Joe-E: A Security-Oriented Subset of Java

Adrian Mettler David Wagner Tyler Close
University of California, Berkeley waterken.org
{amettler, daw}@cs.berkeley.edu tyler.close@gmail.com

Abstract

We present Joe-E, a language designed to support the development of secure software systems. Joe-E is a subset of Java that makes it easier to architect and implement programs with strong security properties that can be checked during a security review. It enables programmers to apply the principle of least privilege to their programs; implement application-specific reference monitors that cannot be bypassed; introduce and use domain-specific security abstractions; safely execute and interact with untrusted code; and build secure, extensible systems. Joe-E demonstrates how it is possible to achieve the strong security properties of an object-capability language while retaining the features and feel of a mainstream object-oriented language. Additionally, we present ways in which Java's static type safety complements object-capability analysis and permits additional security properties to be verified statically, compared with previous object-capability languages which rely on runtime checks. In this paper, we describe the design and implementation of Joe-E and its advantages for security and auditability over standard Java. We demonstrate how Joe-E can be used to develop systems with novel security properties that would be difficult or impossible to ensure otherwise, including a web application platform that provides transparent, transactional object persistence and can safely host multiple mutually-distrustful applications in a single JVM.

1 Introduction

This paper describes the design and implementation of a programming language, called Joe-E, which 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. 1, 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

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

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

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

Figure 1. An append-only logging facility.

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.

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.

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.

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

```
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. An untrusted image decoder might implement this interface.

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. 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. 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

```
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 3. A secure abstraction that supports flexible use of currencies.

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. 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 encour-

ages 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.

This paper. In the rest of this paper, we describe the goals we had for the Joe-E language (§ 2), the design of the language to meet these goals (§ 3–4), patterns that can be used in reasoning about security of Joe-E applications (§ 5), and our implementation of the Joe-E language and runtime (§ 6). Finally, we evaluate the degree to which Joe-E has met our goals by discussing our experience with the Waterken server and how it makes use of Joe-E to host mutually-distrustful web applications (§ 7).

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.

2.1 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 (§ 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 (§ 3.1) and exposing a capability-secure subset of the Java class libraries (§ 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 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). 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 (§ 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 (§ 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). 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 counterparty 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 (§ 5.3).

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 (§ 5).

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 (§ 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 (§ 5.4), by providing library support for programming with immutable data, and by forbidding mutable global variables (§ 4.2, § 5.2).

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.

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. 4 and § 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.

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 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.

[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 4. Overview of restrictions that Joe-E imposes to enforce capability security.

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 object invariants that could be crucial to security, breaking encapsulation. See Fig. 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.

```
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 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.

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.

4.2.1 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 [22]. 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.)

4.2.2 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*,

[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.

`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 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.

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. 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

[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 6. There is a security risk, if exceptions can contain capabilities.

necessary for two reasons. First, the behavior of the JVM after a `VirtualMachineError` is technically undefined [9, §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 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

```
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 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.

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. 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. 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 [10, §4.8] contains further discussion and analysis of these workarounds.

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.

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 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.

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 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 can-

[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.

not 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.

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 [11, §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.

5.4 Immutability

Joe-E's support for immutable types (§ 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. 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. 8 shows a classic Java vulnerability and how Joe-E's immutable types eliminate the vulnerability pattern.

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

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

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

Figure 8. Above, 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. Below, a natural way to write this code in Joe-E is secure without defensive copies, thanks to the use of immutable types.

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.

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. 9 shows a typical Java queue interface, followed by a Joe-E queue interface that predefines the at-

```
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 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.

tenuated 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.

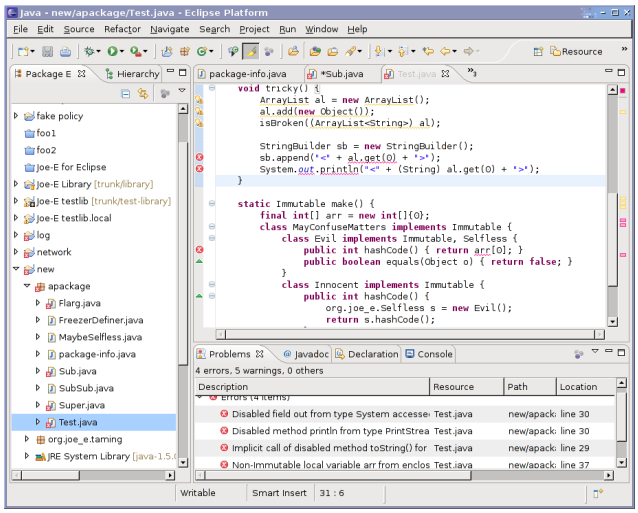

Figure 10. The Joe-E Verifier for Eclipse

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 compiler to ensure that tamed-away `toString()` methods will not be invoked [10, § 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 (§ 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.

7 Evaluation

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.

We have used Joe-E to build two interesting applications. In previous work, we ported an open-source HTML parsing library, originally comprising over 10K lines of Java code, to Joe-E, providing deterministic and side-effect free HTML parsing [4]. Here, we describe the Waterken server, which provides secure isolation and cooperation between mutually distrustful web applications written in Joe-E.

7.1 Waterken

In its standard distribution, Joe-E supports design and review of code that is single-threaded, transient and local. The Waterken software extends this scope to code that is multi-threaded, persistent and networked. Waterken follows the asynchronously communicating event loop model [11]. 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 re-

main 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).

7.1.1 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. 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.

7.1.2 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 [4] 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.

8 Related Work

Capabilities have a long history as an approach for securing systems [8]. Early multi-user 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 [6], EROS [18], and derivatives. We share the view of capabilities as programmatically invokable references, but integrate them into the language.

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 [13]. W7 implemented these features in a Scheme environment and provided an early example of language support for capabilities [17]. Joe-E was heavily influenced and inspired by E, a seminal object-capability language [11]; 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 [19], an object-capability variant of Oz, and Emily [20], 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 [12] 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 [2] 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 [21]: they subset a base language, then augment its functionality by adding libraries.

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 [5]. 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 [15] 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 [3] and HiStar [23] operating systems enforce information-flow policies at a per-process granularity. Jif [14] 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. Like Joe-E, Jif is based upon Java, leveraging programmer familiarity with Java. 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.

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 [16]. The privman library [7] factors out much of the logic of implementing a privilege-separated program. The Wedge toolkit [1] 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.

In previous work we examined how to verify that methods are functionally pure (deterministic and side-effect-free), and argued that purity has useful applications in computer security [4]. That work described how, given an object-capability language, one can extend it to obtain a *deterministic* object-capability language; it described how deterministic object-capability languages support verification of functional purity; and it discussed the application of those techniques to Joe-E. In contrast, this paper is concerned with how to design a practically useful object-capability language in the first place, rather than how to support determinism and verifiable purity.

9 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.

Acknowledgements

Mark Miller and Marc Stiegler made major contributions to the early design of Joe-E, and we gratefully acknowledge their sage guidance and advice. We thank Devdatta Akhawe, Arel Cordero, Matthew Finifter, and the anonymous reviewers for their helpful comments on earlier versions of this paper.

This material is based upon work supported by the National Science Foundation under grants CNS-0716715, CCF-0424422, and CCF-0430585. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: splitting applications into reduced-privilege compartments. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*, pages 309–322, 2008.

[2] D. Crockford. ADsafe. `http://www.adsafe.org`.

[3] 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 *20th ACM Symposium on Operating System Principles (SOSP'05)*, pages 17–30, 2005.

[4] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in Java. In *15th ACM Conference on Computer and Communications Security (CCS'08)*, pages 161–174, 2008.

[5] 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.

[6] N. Hardy. KeyKOS architecture. *SIGOPS Operating Systems Review*, 19(4):8–25, 1985.

[7] D. Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284, 2003.

[8] H. M. Levy. *Capability-based computer systems*. Digital Press, Maynard, MA, USA, 1984.

[9] T. Lindholm and F. Yellin. *Java(TM) Virtual Machine Specification, The (2nd Edition)*. Prentice Hall PTR, April 1999.

[10] 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`.

[11] 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.

[12] 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`.

[13] J. H. Morris, Jr. Protection in programming languages. *Commun. ACM*, 16(1):15–21, 1973.

[14] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.

[15] M. Odersky. The Scala programming language. `http://www.scala-lang.org`.

[16] N. Provos. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–242, 2003.

[17] J. A. Rees. A security kernel based on the lambda-calculus. *MIT A.I. Memo 1564*, 1996.

[18] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 170–185, 1999.

[19] F. Spiessens and P. V. Roy. The Oz-E project: Design guidelines for a secure multiparadigm programming language. In *Multiparadigm Programming in Mozart/Oz: Extended Proceedings of the Second International Conference (MOZ 2004)*, pages 21–40. Springer-Verlag, 2005.

[20] M. Steigler and M. Miller. How Emily Tamed the Caml. Technical Report HPL-2006-116, HP Laboratories, August 11, 2006.

[21] 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.

[22] 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`.

[23] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006.