automatically cast it to the `AdminString` type. Writes to such a field are replaced by a empty string if the value being written is not an `AdminString`.

7.5.5 Programmatic templating

Web applications sometimes build up HTML using string operations instead of using templates. Our `HTMLString` type supports construction of HTML by concatenation, and supports the combination of trusted HTML components of type `HTMLString` and untrusted strings that do not belong to this type. We implement this by overloading the `+` operator on strings to generate a `HTMLString` as output when we can verify that it is safe to do so. When concatenating two `HTMLString`s, we verify that the labeled starting parse context of the second string is compatible with the ending parse context of the first. When adding a regular string to an `HTMLString`, our current implementation checks to see that the regular string is appropriately escaped for the PCDATA context and that the `HTMLString` is consistent with this context. If this check fails, the result of the concatenation is demoted to a regular Unicode string. A more sophisticated approach that performed idempotent autoescaping on just the untrusted portion would also be possible, but we did not find it necessary to do so for the applications we tested.

Code that builds up programmatically commonly uses Python's `%` operator to perform `printf`-style substitution into template-like format strings. We define a new class, `HTMLTemplate`, to allow such string interpolation to be safely performed on HTML content. Objects of this type are treated as HTML source code with holes that are filled in using the same context-sensitive automatic escaping system that we use on templates. To preserve compatibility with the existing semantics of Python string interpolation, which does not perform any escaping, we treat all interpolation points as implicitly "marked safe". Escaping is only applied where our implementation does not recognize the substituted values as HTML trusted for their parse contexts.

The result of this interpolation operation is a trusted-safe `HTMLString`. This is a sound pattern when the template string is a source-code literal and thus trusted, as the inputs are either verified to be trusted strings used in the correct context, or are rendered harmless via escaping.

Another common idiom calling the `join` method on a separator string to assemble a programmatically-assembled list of HTML snippets into a single string of HTML markup. We provide a replacement for this method that returns a `HTMLString` as output when it is safe to do so. The method takes the separator and pieces to join, verifying that each one is either marked as safe HTML for the context in which it is used, or is a string that does not change when escaped.

We provide a tool for refactoring existing code to use `HTMLTemplate`, `HTMLString`, and `html_join`. We implement it this tool using `lib2to3`, which is a refactoring framework in the Python standard library originally designed to facilitate translating Python 2 code to Python 3 code. It is convenient for our purposes because it has a high-fidelity Python parser that preserves spacing and comments, and supports pattern matching against parsed ASTs.

We look at the literal values of strings to recognize probable HTML belonging to two contexts: PCDATA (HTML-formatted text markup) and HTML attribute-value sequences such as `type="submit" id="submit-button"`. We require such attribute-value sequences to start with whitespace, as this seems to be the common idiom in Django code. (This practice arises from a desire to have clean output when there are no such attributes specified, for example with templates such as `<div%s>`.)

Strings that match either of the above patterns and contain a formatting conversion specifier are recognized as the corresponding template types instead. We replace calls to `join` with equivalent calls to `html_join` if they are directly wrapped in calls to `mark_safe` or if the method they are found in contains calls to the `HTMLString` or `HTMLTemplate` constructors. An example of input to and output from our refactoring tool is provided in Figure 7.5.

We additionally provide a library routine to perform safe regular expression substitution on `HTMLString`s. This is a substitute for the `sub` method provided in the standard Python `re` library. Our safe version extracts the portions that are preserved by the substitution from the original `HTMLString`, treating them as new `HTMLString`s with starting context consistent with their original parsing. The substitute values are then inserted between these pieces, performing the same type checks as if string concatenation or `html_join` were applied to the bits. Provided that the replacement value in the regular expression contains no special characters or is an appropriately-typed `HTMLString`, the end result will remain tagged as safe HTML.

Original code that builds up HTML programmatically:

```
def render_list(elements):
    output = ["<ul>"]
    for element in elements:
        output.append("<li>%s</option>" % escape(element))
    output.append("</ul>")
    return mark_safe("\n".join(output))
```

Version that builds up HTML in a verifiably safe way, as automatically rewritten by our static analysis:

```
def render_list(elements):
    output = [HTMLString("<ul>")]
    for element in elements:
        output.append(HTMLTemplate("<li>%s</option>") % escape(element))
    output.append(HTMLString("</ul>"))
    return mark_safe(html_join("\n", output))
```

Figure 7.5: Example of the transformation of programmatic templates to a safe-by-construction form as automated by our static analysis tool.

7.5.6 Library patching

For templates to render correctly without over-escaping, all sources of trusted markup used by web applications must be marked as such using our new marker types. For cases where this markup derives from templates, database fields, or programmatic micro-templates that we have rewritten, it will be properly marked and thus will not be over-escaped.

Another source of HTML commonly encountered in Django applications is libraries used by web applications. A number of the Django standard libraries create HTML programmatically and many mark the output as "safe".

In our implementation, we perform runtime patching of some standard Django libraries so that methods that return programmatically generated HTML return values of the `HTMLString` type. The libraries patched include the Forms module (which includes methods used for rendering form widgets) and the Django administrative interface code. The main changes here are to code used for rendering form controls used in user-facing Django applications and in the Django administrative interface. For these we wrap return values in the trusted `HTMLString` type.

We also rewrite a utility class in the forms module that takes a dictionary mapping attributes to values and constructs a string containing a sequence of HTML attribute-value pairs. We modify the method to perform safety checks so that the method cannot be used for creating attributes or attribute values that cause JavaScript to execute unless these inputs are marked as trusted. We change the output of this method to be marked as trusted for the HTML attribute-name context.

After rewriting the forms and administration modules, we found that the Django translation system could lose trust markings on strings. The translation system contains a dictionary of translations of strings from a initial language, used in the source code, to potentially a large number of other languages. To support this system, the developer wraps each user-facing literal string in the program with a call to the translation routine `ugetttext` (often rebound to `_`). This causes the string to be looked up in the translation table for the version in the user's language.

These translations are sometimes applied to strings containing HTML formatting, and thus belonging to our `HTMLString` type. We modify the translation method to preserve the HTML string trust annotation when a string is translated. Unfortunately, this means that the translation strings must be trusted as well as the literal HTML text in the program, but there is no clear way around this if translations are provided for HTML strings. (HTML is used in translations to reflect differences in sentence structure between languages.)

7.5.7 Limitations

We currently only fully support safe escaping into HTML contexts, not JavaScript or stylesheet contexts, as we have not implemented a state-exposing parser for these languages. We instead emit a warning when any string is interpolated into contexts that are not yet supported. These contexts have been handled in prior systems [66], and we could support them in our implementation as well with additional engineering work.

Our implementation currently tracks HTML contexts in `HTMLString`s and template outputs in a fully dynamic fashion. While this offers maximal flexibility, one can achieve better performance with a static approach [61]. Our contributions are largely orthogonal to this design decision.

7.6 Evaluation

We ported eleven open-source Django applications to our modified version of Django that performs context-sensitive autoescaping. For each one, we tested its functionality in mitigation mode and also retrofitted it to work properly in strict mode. The first ten applications were selected from the open-source Django web sites listed on django-sites.org. They are the first ten that we were able to get up and running successfully on a stock Django 1.3.1 installation (without any of our custom code). Django CMS was additionally selected as a complex, popular open-source Django application. The applications we examined totaled 23,405 source lines of Python code (40,849 including third-party Django-aware libraries) and 6,278 (10,007) lines of HTML template code[3], as detailed in Table 7.1.

We sought to answer the following questions:

[3] Source lines of code were measured using David A. Wheeler's SLOCCount [76] for Python, with test code excluded. For templates, we counted physical lines of files in template directories with extensions `.htm` and `.html`.

Application	LOC		Library LOC		Audit Pts.	Mitig.	Strict Mode Retrofit Effort			Audit Pts.
Name	Python	Templ.	Python	Templ.	Before	Mode	DB Ann.	Auto.	Man.	After
Adlibre TMS	2,606	187	4,841	170	12	No	0	37	0	0
Douglas Miranda	361	341	2,697	3,507	21	Yes*	1	6	3	0
Fabio Souto	314	504	0	0	5	Yes*	2	0	5	2
Pinax Forum	556	269	0	0	17	Yes	1*	2	2	1
GoDjango	305	255	0	0	1	Yes*	1	0	0	0
JQChat	454	300	0	0	7	Yes	1	3	0	0
NiftyURLs	207	149	0	0	2	Yes	0*	0	2	1
PythonKC	122	193	0	0	5	Yes	0*	0	2	1
Yume Blog	694	250	220	0	9	Yes*	5	0	0	0
Zinnia	4,172	1,933	0	0	36	Yes*	1	22	8	4
Django CMS	13,614	1,897	9,686	52	141	Yes*	2	3	12	2

Table 7.1: Applications we retrofitted to use our context-sensitive autoescape implementation.

How effective is mitigation mode at preserving functionality of the application? To answer this question, we tested whether running the applications in our mitigation mode caused any functionality loss. We consider this effort successful if all functionality appeared to be preserved in a manual exploration of the site that aimed to test all features. Mitigation mode preserves most markup, but blocks all JavaScript and certain nonstandard tags such as `<blink>` when they originate from sources other than templates. For some applications, the only way for this content to occur and get blocked by mitigation mode is for it to be provided explicitly by the administrator. These applications are marked with an asterisk in the table's Mitigation Mode column.

How much developer effort is needed to retrofit applications to strict mode using our techniques? As a quantitative estimate, we counted the number of fields annotated and code changes needed to enable all application functionality we tested in strict mode. (If it is practical to adopt, this mode is preferable because of its stronger security guarantees.) Numbers for code changes are total lines of code added, removed, or replaced, including `import` statements. They are broken down by changes that are automatically suggested by our refactoring tool and those that needed to be manually determined by the developer. All automated changes are safe by nature, as `HTMLString` and `HTMLAttributeString` are only applied to source code literals, and the runtime logic for the other method calls ensures that the all markup in the output derives from `HTMLString`s. Numbers with asterisks indicate the cost after the security bug we found is fixed or, for PythonKC, once a generic HTML sanitizer is used. Qualitatively, we describe the changes in the following section. More details are also given in the sections for each application.

How difficult is it to verify that the retrofitted application is secure? We describe the reasoning needed and code that must be audited for a reviewer to gain confidence that the application is safe from cross-site scripting. We briefly compare to the auditing effort one would incur to verify correctness using a contextual autoescaper that trusted the pre-existing opt-outs. We have counted the number of locations in the retrofitted code that require local auditing to ensure correctness, which can be compared to the approximate number of audit points in the original application. Performing a security audit on the original application may be more difficult than the numbers suggest, however, as we discuss below.

Unfortunately, our verification of security for these applications is subject to the limitations of our infrastructure. As mentioned above, our prototype cannot prevent script injection arising from interpolation into JavaScript or stylesheet contexts. A few of the applications do make use of these contexts, but most do not.

7.6.1 Results

Functionality in mitigation mode. We found that running in our mitigation mode preserved the functionality of all but one of our applications. In the one application that failed, output from a template containing trusted JavaScript is at one point subjected to string operations that result in loss of its `HTMLString` type, leading to subsequent over-escaping. Fortunately, this application could be retrofitted to full functionality in strict mode by our automated code rewriting alone.

In six of the remaining applications, mitigation mode's behavior of blocking all scripts in content that does not derive directly from a template invocation results in the possibility of false positives if administrator-written HTML contains script tags. Five of the six were blogs or similar applications, in which the trusted content takes the form of posts or descriptions. The use of scripts in blog posts is likely to be uncommon, but is supported by the applications when running on the regular Django template engine. The only application where we anticipate mitigation mode causing problems is Django CMS. Here, JavaScript could be more common, as CMS pages are likely to include a broader, more general class of web content than blog posts.

Developer effort in mitigation mode. In our experiments, we found that developer effort to adopt strict mode was modest but nonzero, with most applications requiring at least one database annotation. We believe that these database annotations will be fairly easy for developers to add; they only require identifying which fields can contain HTML. We considered writing a tool to aid in detecting such fields automatically based on database contents, but we think that this task will usually be easy enough that such a tool would be of little benefit.

Four of our applications required no manual changes aside from database annotations, and no application required more than twelve lines of manual code modifications. One application required additional changes due to building up programmatic HTML with code that our purely token-level static analysis was not able to rewrite, but a straightforward intraprocedural data flow analysis would be able to handle. We manually added in the missing casts to `HTMLTemplate`, which all occurred in close proximity to changes automatically suggested by our static analysis tool.

Three applications required a single cast to the trusted HTML type at the point where a library HTML generation or sanitization function was called. One application invokes Markdown on user input. In sanitization mode, the Markdown code is designed to generate safe HTML from untrusted inputs. Two other applications loaded HTML content from an external website or XSS feed and displayed it to the user. Doing this in a secure fashion requires the use of an HTML sanitizer, such as the sanitizing parsing mode of `html5lib`. A cast to the safe HTML type is then required at the point where the HTML generation or sanitization method is called.

Other code patterns that resulted a the need for manual instrumentation were more varied. In one case, an HTML calendar was built using standard library code, which our rewriter could not modify as it was not application code. The same application used custom logic to add a form to the administrative interface in such a way that our implementation did not recognize it as admin-exclusive code. We thus had to manually verify that it was only accessible to the administrator in order to mark the inputs as trusted.

Fabio Souto's blog stores trusted content in the database, but it is not interpreted as just HTML. As additional constraints must be enforced on this data to ensure safety from cross-site scripting, it is not appropriate to use a HTML-based trust type with it. We instead use `AdminField`s for this data; the `AdminString` type is checked before the data is transformed into its final form and cast to the HTML type for output.

Django CMS requires the use of `AdminString` for one field potentially containing trusted non-HTML content. Django CMS also defines a limited pseudo-template language which extends HTML in a limited fashion. Fortunately, `HTMLField` suffices for this field, as the interpreter is easily implemented using safe regular expression replacement on `HTMLString`s.

Security review effort. To estimate the cost of security review for the original applications running on unmodified Django, we count the number of points that must be audited to verify security. These include opt-outs from automatic escaping as well as database writes for applications that store trusted values in the database.

Not only do legacy Django applications have a large number of opt-outs, but the cost of verifying the correctness of these opt-outs may be more difficult than their numbers suggest. In a legacy automatic escaping system, verifying that an opt-out is correct is not easy. It is necessary to determine both where the data is coming from and where it is being used in the output document in order to verify that the opt-out is appropriate. One or the other of these is often nontrivial to determine. For opt-outs in code located near data sources, it is easy to determine the origin of the data. Unfortunately, the auditor must then track down the locations where this data is actually output in order to verify that it is used in an appropriate context. For opt-outs in templates, on the other hand, it can be fairly easy to determine the output parse context. However, it is then necessary to figure out where in the code the template is invoked, what arguments can be passed to it, and where these arguments come from.

In a context-sensitive system in which trusted HTML strings are typed with their intended output context, these complexities are avoided. When manual casts are necessary, data can be programmatically marked with its context at its origin, and the type-checking done by the template engine will ensure that there is not a context mismatch when the data is used in a template.

Making use of trusted data stored in the database is hard to verify without the automatic enforcement of database trust annotations provided by our ORM integration. While our system automatically checks all writes to trusted fields of the database, verifying the correctness of similar reasoning for code running on the standard Django platform would require a code audit of writes to sensitive fields. Absent database instrumentation, uses of `save`, `create`, `get_or_create`, and possibly other library functions that can modify the database must be verified. Calls to these functions therefore become audit points for applications running on the legacy Django platform.

No security audits are needed to ensure security from cross-site scripting in mitigation mode. In strict mode, only explicit manual casts to our trusted HTML types require review for security assurance. Database annotations are untrusted, as the invariant that the inputs are safe for use in HTML is maintained on writes. Our automated code modifications are safe by construction as they only mark static strings in the program as safe HTML, which then only combine with untrusted inputs in a safe way.

Five of the applications were retrofitted to full functionality in strict mode without any trusted casts, and as such are safe without requiring security review. For the remaining applications, it is necessary to justify all trusted casts in order to gain security assurance. The applications making use of libraries that provide safe HTML generation or sanitization are easy to review once the libraries used have been verified to be correct. In a production system, it may be appropriate to provide standard wrappers for such libraries that perform the cast to a trusted type automatically, eliminating the need to audit such trusted casts. The remaining trusted casts in our retrofitted applications can also be justified with local code review. The functions that perform the trusted casts are easily seen to first verify that the input is of a trusted type or that the invoking user is an authorized administrator.

7.6.2 Adlibre TMS

Adlibre TMS is a timesheet and expense-tracking web application. It has screens for data entry and various reports on the entered data. Entry of some kinds of data (such as names of employees and suppliers) is entered using the administrative interface. It stores only plain-text strings and numeric data in its database.

Adlibre TMS was the only application that did not work correctly in mitigation mode. Some custom widgets in this application include inline JavaScript. This code is contained in a template, and thus is marked as trusted, but the output of the template is subject to additional string operations before being returned to the form-drawing code, resulting in loss of the `HTMLString` type on the template output.

Fortunately, the application was easily retrofitted to preserve full functionality in strict mode. Because there is no trusted HTML data stored in the database, no database annotations were required. Adlibre TMS defines custom HTML code for data entry widgets it defines, and a library it includes (`uni_form`) does as well. Our rewriter was able to automatically change all necessary code to use `HTMLString` and `HTMLTemplate`.

Even though this application does not make use of our support for trusted database fields, our approach still simplifies security review. No manual security review is required using our framework. Without our infrastructure, one would need to manually vet the use of `mark_safe` in the five custom widgets defined in the Adlibre TMS code as well as in the libraries used to get similar security assurance.

7.6.3 Douglas Miranda's site

Douglas Miranda's site contains a blog and a list of open-source projects he participates in. Data for both of these sections is added and edited solely by means of the administrative interface. The site has a single trusted HTML field which contains the HTML source code of blog posts. No filtering is done on posts.

The site works correctly with no changes in mitigation mode, with the exception of posts containing inline JavaScript, which is escaped when posts are displayed.

In strict mode, the post content field must be annotated as containing HTML to avoid breaking formatting in blog posts. No other changes are needed to any other Python code in order for correct functioning of the user-facing site.

The administrative interface is not so straightforward. In the `filebrowser` library, a custom widget for selecting files to upload to the website makes use of a programmatic template. In addition to several automated changes, there are three string templates of the form `'%s %s'` that give no intrinsic indication that HTML markup is being composed. We had to rewrite these to `HTMLTemplate("%s %s")` manually as part of the retrofit process. Fortunately, these were all in functions where other strings were detected as HTML and were automatically rewritten. We anticipate that a person reviewing the diffs would easily infer that the additional changes would be necessary.

While the application itself does not use `mark_safe` and has only one use of `safe` in its templates, the third-party libraries it uses have 6 instances of `mark_safe` and 13 uses of `safe` in templates. To verify that untrusted content never makes it into the HTML content field of an article object, one must also verify that the single call to `save` does not take untrusted input. This is straightforward because it occurs in a subclass' override of the `save` method.

7.6.4 Fabio Souto's blog

Fabio Souto's blog is a simple blog with posts authored in the administrative interface. Posts are authored in HTML directly, or optionally using Markdown, with the output of the Markdown compiler being pre-computed and stored in the database along with the Markdown source code. No filtering is done on Markdown input or output or on directly-entered HTML. HTML `<code>` blocks in posts are automatically syntax-highlighted, but only when the posts are displayed.

As above, the site works in mitigation mode, except for posts containing JavaScript, which is saved to the database, but escaped on display. Unfortunately, this application is not as easily handled in strict mode.

The conversion from Markdown to HTML takes place in the post model's `save` method. It checks if the `body` field is empty, and if so, populates it with the result of invoking markdown on `body_markdown`. Code syntax highlighting is implemented as a custom template filter `render`, used when the HTML version of a post is output on a web page. This filter parses the post with BeautifulSoup and performs syntax highlighting on `<code>` blocks in the post using pygments. Along the way, it also runs the post through Markdown (potentially for a second time!), with any code blocks removed so as to avoid it rewriting them.

As a result of these transformations, code modifications are needed to preserve functionality in strict mode, even if both fields are marked as containing trusted HTML. Another problem with using the `HTMLString` type for these fields is that it does not sufficiently restrict their contents to prevent cross-site scripting. As markdown can introduce JavaScript to documents which do not already contain it when parsed as HTML, it is insufficient to verify that the inputs to the `render` filter are of type `HTMLString`. For this reason, we annotate `body` and `body_markdown` as `AdminFields`.

We modified the code that invokes Markdown on the contents of `body_markdown` to check that its input is of the `AdminString` type and to cast its output to the same type. We also changed the `render` filter to check that its input is an `AdminString`, and to cast its output to `HTMLString` for display. Each of these two methods introduces a trusted cast, which can be verified in a security review of that method. The burden of security review for this application is thus still reasonable.

To verify security without database annotations, it would be necessary to gain confidence that untrusted content cannot be injected into either of these fields. In this case, that task is not too onerous due to the small scale of the application. The entire code base contains no calls to `create` or `save` aside from the one in the Post model's custom `save` function. The application contains four uses of `safe` in templates, which would also require audit.

7.6.5 Pinax Forum

The package `forum-pinax` defines a simple web forum, compatible with the naming conventions used by pinax, a system for composable Django components. It makes use of one trusted database field, which stores HTML generated by Markdown.

The code attempts to avoid cross-site scripting by escaping the user's post before it is sent to the Markdown interpreter, but this is insufficient, as in its default mode Markdown can introduce JavaScript URLs even if its input does not contain any HTML special characters.

This application works in mitigation mode with no changes or loss of functionality (except for fixing the cross-site scripting vulnerability). However, in this mode, XSS attacks are persisted to the database and only cleaned on display.

In strict mode, with no code changes we over-escape all messages because the output of the Markdown interpreter is not recognized as trusted HTML. Python Markdown has an option that is intended to ensure safety for untrusted input. After enabling this option to fix the cross-site scripting bug, we can regain program functionality by marking the output of the Markdown call as an `HTMLString`. This is then the only program point requiring security audit.

There are four uses of `safe` in templates, one of which is for the forum posts' HTML contents. The remaining are for explanatory text used in forms; in two of the three cases, this text actually contains HTML. There are 13 invocations of `save`.

7.6.6 GoDjango

This is a simple application for showcasing a collection of videos. Its only data entry is by way of the administrative interface.

It works in mitigation mode without any code changes, but disallows JavaScript that would otherwise be allowed in video descriptions.

In strict mode, it is necessary to annotate the video description field as trusted to avoid over-escaping. Once this field is annotated, the application requires no more changes for functionality or auditing for security using our framework.

7.6.7 JQChat

JQChat is a JQuery-based chat system. Chat messages are stored in a database and are retrieved by JQuery (JavaScript) code on the client using JSON.

The application makes use of a single field that is trusted to contain HTML markup that is transmitted to clients. The application writes to this field using code that explicitly builds up HTML markup. Because the core functionality of the app takes place over JSON, for which we do not have a parser and thus do not run our modified template engine, the application's user-facing functionality works with no changes in both mitigation and strict mode. The administrative interface, in which the history of chat messages is displayed on the page for each room, over-escapes by default in strict mode but not mitigation mode.

To avoid this over-escaping in strict mode (and for aid in auditing security of the full system), the field containing HTML can be marked as a `HTMLField`. The code that writes to the field can be automatically refactored to use `HTMLTemplate`, ensuring that only safe HTML is written to the database.

JQChat does not use `mark_safe` and has only one usage of `safe` in HTML code. This is in a template that is implicitly invoked by admin code, and is fairly easy to audit. There are six uses of `save` to verify, although none are especially difficult.

(We have not evaluated whether the application correctly guarantees correct JSON, and therefore whether it may be vulnerable to a JSON-based script injection attack. Our system could defend against this attack if we included a JavaScript parser, whereas one based on the existing context-insensitive `mark_safe` could not be made to do so.)

7.6.8 NiftyURLs

NiftyURLs is a site that displays headlines from a variety of other aggregator sites. It scrapes HTML-based RSS feeds from these sites and re-displays their contents. It does not make any use of a database, loading the content anew on each page request. As described in Section 7.3, it is vulnerable to cross-site scripting.

Running this application in our framework's mitigation mode protects against this attack, and does not break any of the site's functionality. Mitigation mode's HTML sanitization approach is a good fit for this application.

The vulnerability can be fixed by replacing the `striptags` filter in used in the two templates with one that adequately defends against arbitrary untrusted HTML, such as the `html5lib` sanitizer. The site then works correctly in strict mode if the filter is modified to mark its output as a trusted `HTMLString`.

7.6.9 PythonKC

PythonKC is the website of the Kansas City Python Meetup group. The site consists primarily of data related to upcoming and past meetings retrieved via the Meetup API. Data from Meetup is retrieved via the use of a custom library. This library queries `meetup.com` with appropriate parameters, expecting a JSON result which it parses and uses to construct objects with retrieved data stored in appropriate fields. The Meetup query API does not do any filtering of the data returned from `meetup.com`, which can contain HTML in certain fields.

Presumably to avoid any dependencies on Django, the Meetup-data retrieval library does not mark its outputs as safe. Instead, templates use the `safe` filter when outputting potentially-HTML data from Meetup.

In mitigation mode, the site appears to work correctly with no changes. Meetup itself is likely to prohibit most content that a XSS sanitizer would filter out, so we don't anticipate any content being broken.

To implement the site in strict mode, it would either be necessary to use a HTML sanitizing filter and mark the output of the filter as trusted, or to commit to fully trusting content from Meetup. This could be done by creating a modified or wrapped version of the Meetup library that marks the fields containing HTML as being of trusted type `HTMLString`. Auditing would focus on verifying that the Meetup API gets data from the right source, and the justification for the trust relationship between the site and Meetup. For the numbers in the table, we assume that an HTML sanitizer library is used, providing a location where a cast to `HTMLString` can be added.

To audit the original PythonKC code for security with standard Django, it would be necessary to justify its five uses of `safe` in its templates. This would involve verifying that the Meetup library code only downloads trusted content, and that the display template is only invoked with data derived from this library.

7.6.10 Yume Blog

Yume Blog is the open-source code for Hicro Kee's personal blog. It supports blog posts with summaries as well as undated content pages. All content can only be created and modified through the administrative interface. It includes a comment system that supports plain-text comments.

The site works in mitigation mode, except for posts containing JavaScript, which is saved to the database, but escaped on display. To support JavaScript, field annotations are needed, which are sufficient for the site to work correctly in strict mode as well.

Yume Blog stores trusted HTML in five database fields, divided across three models (post summary, post contents, page contents, site info displayed on main page, and site-wide copyright block). In strict mode it is necessary to annotate these trusted fields. After doing so, the site works securely without any code modifications.

An audit based on Django's safe strings and manual verification that only trusted data is stored in the database would need to audit one use of `mark_safe`, seven uses of `safe`, and one use of `save`.

7.6.11 Zinnia

Zinnia is another blog framework. It supports the MetaWeblog XML-RPC API for management by compatible publishing tools. It uses Django's standard comment module to support plain-text comments. It stores trusted HTML in a single field, the contents of blog posts. These posts are edited solely though the administrative interface.

The entire site works in mitigation mode without any source code modifications or annotations. Posts containing JavaScript cannot be displayed, and have the JavaScript escaped on output.

Marking the single field with HTML content as containing trusted HTML is sufficient for the user-facing code of the site to work as intended, with the exception of a calendar display. This calendar is built by extending an HTML-calendar construction class in the Python standard library to display links to blog posts for each date. Because some of the content construction is still performed by the library code, which does not use our marker types for strings, the output is not marked as safe. (We would be able to automatically rewrite all of this code if it was not in the library.) We thus had to add an explicit cast to `HTMLString` in the application code that requires manual review to validate.

Zinnia provides a form for quickly adding blog posts from the main screen of its administrative interface. This uses a general-purpose form rather than the type of form used only for the administrative interface. Our framework was therefore unable to automatically infer that the content was safe for storage in the HTML-trusted database field. By adding an explicit cast to `HTMLString` for the data submitted via this form, we restored application functionality. A similar situation applies to the inputs received via the application's XML-RPC interface to add and update blog posts.

To review Zinnia under standard Django, we would have to audit 14 uses of the `safe` filter. Excluding test code, there are seven calls to `save` and approximately fifteen to `create` and `get_or_create` that would also require security review.

7.6.12 Django CMS

The Django CMS (content management system) allows a developer to create a website skeleton that can then have content added and edited by way of a web interface. It defines a plugin framework and comes with a set of standard plugins for handling features such as site navigation, images, and Google Maps. A site consists of a hierarchy of pages, each of which is based on a Django template that defines static content and locations in which CMS plugins can be placed. Both the Django admin interface and an in-page JavaScript toolbar can be used to modify and configure the plugins in each template location.

Each plugin defines a subclass of `CMSPluginBase`. This class specifies the plugins database model, which stores data for its instances. It also specifies a template to use for rendering the plugin, and a `render` method that returns a dictionary of variables to pass as arguments to this template.

Running Django CMS in our framework in mitigation mode preserves application functionality, with the exception of plugin contents that contain JavaScript. The standard plugins do not appear to contain inline JavaScript, but the `Snippet` plugin allows the site designer to specify an arbitrary snippet of HTML, which could include JavaScript. Fortunately, this plugin is, according to documentation, designed primarily for prototyping and not for production use.

As a content management system, the majority of the data Django CMS processes is trusted content specified by the site developer. Perhaps for this reason, it does a number of unusual and risky things with this input which our framework does not handle in an automated fashion. Considering this, the manual changes needed to preserve functionality for strict mode are not unreasonable.

Most plugins (including images, file download links, and Google Maps) store data in their database model that is not HTML markup, and thus does not need to be trusted. These plugins work correctly under in strict mode with no modifications.

The Snippet plugin can be used for a literal snippet of HTML, which can optionally be filtered through a specified template. Actually, as an undocumented feature, if a template is not specified, the input itself is rendered as a Django template, and passed two variables that are not obviously useful. The functionality of rendering Snippet's input as a template means that using an `HTMLField` may not be sufficient to prevent content injection. We instead used an `AdminField` for this input. In the case that a template is specified, we cast the content of input field to an `HTMLString` and pass it in as the appropriate argument to this template. If the input is to be itself treated as a template, we wrap the `Template` constructed from it in a `SafeTemplate` (the type used by our modified template engine), so that its output will be trusted HTML and thus not over-escaped.

The Text plugin allows the specification of HTML markup along with inline insertions of other plugins. This appears to be the primary method intended for Django-CMS users to build up site content. There are two different conventions used to represent the location of plugins in Text objects. When stored in the database, the location of plugins is indicated by a string that resembles a Django variable (e.g., `{{ plugin_object 1 }}`). These are converted to image tags depicting placeholder icons in the administrative interface's editor; and when the Text object is saved, image tags of this form are converted back to the database representation. Both conversions take place by way of regular expression replacements, and were easily manually rewritten to use our safe regular expression replacement method. When the `Text` object is rendered, its contents are parsed, with inline plugins being rendered recursively. Our safe regular expression replacement support is able to handle this case as well.

Auditing the security of Django CMS without our platform changes would require validating uses of `mark_safe` and the `safe` template filter, as well as all writes to the database. There are 24 instances of `mark_safe` in the Django CMS code, and 8 uses of the `safe` filter in its templates. There are 89 uses of `save` and 15 of `create` in the code base, though it may be possible to exclude some of these if they belong to code that would never be invoked for the specific Django CMS site being reviewed.

7.7 Conclusions

Contextual automatic escaping is an attractive tool for reliably defending web applications from cross-site scripting vulnerabilities. Unfortunately, programmer-specified exemptions from automatic escaping are common and provide ample opportunities for cross-site scripting vulnerabilities in web applications. Validating such exemptions through code review is laborious with current implementations. Explicitly annotating trust placed in database contents helps to reduce the need for such error-prone, trusted opt-outs. For HTML content built programmatically, we show that framework-provided functions can support patterns that provide similar safety to explicit templates. Retrofitting existing code to use these functions is often easy to automate with simple parse-tree-level source code analysis. With these innovations, reviewable security against cross-site scripting and content injection is practically achievable for many applications.

Chapter 8

Normalization of Web Templates for Reliable Inference of HTML Contexts

8.1 Introduction

The web platform has become a popular way to host sophisticated applications, yet also a major target for attack. As more information is managed by web applications, web vulnerabilities pose an increasing threat to users. Cross-site scripting (XSS) and other attacks that inject undesired HTML content persistently account for a significant fraction of these vulnerabilities.

Web templating languages are a popular mechanism for creating the HTML output of dynamic websites, and thus are a logical place to introduce security mechanisms for defending against content injection attacks. For instance, context-sensitive automatic escaping systems parse the HTML found in the template to identify the parse context of each dynamic variable, and then automatically escape such values at runtime in a way that is appropriate for these parse contexts.

By their nature, such schemes depend on having a reliable parse of the template's HTML. The security of the scheme rests upon the correctness of the parse: if the security tool parses the HTML differently than the browser will, then the security tool might apply the wrong escaping function, possibly leading to a security breach.

As a server-side mechanism, context-sensitive autoescaping requires accurate parsing of the HTML output by templates without help from the client's browser. This is challenging for a number of reasons:

- First, browsers accept syntactically invalid HTML, and web designers often unknowingly include HTML with syntax errors in their templates. The parser must parse even invalid HTML in the same way as browsers will.
- Second, different browsers will sometimes parse the same HTML document differently. This is especially common for HTML documents containing invalid HTML. As the automatic-escaping system operates at the web server, it has to commit to an interpre-

```
<script><!-- document.write("fred@foo" + ".com");
        // hide email from scrapers </script>

{{name}} <script src="widget.js"></script>
```

Figure 8.1: A template with a hole of indeterminate parse context. With Firefox 3.6, the `name` string could end up being treated as either JavaScript or as HTML, depending upon its value at runtime.

tation of the document independently of the browser. This situation appears to create an impossible dilemma for such a system.

- Third, templates interleave HTML with template directives, and the HTML that is produced by a template may depend upon the value of runtime variables. For instance, a template can contain conditionals (if-then-else statements), where a snippet of HTML is conditionally introduced, depending upon the value of a particular variable at runtime. Thus, a single template could potentially generate many different HTML documents, each with its own parse structure. We seek to infer contexts for the template statically, which means that we must anticipate the parse structure of all possible HTML documents that could be generated by the template. A template can have an exponential number of possible paths to consider.

These challenges apply not only to any compile-time system for context-sensitive autoescaping, but also any other system that attempts to parse templates in advance and make security judgements or introduce appropriate security mechanisms. For instance, the same challenges apply to static analysis of web templates.

In this chapter, we demonstrate how to address these challenges. Our approach is based upon *normalization*. We transform the original template into a normalized template that preserves the functionality of the original but makes it easier to reason about the parsing of its outputs. Our normalization corrects syntax errors in the HTML, ensuring that the output conforms to a subset of HTML more likely to be consistently parsed by all browsers, thereby addressing the first two challenges. Our normalization also modifies the template so all template conditionals will respect the HTML parse structure in a clean way, making it possible to anticipate the parse structure of all possible outputs from the template and thereby addressing the third challenge. Our solution provides a principled foundation for static analysis of web templates, and increases the security assurance that can be provided by compile-time context-sensitive autoescaping systems.

Without normalization, the server's and client's view of the document structure may fail to match. One example of this is given in Figure 8.1. According to the HTML5 specification, `{{name}}` is in the HTML body context. An automatic-escaping system based on an HTML5 parser will infer this context for this location, and will apply an escaping function appropriate for the context. Replacing less-than signs and ampersands with their HTML entities is sufficient to sanitize untrusted inputs in this context, and thus would appear to be sufficient escaping for this variable. Unfortunately, it is insufficient in this case as some

```
<h2>{{users.uid.name}}’s Profile</h2>
<img alt="{{users.uid.image_description}}" src="{{users.uid.image}}">
<a href={{uid}}/posts>Posts</a>
```

Figure 8.2: A template with multiple interpolation contexts having different sanitization requirements. This example is difficult for autoescaping systems to handle safely, in general, because of the challenges in escaping unquoted attributes. Our approach solves this problem by normalizing the template to avoid such tricky cases.

browsers can be tricked into parsing the document differently. Given an attack string such as `alert(’xss’); // -->`, which will pass unchanged through the proposed escaping function, Firefox 3.6 and earlier will interpret the script block as ending at the second `</script>` tag instead of the first. This causes the `{{name}}` variable to be treated as JavaScript source code instead of HTML body text. As the extra `</script>` and `<script>` are preceded by JavaScript comments, the injected JavaScript will parse and run successfully.

In addition, some contexts are hard to sanitize with confidence without rewriting. The `{{uid}}` variable in Figure 8.2 appears in an *unquoted* attribute, which is a particularly difficult context to sanitize safely. One might think that escaping single quotes, double quotes, and whitespace characters would be sufficient, but this turns out to be not the case. An attack string like `‘‘onclick=alert(1)` can break out of an attribute and define an additional attribute-value pair in Internet Explorer. IE treats the backquote character ‘ as an additional quoting character, and allows spaces to be omitted after a quoted attribute. Thus, this obscure feature of Internet Explorer can be used for cross-site scripting attacks if the template in Figure 8.2 is not handled very carefully. In some browsers, other characters may also terminate an unquoted attribute value. This makes it very challenging to devise an escaping function that will make it safe to interpolate data into an unquoted attribute. Instead, a better solution is to rewrite the template to avoid difficult cases like this. That is the approach we follow: we normalize the template so that (for instance) all attribute values are quoted.

These examples illustrate the challenges of inferring how browsers will parse an HTML document produced by a template. Being able to infer the parse context of variable inclusion reliably at the server is crucial for a number of security analyses and applications. Normalization of templates is necessary to ensure that parse contexts where dynamic values are inserted into the template are accurately characterized. This is the problem we solve in this paper.

The contributions of this work are:

- We propose a well-formedness property for HTML that to our knowledge ensures consistent, predictable parsing across all browsers in common use, including pre-HTML5 versions.
- We identify a strategy for subsetting template languages that ensures that all possible HTML documents generated from templates will satisfy the well-formedness property

mentioned above.
- We develop methods to transform existing template code so that it falls within the above subset.
- We implement our methods and evaluate their applicability to real-world open-source Django templates.

8.2 Problem

In this paper, we focus on the problem of static analysis of web templates to determine how content in the template will be parsed. In particular, we want to, given a set of web templates, *normalize* them into a form that preserves the content output by the template, while allowing the parse contexts in the template to be reliably determined by the server. Specifically, given a normalized template, it should be easy to determine, for every variable in the template, the parse context in which that variable's value will be interpreted by the browser.

We can formalize the problem as follows. A template T can be considered as a function that accepts a set of runtime inputs I and produces an HTML document $T(I)$. We wish to find a normalization algorithm N and a simple, efficient static analysis S that satisfy the following properties. Given a template T, let $T' = N(T)$ denote its normalization under our normalization algorithm. We define the static analysis S so that $S(T')$ predicts a parse context for every fragment of T'. We want the static analysis to be both sound and complete. By *sound*, we mean that for every possible template input I and every possible prefix P of the HTML document $T'(I)$, we want the parse context after the browser parses P to be as predicted by $S(T')$. By *complete*, we mean that the static analysis S predicts a parse context for every point within T'. These guarantees are only required to hold for normalized templates T' output by the normalization algorithm N.

Our motivation comes from context-sensitive automatic escaping. Context-sensitive autoescaping is a powerful tool for defending web applications from cross-site scripting and content injection attacks. A static context-sensitive automatic escaping system must select, for every variable interpolation within the template, a suitable escaping function to apply to that variable at runtime. The escaping function needs to be chosen so it will remove any character sequences that will be interpreted by the browser as contributing to the syntactic structure of the document. In order to determine the correct escaping function to apply, the autoescaping system needs to be able to predict how the document will be parsed by the end-user's browser. In other words, we need a way to predict the parse context in which each variable will be parsed, when its value is inserted at runtime. Moreover, if the escaping function is selected at compile-time (before the values of the variables are known), the prediction must be correct for every possible invocation of the template (i.e., for all possible values of the variables). Thus, any robust static autoescaping system needs to solve the problem we have articulated above.

Dually, autoescaping supports static analysis of templates. We assume in this work that

```
<html><head><title>Parent template</title></head>
<body>
{% if foo %}<script>{% endif %}
{{content.possiblyscript}}
{% if foo %}</script>{% endif %}
</body></html>
```

Figure 8.3: The control directives in this template mean that the parse context of the variable `{{content.possiblyscript}}` varies, depending on the value of `foo`.

the runtime values of all template variables are escaped appropriately for the parse context into which they are interpolated. This assumption can be discharged by use of a context-sensitive autoescaping system. (Without this assumption, it would be impossible to predict the parse state immediately after any template variable.)

8.2.1 Basic HTML Normalization

The simplest possible kind of template consists solely of static HTML content: no dynamic content, no variable interpolations, and no template directives. The problem is already non-trivial even in this simplest case. Because different browsers may parse the same HTML document differently, it is not always possible to predict the parse context for every point within the document. This is particularly common for malformed HTML documents, as different browsers employ different strategies in guessing what the document author meant. For historical reasons, sometimes browsers employ these strategies even on well-formed standards-compliant documents.

8.2.2 Control Directives

Full-featured web templates pose a number of additional challenges beyond normalization. Control-flow directives, such as if-then blocks, are an important feature of many template languages. Each branch of an if directive corresponds to a different path through the template (a different sequence of static content and variable interpolations). In the general case, which of the two paths are taken at the location of the if can affect the context for variables in the template that follow the if block, as in Figure 8.3. This is problematic as would prevent us from uniquely identifying the parse context of that variable.

Control-flow directives are also challenging because they can cause exponential growth. A template with n if-then blocks can potentially have 2^n different paths, each of which may correspond to a slightly different parse structure. Thus, it is not feasible to exhaustively enumerate all possible paths: we need a general algorithm that scales better.

8.2.3 Template Inheritance and Inclusion

In addition to control statements, a number of templating systems include constructs that permit reuse of template code. For example, *template inheritance* allows for a template to be extended by child templates. A template can define named blocks and child templates can override those blocks. When the child template is invoked, the parent template is first loaded, but with the child's versions of any named blocks replacing those of the parent. Child templates can extend templates that are themselves children, resulting in a tree-shaped inheritance hierarchy. Many languages also support *template inclusion*, in which specified locations in a template are filled in with the output of another specified template.

As a result, the parse context for variables in a particular template file can potentially vary from invocation to invocation, if its code has been included or overridden by other templates.

8.2.4 Deployability

Our techniques should be applicable to legacy code: we envision the normalizer would be applied to existing templates to produce a diff, and developers would be asked to apply the normalizer's suggested changes. For developer acceptance, normalization should avoid unnecessary changes to the templates. Thus, the number of changes made by the normalizer is an important metric by which we evaluate our scheme.

8.3 Approach

Our approach is based on normalization of both HTML and templates. We identify a conservative, well-understood subset of HTML that we believe will be consistently parsed across browsers (Section 8.3.1). Additionally, we identify a set of syntactic locations in HTML where appropriately-constructed content can be inserted without changing the state of the parser. In order to ensure that template constructs preserve the ability to unambiguously parse documents statically, we aim to rewrite existing templates into the subset of the template language in which control constructs appear only at these locations and the HTML content in each branch returns the parser to its initial state before the branch. This ensures that template directives are "well-nested" with respect to the document's parse structure. Normalization also ensures that the output document is in our HTML subset, that all tags are properly nested, and that all open tags are closed in the proper order.

8.3.1 HTML Normalization

We normalize HTML to ensure that the document will be syntactically well-formed with respect to the HTML4 and HTML5 specifications. In particular, it is well-formed SGML that falls within the subset identified in the HTML4 spec as supported in practice by browsers, and can be parsed by HTML5 parsers without triggering a parse error. While this

- all tags are properly opened and closed (except for established self-closing tags)
- all tags are properly nested
- all attributes values are quoted with standard single or double quotes
- all attributes are separated by at least one character of whitespace
- all less-than and greater-than signs in the body of the document are HTML entity-escaped, including in attribute values
- all entity escapes are correctly formed
- all comments begin with `<!--`, end with `-->`, and have no intervening sequences of two consecutive hyphens, ensuring that they are valid and end at the same character under all versions of HTML and SGML
- a warning is issued if an IE conditional comment is seen
- script blocks do not contain any comments, nor the sequence `</` in their bodies

Figure 8.4: Normalization invariants enforced by our rewriter.

normalization falls short of full validation, it should be sufficient to avoid confusion about the intended document structure for any parser written to implement *either* the HTML4 or HTML5 specification. We perform additional normalization to account for known deviations in the parsing of well-formed input by popular browsers.

The HTML subset that our normalizer rewrites documents to is summarized in Figure 8.4. We describe how we rewrite existing documents to this form below.

Our basic approach is to parse the HTML following the HTML5 standard to generate a DOM. Then, we serialize the DOM to an HTML document. This step gets us most, but not all, of the way to ensuring that the result is well-formed.

The HTML5 specification performs a number of transformations to a document in the process of parsing it into a DOM tree, and serializing it back out to HTML. The output of this process is is a document that should be treated the same as the original by an HTML5-compliant browser, and which additionally satisfies a number of well-formedness properties:

- All tags will be closed: they will have an explicit matching closing tag, except for self-closing tags such as `img` that do not require one.
- Tags will be properly nested. (This means that there are no interleaved tags such as `<b>bold<i>bolditalic</b>italic</i>`.)
- Content in invalid locations may be *foster-parented* and relocated in accordance to the HTML5 spec.
- "Bogus comments" of the form `<!foo bar>` are converted to well-formed comments (`<!--foo bar-->`).
- Less-than signs that do not start tags (but are found in contexts where tags could exist) are converted to `&lt;`.
- Attribute values that contain whitespace and certain other characters will be quoted.

We invoke the HTML5 reference parser with carefully-selected options to increase the degree of normalization it provides. This ensures the following additional properties:

- All "optional" tags that were implicitly added to the DOM will be explicitly included in the output.[1]
- All attribute values are quoted with single or double quotes.
- Less-than signs are escaped in attribute values.

Standards before HTML5 have strictly limited the set of characters that are allowed in unquoted attribute values and suggest that all attributes be quoted for maximum compatibility. Handling of unquoted attribute values that contain characters not on the list was not standardized and varies substantially among older browsers. While HTML5 provides more detailed rules for parsing unquoted attributes, but they are not necessarily consistent with all legacy browsers. In order to ensure that the output is correctly parsed by pre-HTML5 parsers, we specify the parser option that ensures that all attributes are quoted with standard quoting characters (`'` or `"`), including where the default behavior would be to leave them unquoted.

Certain tags, such as `<noframes>` and `<noscript>` are traditionally not parsed when the associated condition is not met (e.g., by a client that supports frames and has scripting enabled). Instead, a simple textual search is made for the matching close tag. In order to avoid the possible ambiguity posed by this parsing inconsistency, our normalization process aims to ensure that the string `</` only occurs in properly matched closing tags. We enable an option that esapes less-than signs even in quoted attribute values, where the default is to leave them in, to eliminate the sequence `</` in these locations. We also modify the parser to escape greater-than signs in attribute values, as recommended in the HTML4 specification to avoid any confusion in older browsers as to where tags end.

The HTML5 specification defines three classes of HTML documents: valid, invalid but free of parse errors, and documents with parse errors. While the specification describes rules for parsing all possible documents, it specifies that HTML5 parsers are permitted to abort and refuse to process a document when a parse error is encountered. We would like to ensure that the document does not have any parse errors. Even with the options we specified, parsing an HTML document with the HTML5 parser does not guarantee that the resultant DOM structure, or its serialization, is free of parse errors.

For instance, tag names containing invalid characters are preserved, even though they could confuse some parsers. Tags may still appear in invalid locations that would continue to result in inconsistent parsing. An absence of reported parse errors is unfortunately insufficient to ensure normalization. As our output is generated by serializing an HTML5 DOM tree, in order to ensure that it is well-formed at the tokenization level, it is necessary to ensure that tag and attribute names only contain valid characters. For some malformed documents, the reference implementation will generate DOM trees with tag and attribute names containing invalid characters. We modify the HTML5 reference parser to strip invalid characters appearing in these locations.

[1] We modify the parser to avoid inserting `<tbody>` tags. This tag was introduced relatively recently in HTML4, and is usually omitted by developers.

Skipped Blocks

For consistency with legacy browsers, the HTML5 parsing rules do not interpret the contents of `<iframe>`, `<noembed>`, `<noframes>` and `<noscript>` blocks, their contents being treated as a blob of "raw text" terminated by the appropriate closing tag. This shortcut of not actually parsing the content of such blocks can lead to documents whose structure will be different if the block is parsed, e.g. if scripting is disabled. This occurs because strings like `</noscript>` can occur in HTML without necessarily closing the current `noscript` tag. This can occur when it appears in a comment, quoted attribute value, or unparsed context such as JavaScript or inline CSS. Additionally, a document could have nested `noscript` blocks, causing the first end tag found by a simple string search to be the wrong one. If the content of the `noscript` block is treated as raw text, as in the HTML5 spec, serializing the parsed DOM will not resolve this type of ambiguity. (The HTML 5 spec seems to assume that all browsers support frames, providing no rules for parsing a document if frames cannot be supported. While this is a reasonable assumption for current and future desktop browsers, some mobile browsers lack frame support. Browsers that do not support frames are likely to support the existing, well-established `noframes` tag as a way to display appropriate error messages or links to alternative content, in contrast to the HTML5 spec that indicates that the tag's contents should not be interpreted.)

Our modified HTML5 parser parses the contents of such blocks and ensures that they are normalized in the same fashion as the rest of the document. We additionally check that, in the retrofitted output, nothing within such blocks looks like a possible closing tag for the block. This ensures that the legacy parsing behavior of a simple textual search for a closing tag will end the block at the same point as parsing its contents.

Comments

Comments are another case in which real browser implementations (which are not all HTML5-compliant) are likely to differ in their parsing behavior. HTML comments have historically been poorly specified. They are based on SGML comments, which until HTML5 served as the closest thing to a thorough specification. In the simplest case, comments start with `<!--` and end with `-->`, but the SGML spec has counterintuitive rules that permit just a greater-than sign to end the comment depending on how many `--` sequences precede it. None of the early browsers implemented this full specification. Most just searched for the sequence `-->`, or the first `>` after at least one `--`, as currently mandated by HTML5. When Opera and Mozilla tried to follow the SGML spec more accurately, it broke many sites, and had no benefit since essentially no web authors were aware of or made use of SGML comments in their full generality. For comments that are in a standard HTML context, i.e. where HTML tags and displayed text can appear, we ensure that comments are in the most standard form: they start with `<!--`, end with `-->`, and contain no intervening sequences of two consecutive hyphens. This matches the examples of well-formed comments in the HTML 4 specification [54, §3.2.4], though the specification also permits whitespace between the `--` and `>` of the end of the comment. It does not explicitly prohibit hyphens within the comment but suggests that two or more adjacent hyphens inside comments should be avoided.

Comments in `script` blocks are particularly challenging to parse, as browsers use complicated heuristics to determine the end of the block while allowing the string `</script>` to occur in certain places in the script code. The HTML4 spec implies that browsers do not need to support script blocks that contain even the two-character sequence `</`, but actual browsers attempt to handle HTML markup, including closing tags, in script blocks. This markup occurs primarily in JavaScript strings representing HTML that is dynamically written to the document. Most browsers support matching `<script>`/`</script>` pairs within a script block, at least within a comment. (It is traditional to place all of a script block's code inside a comment in order to prevent it being rendered as visible text in ancient browsers that do not recognize the script tag.) This means that simply searching for the string `</script>` is insufficient to determine the end of a script block.

HTML5 specifies that `</script>` should not be interpreted to end a script block as it normally would under certain circumstances. Essentially, it does not end the block when it is inside a comment and follows something that looks like an opening `<script>` tag with no intervening `</script>`. This is a rule adopted in the belief that it closely resembles the rules of mainstream browsers, but differs at least from Firefox in certain corner cases. In contrast to this rule, the pre-HTML5 parser in Firefox appears to allow an unbounded number of `</script>` sequences within a comment, provided that the comment can be parsed and is followed by a valid `</script>` tag.

We avoid this inconsistent behavior by ensuring that there are no comments and no occurrences of the character sequence `</` within JavaScript blocks. We accomplish the former by stripping any `<!--` that occurs at the beginning of a script block (possibly with intervening whitespace) as well as any `-->` that occurs at the end of a script block. The sequence `<!--` detected elsewhere in the script block will cause our retrofitting tool to fail. In order to remove the sequence `</`, we replace it with `<\/`, which preserves functionality in the likely case that the sequence occurs within a JavaScript string. In the uncommon case that the sequence `</` occurs within a regular expression literal, e.g. `/</`, this heuristic will break the regular expression. Such a regular expression can be manually rewritten by replacing the `<` character with the unary group `[<]`.

IE Conditional Blocks

Internet Explorer supports a nonstandard conditional block construct that allows tests against parameters of the browser environment, such as the IE version number. These blocks are defined to be HTML comments, and other browsers usually interpret them as a regular comment, having the effect of never displaying their content. They start with the sequence `<!--[` and end with `]-->`. These are potentially unsafe to ignore when normalizing documents, as they can open or close tags, leaving the browser in a different parsing context (at least in terms of which tags are open) than if they are skipped. We warn if any such comments are encountered, but do not currently attempt to handle these further.

IE also supports "downlevel-revealed" conditionals, in which the start and end delimiters for the conditional are "bogus comments" that start with`<![if` *some test*`]>` and end with `<![endif]>`. These are also problematic because this type of comment is not valid HTML and thus may not be treated the same by all browsers. Most browsers treat them as the HTML5 spec does, namely treating it as a comment that terminates at the first `>`. Even if all other browsers treated these comments identically, however, IE will skip the contents between the start and end conditional delimiters if the condition is not satisfied. We covert these to standard comments in the process of normalization, which preserves the behavior of standards-compliant browsers but has the possibility of breaking IE.

XHTML Features

Web pages can be served as HTML with the `text/html` MIME type, or as XHTML, as indicated by the `application/xhtml+xml` MIME type. XHTML documents can also be served as `text/html`, provided they can be correctly parsed as HTML. Not all XHTML documents can be parsed correctly as HTML. Some will be parsed incorrectly if served with an HTML MIME type, due to use of features present in XML but not HTML.

In XML (and thus XHTML), any opening tag can be converted into a self-closing tag by including a `/` before the `>` that ends the tag. In HTML4, such trailing slashes are traditionally ignored by browsers, making the self-closing form "work" for tags that are automatically self-closing in HTML such as `img`. Technically, however, they inherit a different meaning from SGML, the "NET shorttag" syntax, whereby markup of the form `<foo/bar/` is equivalent to `<foo>bar</foo>`, and a tag like `<hr />` would result in an extraneous visible `>` being added to the document. For backward compatibility with existing documents and browsers, the HTML5 parser ignores self-closing slashes. This means that in HTML, the only self-closing tags that are correctly parsed are those with no closing tag such as `<img>`. While XHTML documents commonly include other self-closing tags such as `<div clear="all" />`, such tags will *not* be treated as self-closing if served with an HTML MIME type. The HTML5 reference parser, which is designed under the constraint of attempting to reconstruct the behavior of non-XML-aware browsers, interprets such tags as (likely unmatched) opening tags. Since we are rewriting the document, we can and do correctly interpret such intended self-closing tags when reading in documents. (When serialized as HTML, they become an open tag immediately followed by a close tag.) This modification can make it easier to normalize the document, as it reduces the number of erroneous unclosed open tags present that could lead to failure of our rewriting algorithms.

Unlike HTML, XHTML also supports XML CDATA sections. These are sections of the document that are treated as unparsed text, allowing formatting characters like `<` to appear without resulting in XML markup. The HTML5 spec supports CDATA sections as well, but only in embedded XML contexts such as MathML. Some HTML parsers may support CDATA sections elsewhere, even in HTML rather than XHTML documents. In HTML contexts and JavaScript blocks, the HTML5 parser sees CDATA sections as "bogus comments" and converts them to standard comments, avoiding this ambiguity. It does not make any changes to such sections in `<style>` blocks; our modified parser flags a fatal error if we see CDATA in this location.

Additional Fixes

In addition to these parser modifications, we perform some additional modifications on the DOM tree to increase the number of documents we can successfully retrofit to a normalized form.

We move any content that was encountered after the close of the HTML `<body>` tag into the body. The HTML5 spec directs browsers to render this content as part of the body, but to preserve its original location in the DOM representation (resulting in a DOM not possible from a valid document), apparently in order to emulate legacy browser behavior. By moving the content into the body, we preserve the rendering of such documents in the majority of browsers while potentially being able to normalize them to be valid.

While we cannot prove that is normalization suffices to ensure consistent parsing across all browsers, we argue that it substantially advances this goal in Section 8.5.1.

While we address a number of cases where malformed documents can lead to a DOM that serializes to a document that still causes parse errors, there are additional cases that we do not normalize. In order to ensure that all documents output by our normalizer can be interpreted by an HTML5 browser without any parse errors, we add a sanity check. Before returning the DOM tree from a parser invocation, we serialize the DOM we have generated and reparse it. We verify that this second parse does not raise any HTML5 parse errors (other than ones related to its DOCTYPE).

8.3.2 Template Normalization

Normalizing a template is more difficult than a static HTML document as its content is dynamically generated, with an unlimited number of different possible outputs. Fortunately, we can take advantage of the limited expressivity of a templating language in order to reason about the form that the possible outputs of a given template can take.

Hole Filling.

The simplest case is that of a "straight-line" template that consists only of a static skeleton and holes (template variables) that are filled in at runtime. We assume that the value of these variables will be escaped or sanitized appropriately at runtime. In particular, we consider the escaping appropriate if it ensures that the remainder of the document will be parsed in the same way as if the hole had filled with an empty string.

This formulation gives us a natural way to extend our HTML parser so it can parse templates containing static HTML and variables. We replace each variable interpolation with a "magic string": a marker that uniquely identifies the original interpolation but does not affect the parsing of the document (i.e., does not change the parse context). This magic string needs to take on different forms depending on where it occurs in the document. In most HTML contexts, white space is ignored and is thus it is safe to replace the hole with a sequence of whitespace characters. In other locations, such as unquoted attribute values, replacing a hole with whitespace could change the parsing of the document.

While we do not have access to the document's HTML parse when magic strings are chosen, it suffices to examine the immediately preceding characters in order to tell which type of magic string to use. If the hole is immediately preceded by a non-whitespace character other than >, we consider it safe to use alphanumerics to represent the magic string.[2] Otherwise, we use whitespace characters, unless the hole is preceded by an = and optional whitespace, in which case we use alphanumerics in order to keep the hole attached to a possible adjacent unquoted attribute value. In both cases, the magic string starts with a pseudorandom sequence in order to avoid collision with any string occuring in a real document.

After replacing holes with magic strings, we pass the resulting template through the normalizing HTML parser described in Section 8.3.1. Thanks to the markers, we are able to identify where each variable interpolation occurs in the parse tree and infer its parse context.

Path exploration.

The problem becomes more complicated once conditional branching constructs such as ifs and loops are added to the template. There is no longer a single unique "skeleton" for the template's output. Instead, there is a potentially-exponential number of straight-line templates to consider.

The conceptually simplest approach would be to explore all possible paths through the template, analyzing each one separately. One could then take the resulting set of normalized straight-line templates and (1) identify the full set of possible contexts for each hole in the original source template and (2) infer a combined template that reinserts branching constructs in appropriate places. Unfortunately, the number of paths can grow exponentially, rendering this approach impractical. Additionally, it may be very complex to reconstruct a single template that merges a large number of paths simultaneously. Therefore, we take a different approach.

We devise a traversal-based divide-and-conquer approach that both reduces the number of paths we must explore and makes merging manageable. In order to eliminate the need to explore every path, we ensure that conditionals are composable, allowing exploration of a smaller set of paths to be predictive of all possible paths. We achieve this by rewriting the template to make all conditionals align with the tree structure of the HTML document. In particular, we ensure that the parse state (including the stack of open elements) after each branch of a conditional is identical to the parse state before the conditional. This requirement helps us avoid exponential blowup in possible parse states.

To achieve this property, we limit conditional branching statements to occur in three

[2] We test for > because only whitespace is allowable in certain locations in tables, where other text is not.

possible locations:

- body text (between tags, i.e., PCDATA context)
- within a tag, in place of a tuple of attribute-value pairs
- within attribute values

We do not assume that the conditionals in the input template strictly match these locations; we instead rewrite each input template to an equivalent one where control statements fall in such locations, when it is possible to do so.

We use a recursive algorithm to perform this rewriting. Accomplishing this requires the use of two separate parsers, one for the template language and one for HTML, and operation on tree structures corresponding to each. The conversion algorithm operates on a template's parse tree, recursing down into the structure of the template. On the way back up the recursion, it builds up a family of conditional-augmented HTML parse trees from the bottom up, at the top level yeilding an augmented HTML parse tree that can be serialized into the rewritten template.

A template parses to a tree containing two general categories of nodes, leaf nodes and interior nodes. Leaf nodes include variable nodes, text nodes, and various other template directives that do not contain further template content. Interior nodes, such as block nodes and conditional nodes, contain one or more subsidiary sequences of other nodes such as a block's contents or the contents of the true and false branches of a conditional node. If these lists are explicitly included as vertices in the template parse tree, the graph consists of alternating levels containing nodes and lists.

Our traversal algorithm can be described as two mutually-recursive routines, INTEGRATE and SUMMARIZE, that occur at these two types of alternating levels in the template's parse tree.

For the sake of simplicity, we will assume in the algorithm descriptions below that the HTML DOM tree is a simple multi-way tree consisting only of generic, untyped nodes. Each node contains some contents *val* and a list of children *ch*. Complications that arise from the fact that an HTML parse tree contains multiple types of nodes are a natural tweak to the general algorithm for simple graphs.

SUMMARIZE is invoked on each interior node in the template. It determines the minimal diff between the HTML parse trees corresponding to the different possible paths through that interior node. The minimal diff is found by recursing on the list of HTML nodes, winnowing down the location of the diff by excluding common prefixes and suffixes. The algorithm proceeds deeper into the HTML parse tree as long as the difference between its inputs can be reduced to a single node that differs only in its children. Once the minimal diff is found, it is replaced by a specially-typed BRANCHNODE that contains the alternate HTML content on the two branches.

INTEGRATE is invoked on the list of template nodes corresponding to one path of an interior node. It first determines a base HTML document to serve as a common basis for comparison by taking the default branch through the sequence of nodes. For each conditional

Algorithm 1 Converting template conditionals to be tree-aligned

```
function ReplDiff($T_1$: TreeNode list, $T_2$: TreeNode list)
    $P, T_1, T_2 \leftarrow$ RemoveCommonPrefix($T_1$, $T_2$)
    $S, T_1, T_2 \leftarrow$ RemoveCommonSuffix($T_1$, $T_2$)
    if Len($T_1$) = Len($T_2$) = 1 and $T_1[0].val = T_2[0].val$ then
        $R \leftarrow$ TreeNode($T_1[0].val$, ReplDiff($T_1[0].ch$, $T_2[0].ch$))
    else
        $R \leftarrow$ BranchNode($T_1$, $T_2$)
    return $P + R + S$

function Summarize($P$: string, $N$: TemplateNode, $S$: string)
    $T \leftarrow$ Integrate($P$, $N.true$, $S$)
    $F \leftarrow$ Integrate($P$, $N.false$, $S$)
    return ReplDiff($T$, $F$)

function TreeMerge($B$: TreeNode list, $\Lambda$: TreeNode list)
    $R \leftarrow [\,]$
    while $\Lambda \neq [\,]$ do
        $P, B, \ell \in \Lambda \leftarrow$ RemoveCommonPrefix($B$, $\ell \in \Lambda$)
        $R \leftarrow R + P$
        $\Lambda' \leftarrow \big[\ell \in \Lambda \,|\, B[0] \neq \ell[0]$ and $B[0].val = \ell[0].val\big]$
        if $\Lambda' \neq \emptyset$ then
            $R \leftarrow R+$ TreeMerge($B[0]$, $\ell[0]$ for $\ell \in \Lambda'$)
            $\Lambda \leftarrow$ [tl $\ell$ for $\ell \in \Lambda - \Lambda'$]
            $B \leftarrow$ tl $B$
        else
            $F \leftarrow$ unique $\ell \in \Lambda \,|$ hd $\ell$ is BranchNode
            $B, B', M \leftarrow$ RemoveCommonSuffix($B$, $F$)
            $R \leftarrow R + M$
            $\Lambda \leftarrow [\ell - B'$ for $\ell \in \Lambda - F]$
    return $R + B$

function Integrate($P$: string, $L$: TemplateNode list, $S$: string)
    $B \leftarrow$ TreeParse($P$ + ChooseDefault($L$) + $S$)
    $\Lambda \leftarrow [\,]$
    $L_P \leftarrow [\,]$
    while $L \neq [\,]$ do
        $\ell \leftarrow$ hd $L$
        $L \leftarrow$ tl $L$
        if $\ell.ch \neq [\,]$ then
            $\Lambda \leftarrow \Lambda$ + Summarize($P$ + Print($L_P$), $N$, Print($L$) + $S$)
        $L_P \leftarrow L_P + \ell$
    return Merge($B$, $\Lambda$)
```

node in the node list, it invokes SUMMARIZE on that node, and collects the results of these invocations.

It then performs a depth-first traversal of the base document HTML tree in order to merge the summarized differences. This consists of repeatedly determining the first location that any document in the set of summary documents differs from the base document. Once this location is determined, the common prefix and the summary placeholder are output, and the portion of the remaining documents corresponding to the placeholder is removed. Once the list of summaries is exhausted, any content from the base HTML tree is appended to yield the integrated document.

At top-level, we invoke INTEGRATE on the template nodelist, and get back the HTML parse tree augmented with variable nodes and branch nodes, representing a normalized version of the template. Serializing this tree yields the rewritten template.

8.3.3 Template Inclusion

In addition to holes and branching constructs, we also support template inclusion. In a template system with inclusion, it is necessary to distinguish between templates that are solely included and those that are used as top-level templates. For our current implementation, we initially assume that any template that contains the case-insensitive string `<html` is a top-level template, and use these as seeds for our traversal. Given a "seed" template, we retrofit it as above, with any template inclusion points treated as conditional branch points. We parse the template both with and without the included content, find the minimal diff between these two trees, and rewrite both the including and included template to reflect this boundary between the two templates.

If the same template is included in multiple locations (either in the same file or across different files), we verify that all such rewritten versions are identical. This is necessary to ensure that the same retrofitted version of the included template can be used in all places that it is included.

8.3.4 Template Extension

In some template languages, it is possible to define a template that "extends" another template. Similar to extending a class in an object-oriented language, the new template inherits the text of the original template, but can override sections of it. This is done through the use of named blocks. The original template defines inline named blocks in specific locations that can then be overridden by child templates. Child templates consist of an `extends` declaration indicating their parent template, followed by named blocks filled with new content. To retrofit a child template, it is first necessary to retrofit the parent template. When we see a template that inherits from another template in our initial enumeration of the files in an application, we assume that its ultimate ancestor must be a top-level template. We accordingly traverse that template's parents up to the top-level ancestor and then retrofit them in reverse order.

When retrofitting a child template, we use its parent template as a base. If the parent template itself extends some other template, we first flatten the sequence of ancestor templates to an equivalent single template. We then treat all blocks that are overridden in the child as if they were conditional branching structures between the base template and the child template. We verify that the diff inferred for such structures is consistent with the already-retrofitted base template, and rewrite the child template's version of the block if necessary. Once all block overrides have been verified in this way, we write out the child template with the rewritten blocks substituting for the original blocks in the template. (Blocks that are defined by the child template but do not exist in the flattened parent template are stripped, as they are dead code.)

8.4 Implementation

We target Django, a Python-based web application framework. Our implementation reads in a set of Django templates, which possibly extend and include each other, and outputs a rewritten version of the templates that satisfy our normalization properties. Any errors that prevent our tool from working are reported, and the affected files are not output.

Our normalizer is based on a modified version of the Python HTML5 reference parser. Our modifications mostly serve to preserve additional information that the reference parser discards because it does not affect the document's rendering in an HTML5-compliant web browser. We preserve this data to help us avoid changes to the document unnecessary for normalization. For instance, we modify the reference parser to preserve the precise whitespace usage and the order of all attributes so that normalization does not introduce trivial changes to the template.

The reference parser assumes that the browser supports frames and has JavaScript enabled and consequently discards the contents of `noframes`, `noscript`, and `iframe` blocks. We modify it to preserve and parse the contents of these blocks. Also, we modify the reference parser to parse the contents of `noembed` blocks, to reflect that their contents may be used by old browsers in place of an adjacent embedded object if it cannot be loaded (the reference parser ignores their contents, which is unsafe).

8.5 Evaluation

In this section, we argue for the soundness of our normalization approach and evaluate the ability of our rewriting tool to normalize real-word templates belonging to open-source Django applications.

8.5.1 Correctness Argument

We believe that our normalization procedure ensures consistent and predictable parsing for untrusted content across all browsers in use. Specifically, we believe that the following two properties hold:

- All browsers will parse the output of the normalized template in the same way, i.e. the HTML parse context of each location in the template where a variable occurs will be consistent across all browsers, including pre-HTML5 browsers.
- The HTML parse context at each point in the output can be determined via simple, straightforward static analysis of the corresponding input templates.

HTML Normalization

In the simplest case, the input template does not contain any holes or template directives. Here our goal is just to ensure that the output is in a sufficiently canonical form that all browsers in common use will parse it the same way. We ensure this by outputting a canonical serialization of a DOM tree rather than directly transforming the input template. This canonical serialization ensures the normalization properties in Figure 8.4. By restricting the document to such a straightforward, unambiguous subset of the browser's input space, it is much less likely that corner cases triggering inconsistent or unexpected behavior will be encountered.

The output of our HTML normalization algorithm is a well-formed (free of parse errors) HTML5 document and thus should be parsed consistently by all modern browsers that conform to the specification. This fact is ensured by our sanity check at the end of the normalization process (See §8.3.1), except for cases in which browsers deviate from the specification. Recent versions of Internet Explorer that otherwise follow the HTML5 standard diverge from it in their support for conditional comments; while we do not interpret and normalize the contents of such comments, we do print a warning if they are detected.

Our modified interpreter will always descend into and parse the contents of `noembed`, `noscript`, `noframes`, and `iframe` blocks. These tags only display their content if some condition is met, such as JavaScript being disabled, or the browser not supporting frames. In the case that the condition fails to hold, the HTML5 specification, reflecting common implementation practice in browsers, has the parser to skip to the end of the block with a string-based search for the ending tag. For consistency, we must ensure that such an approach to parsing the block will end it at the same point that we find when parsing its content like a normal tag. To achieve this, we must ensure that no string is present within the block that the parser will mistake for the end of the block. (Strings like `</noscript`, `</noframes`, etc., could appear, for example, within attribute values or comments without ending the block as parsed.) Our rules prevent such a sequence occurring in any HTML context within the corresponding block. In the HTML body context and within attribute values, `<` is escaped as `&lt;` when it is not part of a opening or closing tag. In `<script>` blocks, the two-character combination `</` is rewritten as `<\/`. In `<style>` blocks, retrofitting fails if the two character combination `</` is present. Retrofitting also fails if any comments are found within potentially unparsed blocks, or if one potentially unparsed block is nested within another.

The output of our normalizer is well-formed SGML in addition to being well-formed HTML5. We also ensure that our output adheres to a subset of SGML that follows the advice in the HTML4 specification for maximizing compatibility. We do not ensure full validity to the HTML DTD; for example, we do not type-check attributes or verify cardinality cardinality constraints on elements. We prevent some elements from appearing in contexts not permitted by the DTD (e.g. `<meta>` and `<title>` in the body of the document rather than the head), but do not enforce all such "content model" constraints. In general, the constraints we fail to enforce are higher-level in nature, and we do not believe that they are likely to affect parsing in such a way as to affect the determination of HTML contexts.

It is rare for a browser to parse a well-formed HTML4 document in a manner inconsistent with the specification, but we perform additional normalization on the document to avoid the cases we are aware of where this can happen. Comments in `script` and `style` blocks are an example where the determination of the end of the block may vary between legacy browsers even if the block's contents are valid according to the HTML4 specification. Sequences of double hypens in comments can also lead to parsing inconsistencies, as do Internet Explorer conditional comments, which have been discussed above.

Straight-line Templates

For straight-line templates, the only added complexity is the presence of holes that add content to the document. As we are modelling the template as a tree, we must ensure that the substitution preserves the tree structure in order for the parsing of the rest of the document to be unaffected. If each input is unambiguously contained within a single tree node, the state of the parser before and after parsing that input will be the same, as there is a direct correspondence between a node in the parse tree and the current state of the parser building up that parse tree. For the normalized template, the context of each hole can be simply determined by using a parser that treats previous holes as containing the empty string.

We support template holes in the following contexts:

- body text (between tags, `PCDATA` context)
- within a tag, in place of a tuple of attribute-value pairs
- within attribute values

For each of these contexts, there exists a set of values that will result in the HTML parse state being equivalent before and after the template hole. These values correspond to a properly constructed subtree rooted at the location of the hole. We assume that the inputs to the template each fall within this set for the appropriate context. This assumption can be met by applying an appropriate automatic escaping function for each context.

For the PCDATA context, HTML entity escaping will suffice to guarantee this invariant. The parser will remain within the PCDATA context throughout, and all content will be inserted into the the body of the same tag. The only difference in parser state will be the textual content, which does not affect parsing. HTML entity escaping also suffices for attribute values, and will ensure that the content remains within a single attribute value. (Using this escaping function here relies on the invariant that all attribute values in our normalized document are quoted.)

For the attribute context, we must ensure that the input will parse as a list of well-formed attributes. An input of this type can be verified by a sanitization function that parses and outputs such values, or by checking that the input is marked as the output of a trusted function for generating values of this type. Any unmarked inputs can be safely replaced by the empty string. The entirety of the input will be contained in the tag where the variable node appears, so the parse state will not change. The only difference is the set of attributes defined, which does not affect the parser state machine.

For cross-site scripting protection, additional restrictions are required on the contents of some attribute values to ensure that they do not lead to JavaScript injection. These include event handler attributes, which are interepreted directly as JavaScript, and URLs, which can include code if schemas like `javascript:` are used. In an automatic escaping system, it is necessary to detect these cases to provide proper defense against cross-site scripting, and the same is true of static analyses. Our normalization makes it easy to identify the attribute name associated with each attribute value, as we do not allow variable interpolations to exist in locations where they could complicate this determination. Specifically, it will always be the most recent attribute name seen while parsing the document.

Branching Structures

For conditional branching structures and their equivalents, we also wish to show that the parser state is identical before and after each branch in the normalized document. Again, the fact that each branching structure corresponds to a subtree of the document will help us argue that the parser state is the same regardless of which path is chosen for each branch.

We support conditionals in the same locations as template holes. For conditionals, the content of each branch corresponds to a properly formed subtree of the HTML document, and so the state of the parser differs between branches only by the characters in the current state.

We can prove this property using strong induction on the number of conditionals present in template fragments found in a branch of a conditional. The induction hypothesis is that the contents of every template branch with fewer than k conditionals (summed over all branches) satisfy the property that they start and end at the same parse context.

For a branch with zero conditionals, we know that the starting context is one of the three possible contexts where a conditional can be located. As the contents of the template branch are a valid subtree for that location in the HTML document, the parse context before and after the branch are the same. For conditionals located within HTML opening tags (between or within attribute values), the same reasoning holds as for straight-line templates. For conditionals located in the PCDATA context, even though tags can be present, all tags opened are properly closed due to the output being the serialization of a DOM tree. The ending context will be the same as the start, still in the PCDATA context with the same set

Application			Manual Mod.		Diff Size		Excluding `/>`	
Name	Templ.	Lines	Templ.	Lines	Templ.	Lines	Templ.	Lines
django-admin	24	812	0	0	13	62	2	31
damned-lies	61	2649	6	46	43	233	24	123
ccdb	54	2498	3	4	32	175	16	90
fabiosuoto	11	513	0	0	9	43	5	12
niftyurls	7	153	0	0	4	16	1	1
victims	14	378	2	2	7	15	4	4
yumeblog	4	231	1	2	2	16	2	6

Table 8.1: Experimental results for template normalization. The left section shows the Django applications we evaluate on. The middle section shows that most templates are automatically handled by our tool; a small subset require manual intervention. The right section indicates that normalization makes only relatively modest changes to the templates. See Section 8.5.2 for detailed explanation of the columns.

of tags opened. The text and elements already seen within the current element do not affect the behavior of the parser when parsing subsequent elements within that element.

For a branch containing k conditionals, for each conditional at top level, the induction hypothesis ensures that the branch chosen for that conditional does not affect the parse, and so nested conditionals can be skipped without changing the parse. Once the nested conditionals are removed, the same reasoning as used for the base case can be applied to the branch.

8.5.2 Experiments

We ran our tool on the source code of several open-source Django applications listed on `django-sites.org`, as well as templates belonging to the standard administrative interface distributed with the Django distribution. These applications have between 4 and 79 HTML templates each, amounting to between 153 and 2,649 source lines of template code. Our results are summarized in Table 8.1.

The first section of Table 8.1 shows the name of the application, followed by the number of HTML templates in the application and their total lines of code. We only attempted to retrofit templates that generated a top-level HTML document, or that extended or were included by a template that did so. In doing so, we skipped a number of non-HTML templates such as email message contents and RSS feeds, as well as templates used for inline HTML such as those used to define custom template tags.

Django-admin is a collection of library code provided by the Django distribution that provides a web-based interface for viewing and modifying application data. It also includes prebuilt code and templates for dealing with user authentication and registration. Templates of other applications can make use of this module by referencing or overriding its templates; two of the applications we evaluated, damned-lies and ccdb, referenced and overrode templates from django-admin. When evaluating these applications, we included the templates from django-admin along with that provided by the application. The numbers in the table include the django-admin template code for these applications.

The Manual Modifications section of Table 8.1 shows where manual intervention was required to successfully normalize the templates and quantifies the amount of intervention needed. The first column lists the number of template files that we had to edit or add in order for our retrofitting tool to run without fatal errors, and the second column shows the number of lines we had to change in these files. Several of the applications did not require any changes.

The Diff Size columns of Table 8.1 quantify the degree of modification our retrofitting process makes to the original templates. Generally, the number of changes introduced by our tool is relatively modest. As discussed below, these could be further reduced by reasonable relaxations to our normalization properties and improvements to our implementation. The last pair of columns shows the size of the diff given one possible relaxation of our normalization policy, preserving XHTML-style self-closing tag indicators. This would reduce the number of changes by slightly over one-half.

8.5.3 Manual intervention details

We summarize here the changes we had to make manually. The majority of changes were simple, such as inserting missing closing tags in locations that our tool could not handle automatically.

Damned Lies is a web application for managing localization of the open-source Gnome project. It is the largest application we examined as well as the one that required the largest amount of manual changes in order for its templates to be successfully rewritten by our normalizer.

One template used the `ifchanged` template directive (in two locations) to close and reopen a `<ul>` tag to visually group related elements. (See Figure 8.5(a).) We were able to avoid this by changing the template to use the built-in `regroup` directive, which is provided for exactly this purpose and does not introduce unaligned conditionals. In two templates, code for constructing tables was problematic. For one, it was fairly straightforward to relocate opening and closing `<td>` tags so that all conditionals were properly aligned. The second was more involved, and involved a table similar to the one in Figure 8.5(b). Rewriting this template to make conditionals and tags nest properly is trickier; we ended up moving some of the code into a separate, included template (see Figure 8.5(c)).

The remainder of the manual changes required for the programs we evaluated were a result of buggy HTML that our tool was unable to fix automatically. While our tool is able to correct many forms of invalid HTML, in some cases our tool was not able to work out the developer's intent. Fixing these cases required us to manually make minor changes to the templates. In CCDB, a misplaced `{%endif%}` and two erroneously self-closing `div`s caused normalization failures. For victims, the ten failed templates were caused by a `<div>` being erroneously closed with a `</p>` and then nested in another `<p>` tag by its including template.

```
{% ifchanged rel.status %}
   </ul>
   <ul class="foot">
{% endifchanged %}
```

(a)

```
<table><tr><td colspan="2">
<p>Instructions</p>
{%if option_a_enabled %}
  <p>Choice of A or B:</p>
  </td></tr><tr><td>
  Option A
  </td><td>
{% endif %}
Option B
</td></tr></table>
```

(b)

```
<table><tr><td colspan="2">
<p>Instructions</p>
{% if option_a_enabled %}
   <p>Choice of A or B:</p>
{% else %}
   {% include "optionb.html" %}
{% endif %}
</td></tr>
{% if option_a_enabled %}
   <tr>
   <td>Option A</td>
   <td>{% include "optionb.html" %}</td>
   </tr>
{% endif %}
</table>
```

(c)

Figure 8.5: (a) and (b) show problematic constructs that our normalizer could not automatically rewrite, and which required manual modification. (c) shows how we re-wrote (b).

In YumeBlog, our tool broke one template due to its use of a custom looping template tag in combination with an unclosed `<a name>` tag. (Our normalizer cannot guarantee soundness in the case of custom template tags, as we do not understand their semantics.)

8.5.4 Changes made to the templates

Many of the changes our tool makes could be reduced if our tool was more optimized to reduce the size of the modifications made to the templates. Our tool removes the XML-style "/" from self-closing HTML tags, as it is not technically valid HTML. However, it is in widespread use in the web, even for pages served as HTML, and thus all browsers must be able to handle it. If these changes are excluded (as in the last columns of the table), the number of changes is reduced significantly. Other common changes include the closing of unclosed tags. In some cases, conditional constructs were relocated so as to minimize their extent, e.g. from around a tag to within a tag value. This shortens the document, but can result in long lines, which may be stylistically less desirable. A more sophisticated rewriter may be able to preserve the location of such conditionals in order to avoid this modification. (Conditionals can often be placed in a variety of locations while preserving our well-formedness property for templates.)

These changes incur a one-time cost for developers. Overall, the evaluation suggests that our approach and tool do not require undue developer effort and can be applied to real applications.

8.6 Conclusion

We provide an automated tool for normalizing templates such that contexts can be reliably determined. Most templates can be retrofitted automatically, and a small number require additional manual effort. We thus show that it is practical to achieve stronger, more principled assurance for context-sensitive autoescaping and other applications of static analysis to web templates.

Chapter 9

Related Work: Web Templating

9.1 Web Templates

The earliest dynamic web content consisted of server-side scripts generating HTML output, often in response to forms submitted from static pages. The dynamic pages were generally implemented using scripting languages such as Perl, and HTML output was generated with print statements. Template languages were developed as a way to design dynamic web pages in a more document-centric way, with invariant portions of the output document alternating with code to generate the dynamic portions. Template languages, such as JSP, ASP, and PHP, have traditionally been string-based; literal strings in the template are simply concatentated with strings generated programmatically.

Several XML-based template languages have introduced a more structured approach to document assembly. XSLT is an XML-based language used for transforming XML input documents into XML output documents. As it is designed to operate only on XML inputs, it is not directly suited to most web templating applications.

TAL [80] and Kid [70] use validated XML documents as templates. String inputs can be inserted as attribute values, or inside or in place of XML tags. For attribute values, only strings can be output, which Kid automatically escapes appropriately. While this preserves the integrity of the template's parse tree, it does not prevent inclusion of JavaScript or other dangerous content. The programmer must explicitly impose additional restrictions on output beyond those required for enforcing the well-formedness of the output and structural integrity of the template.

Genshi [17] is very much like Kid, but adds some support for policy enforcement on HTML, in the form of a sanitizing filter. This filter checks a chunk of XML (likely generated by parsing a user string) for conformance to a policy. Because these frameworks use validated XML as their template language, there should be no ambiguity over how their templates will be parsed. They offer the ability to output the interpreted template as XHTML or HTML and guarantee well-formedness. Unfortunately, they are harder to use for the programmer. Certain constructs must be implemented in a less straightforward way in order to ensure the template is valid XML.

Unlike XML-based templating frameworks, the approaches based on automatic escaping of string-based templates, such as the ones explored in this dissertation, are more easily applied to existing template code. A context-sensitive automatic escaping system operating on sufficiently normalized templates retains the improved parsing consistency guarantees of structure-aware templating while allowing the more familiar and convenient syntax of existing template languages.

9.2 Autoescaping

Context-insensitive automatic escaping is supported by many modern templating systems; it was introduced in ClearSilver in 2006 [37] (allowing a choice of which automatic filter to use) and Django in 2007 [16]. Context-sensitive templating was first introduced by Google CTemplate in 2009 [66], which runs a parser on the template's output as it is being interpreted in order to determine the current output context.

Samuel et al. [61] provide a type system that can be used to statically determine the contexts of interpolation points in web templates via type inference. This allows for context-sensitive autoescaping without the performance cost of determining the HTML context at runtime that is incurred by dynamic context-sensitive autoescaping systems. They integrate their work into the template compiler for Google Closure Templates [60], and evaluate the correctness and coverage of their type inference using production templates for various Google web services. Like any context-sensitive escaping system, this avoids the problems of incorrect or missing sanitization of untrusted input.

Chapter 7 addresses the orthogonal concern of reducing the number of template inputs that need to be explicitly marked as trusted. While Samuel et al. identified the existence of many templates that could be migrated to their system and were correctly handled without the use of opt-outs from escaping, they did not evaluate the effort required to migrate full applications to use contextual autoescaping or the number of opt-outs required. Additionally, in our support for programmatic template rewriting and database trust annotations, we extend the safety properties provided by context-sensitive escaping beyond the template system to encompass the entire web application.

All existing context-sensitive autoescaping systems, whether dynamic or static, make use of a single parser to determine document contexts, and do not normalize document content. While this works most of the time, proper normalization, as performed in Chapter 8, provides a sounder basis for defenses based on verifying or automatically placing sanitizers and escaping functions.

9.3 HTML Parsing Correctness

In order to address the problem of client-side parsing differing from that used on the server side, a number of papers have proposed adding a mechanism for explicitly annotating untrusted content at the server. This annotation would allow a properly instrumented client to provide defense in depth, as it could render inoperative any injections present in content marked as untrusted. The first system to take this approach was BEEP [28], which provides a way for documents to specify which scripts are authorized to execute on a page. The policies are written in JavaScript, allowing for a great deal of versatility; as examples, the paper includes a whitelist-based policy and a policy wherein a custom HTML attribute identifies regions where scripts should be disabled. Document Structure Integrity [47] and Noncespaces [23] provide alternate mechanisms for marking trusted and untrusted content that aim to be more robust in the face of adaptive attacks and inconsistent parsing. Unlike these systems, a normalization approach does not require any changes on the client side.

Blueprint [38] avoids possible ambiguities in parsing of untrusted content by parsing it into a DOM tree on the server and transmitting this tree using a custom serialization mechanism. It avoids any dependency on the client's parser, instead rebuilding the DOM structure on the client side by use of custom JavaScript. While this avoids the potential difficulties of performing correct normalization, it incurs substantial client-side overhead and makes content inaccessible to clients such as web crawlers that do not execute JavaScript.

9.4 Other Cross-Site Scripting Defenses

A number of other approaches have been proposed for defending against cross-site scripting and other content-injection attacks in web applications. Nguyen-Tuong et al. [48] use dynamic, character-level taint-tracking to prevent PHP variable injection, SQL injection and cross-site scripting. For XSS prevention, they check that sensitive HTML output (such as `<script>` tags) does not contain tainted characters. Violations of this property can are made safe by escaping appropriate for the context where tainted characters are found. Like runtime context-sensitive autoescaping, this involves keeping track of the parse state of the output.

GuardRails [12] provides a more sophisticated taint-tracking system, which allows transformation and access-control policies to be associated with tainted strings. Transformation policies specify the escaping functions to be used for a string when it is output in different parse contexts. This rich policy language is more complex but offers greater flexibility than the trusted-for-a-particular-context designation used in autoescape systems such as our work. GuardRails integrates with Rails' object-relational mapper and uses annotations on database fields for access control policy specification, but not for transformation policies. Unlike our work, which annotates trusted database fields, they persist character-level taint information in the database to guard against stored content injection without over-escaping. In our evaluation, we show that at least for our template-based applications, we can maintain functionality while providing more reliable security guarantees and avoiding the overhead of dynamic taint tracking.

Pixy [29] is a static analysis tool for PHP, designed to detect cross-site scripting and SQL injection vulnerabilities. It is based on static taint analysis, and warns if it detects a flow of data from an untrusted source, such as a query parameter, to a trusted sink, such as an `echo` statement. Invocation of sanitizer functions clears the taint status. This analysis aims to detect missing sanitization, a bug that is no longer possible with autoescaping template systems.

Wasserman and Su [74] and the Saner tool [3] both provide more sophisticated static taint analyses that are also able to reason about the correctness of sanitizer functions. Functions that incompletely sanitize their input and thus still admit script injection can potentially be detected. This work, and other similar work on modeling sanitizers [25], is complementary to normalization and security review of data-flow paths that disable sanitization by the use of opt-outs, but may be helpful in establishing trust in sanitizers and escape functions.

XSS-Guard [7] aims to defend against XSS attacks by instrumenting the web application to perform all string operations on a crafted-benign "shadow" input in parallel to those performed on the actual input. The two documents are then parsed using code extracted from a real web browser and compared; content from the shadow run instead of the real run is used in case of a parse-tree mismatch. This approach is essentially equivalent to dynamic taint-tracking, but used a higher-fidelity browser than previous taint-tracking systems.

ScriptGard [64] is a system designed to detect and fix misplaced sanitizers in large .NET web applications. Its detection of mismatched sanitizers derives from a path-based analysis using an instrumented version of the application and a custom web browser. For a given seed input, the instrumented application tracks "positive" taint information about string literals as well as the set of sanitizers applied to strings. Based on path exploration with a test suite and an instrumented browser, it finds the locations in each control-flow path that are incorrectly sanitized. In addition to use for bug-finding, ScriptGard also has a mode where it uses this information to correct known mismatched-sanitization bugs at runtime, with optimizations to achieve low overhead. Like ScriptGard, we use the notion of positive tainting in our runtime trust annotations in Chapter 7. In contrast to ScriptGard, we aim to support verifiable correctness for legacy code rather than detection of bugs. In addition to defending against buggy legacy code like ScriptGard in our mitigation mode, we also aim to facilitate retrofitting existing code to be verifiably safe from XSS in strict mode.

Chapter 10

Conclusion

This dissertation has demonstrated language subsetting as a practical way to bring the security benefits of object-capabilities to an existing language. Joe-E has enabled enforcing and verifying a number of properties that are helpful to security review and reasoning about programs, including privilege separation, secure encapsulation, immutability, and determinism. It has aided in the building of a number of interesting applications and systems whose security and correctness properties are more easily reviewed due to the features provided by Joe-E. I hope that these contributions will be helpful as the software industry places a greater emphasis on building reviewably secure software.

This dissertation also contributes to achieving reliable security for web applications. I believe that building and facilitating the use of the correct abstractions is essential to improving the security of this category of software. I hope that this work will inform and inspire future development of framework-based techniques for the creation of correct and reviewable web applications.

Bibliography

[1] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *OOPSLA '06: 21st ACM Conference on Object-Oriented Programming Systems and Applications*, pages 57–74, Portland, Oregon, USA, 2006.

[2] M. Backes, M. Dürmuth, and D. Unruh. Information flow in the peer-reviewing process (extended abstract). In *IEEE Symposium on Security and Privacy, Proceedings of SSP'07*, pages 187–191, May 2007.

[3] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, May 2008.

[4] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[5] M. Barnett, K. R. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, 2004.

[6] D. J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *CSAW '07: Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 1–10, New York, NY, USA, 2007. ACM.

[7] P. Bisht and V. Venkatakrishnan. Xss-guard: Precise dynamic prevention of cross-site scripting attacks. In *5th Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, July 2008.

[8] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: splitting applications into reduced-privilege compartments. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[9] W. Bright. D language 2.0. `http://www.digitalmars.com/d/2.0/`.

[10] L. Brown. AEScalc. `http://www.unsw.adfa.edu.au/~lpb/src/AEScalc/AEScalc.jar`.

[11] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. R. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[12] J. Burket, P. Mutchler, M. Weaver, M. Zaveri, and D. Evans. Guardrails: A data-centric web application security framework. In *2nd USENIX Conference on Web Application*

Development (WebApps 2011), June 2011.

[13] Y. Cheon and G. Leavens. A runtime assertion checker for the Java Modeling Language, 2002.

[14] T. Close and S. Butler. Waterken server. `http://waterken.sourceforge.net/`.

[15] D. Crockford. ADsafe. `http://www.adsafe.org`.

[16] Django Software Foundation. AutoEscaping – Django. `https://code.djangoproject.com/wiki/AutoEscaping`.

[17] Edgewall Software. Genshi. `http://genshi.edgewall.org`.

[18] P. Efstathopoulos, M. Krohn, S. Vandebogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *In Proc. 20th ACM Symp. on Operating System Principles (SOSP*, pages 17–30, 2005.

[19] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA '03: 18th ACM Conference on Object-Oriented Programming Systems and Applications*, pages 302–312, Anaheim, California, USA, 2003.

[20] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA '07: 22nd ACM Conference on Object-Oriented Programming Systems and Applications*, pages 337–350, Montréal, Québec, Canada, 2007.

[21] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in Java. In *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 161–174, New York, NY, USA, 2008. ACM.

[22] L. Gong, M. Mueller, and H. Prafullch. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, 1997.

[23] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *16th Annual Network & Distributed System Security Symposium*, February 2009.

[24] N. Hardy. KeyKOS architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, 1985.

[25] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *Proceedings of the Usenix Security Symposium*, August 2011.

[26] HTML4 Test Suite. `http://www.w3.org/MarkUp/Test/HTML401/current/`.

[27] R. Ierusalimschy and N. de La Rocque Rodriguez. Side-effect free functions in object-oriented languages. *Comput. Lang.*, 21(3/4):129–146, 1995.

[28] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *16th International World Wide Web Conference*, May 2007.

[29] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, May 2006.

[30] M. F. Kaashoek, D. R. Engler, G. R. Ganger, n. Hector M. Brice R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and

flexibility on exokernel systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 52–65, New York, NY, USA, 1997. ACM.

[31] D. Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284, 2003.

[32] A. Krishnamurthy, A. Mettler, and D. Wagner. Fine-grained privilege separation for web applications. In *WWW '10: Proceedings of the 19th International Conference on the World Wide Web*, pages 551–560, 2010.

[33] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, J. G. Mitchell, G. J. Popek, and G. J. Popek. Report on the programming language Euclid. *SIGPLAN Not.*, 12(2):1–79, 1977.

[34] G. Leavens and Y. Cheon. Design by contract with JML, 2003.

[35] H. M. Levy. *Capability-based computer systems.* Digital Press, Maynard, MA, USA, 1984.

[36] T. Lindholm and F. Yellin. *Java(TM) Virtual Machine Specification, The (2nd Edition).* Prentice Hall PTR, April 1999.

[37] B. Long. Clearsilver. `http://www.clearsilver.net/`.

[38] M. T. Louw and V. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *30th IEEE Symposium on Security and Privacy*, May 2009.

[39] A. Mettler and D. Wagner. The Joe-E language specification, version 1.1, September 18, 2009. `http://www.cs.berkeley.edu/~daw/joe-e/spec-20090918.pdf`.

[40] A. Mettler and D. Wagner. Class properties for security review in an object-capability subset of java (short paper). In *ACM SIGPLAN 5th Workshop on Programming Languages and Analysis for Security*, June 2010.

[41] A. Mettler, D. Wagner, and T. Close. Joe-e: A security-oriented subset of java. In *17th Annual Network and Distributed System Security Symposium (NDSS 2010)*, March 2010.

[42] B. Meyer. *Eiffel: The Language.* Object-Oriented Series. Prentice Hall, Englewood Cliffs, NJ, USA, 1992.

[43] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[44] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript (draft), 2008. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`.

[45] J. H. Morris, Jr. Protection in programming languages. *Commun. ACM*, 16(1):15–21, 1973.

[46] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.

[47] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *16th Annual Network & Distributed System Security Symposium*, February 2009.

[48] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically

hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, June 2005.

[49] M. Odersky. The Scala programming language. `http://www.scala-lang.org`.

[50] D. Oswald, S. Raha, I. Macfarlane, and D. Walters. HTMLParser 1.6. `http://htmlparser.sourceforge.net/`.

[51] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for java. In *ISSTA '08: 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, 2008.

[52] N. Provos. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–242, 2003.

[53] X. Qi and A. C. Myers. Masked types for sound object initialization. In *OOPSLA '09: 24th ACM Conference on Object-oriented Programming Systems and Applications*, pages 53–65, Savannah, Georgia, USA, 2009.

[54] D. Raggett, A. L. Hors, and I. Jacobs. Html 4.01 specification, December 1999. `http://www.w3.org/TR/html401/`.

[55] J. A. Rees. A security kernel based on the lambda-calculus. *A. I. Memo 1564, MIT*, 1564, 1996.

[56] A. Rountev. Precise identification of side-effect-free methods in java. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 82–91, Washington, DC, USA, 2004. IEEE Computer Society.

[57] A. Rudys and D. S. Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security*, 5(2), May 2002.

[58] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, pages 199–215, 2005.

[59] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Communications of the ACM*, 1974.

[60] M. Samuel and K. Huang. Closure templates. `http://closure-templates.googlecode.com/`.

[61] M. Samuel, P. Saxena, and D. Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *ACM Conference on Computer and Communications Security*, October 2011.

[62] N. K. Sastry. *Verifying Security Properties in Electronic Voting Machines*. PhD thesis, University of California at Berkeley, 2007.

[63] F. Sauer. Eclipse metrics plugin 1.3.6. `http://metrics.sourceforge.net/`.

[64] P. Saxena, D. Molnar, and B. Livshits. Scriptgard: Automatic context-sensitive sanitization for large-scale legacy applications. In *ACM Conference on Computer and Communications Security*, October 2011.

[65] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, New York, NY, USA, 1999. ACM.

[66] C. Silverstein. Ctemplate. `http://ctemplate.googlecode.com/`.

[67] F. Spiessens and P. V. Roy. The Oz-E project: Design guidelines for a secure mul-

tiparadigm programming language. In *In Multiparadigm Programming in Mozart/Oz: Extended Proceedings of the Second International Conference MOZ 2004, volume 3389 of Lecture Notes in Computer Science*, pages 21–40. Springer-Verlag, 2005.

[68] M. Steigler and M. Miller. How Emily Tamed the Caml. Technical Report HPL-2006-116, HP Laboratories, August 11, 2006.

[69] B. Stroustrup. A rationale for semantically enhanced library languages. In *Proceedings of the First International Workshop on Library-Centric Software Design (LCSD 05)*, pages 44–52, 2005.

[70] R. Tomayko. In search of a Pythonic, XML-based templating language. `http://tomayko.com/writings/pythonic-xml-based-templating-language`.

[71] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.

[72] P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, 1992.

[73] D. Wagner and D. Tribble. A security analysis of the Combex DarpaBrowser architecture, March 4, 2002. `http://www.combex.com/papers/darpa-review/security-review.pdf`.

[74] G. Wasserman and Z. Su. Static detection of cross-site scripting vulnerabilities. In *International Conference on Software Engineering*, May 2008.

[75] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. A systematic analysis of xss sanitization in web application frameworks. In *European Symposium on Research in Computer Security*, October 2011.

[76] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.

[77] K.-P. Yee and M. Miller. Auditors: An extensible, dynamic code verification mechanism, 2003. `http://www.erights.org/elang/kernel/auditors/index.html`.

[78] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[79] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. D. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE 2007: 11th European Software Engineering Conference and 15th ACM Symposium on the Foundations of Software Engineering*, pages 75–84, Dubrovnik, Croatia, September 5–7, 2007.

[80] Zope Foundation. TAL specification 1.4. `http://wiki.zope.org/ZPT/TALSpecification14`.