

Verifiable functional purity in Java

Matthew Finifter Adrian Mettler Naveen Sastry David Wagner
University of California, Berkeley
{finifter, amettler}@cs.berkeley.edu, naveen@ksastry.com, daw@cs.berkeley.edu

ABSTRACT

Proving that particular methods within a code base are *functionally pure*—deterministic and side-effect free—would aid verification of security properties including function invertibility, reproducibility of computation, and safety of untrusted code execution. Until now it has not been possible to automatically prove a method is functionally pure within a high-level imperative language in wide use, such as Java. We discuss a technique to prove that methods are functionally pure by writing programs in a subset of Java called Joe-E; a static verifier ensures that programs fall within the subset. In Joe-E, pure methods can be trivially recognized from their method signature. To demonstrate the practicality of our approach, we refactor an AES library, an experimental voting machine implementation, and an HTML parser to use our techniques. We prove that their top-level methods are verifiably pure and show how this provides high-level security guarantees about these routines. Our approach to verifiable purity is an attractive way to permit functional-style reasoning about security properties while leveraging the familiarity, convenience, and legacy code of imperative languages.

Categories and Subject Descriptors

D.2.3 [Coding Tools and Techniques]; D.2.4 [Software/Program Verification]

General Terms

Security, Languages

Keywords

Pure computation, determinism, verification, formal methods, static analysis, object-capabilities

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. 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 existing 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 properties¹:

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 same 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.

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

¹More formal definitions of these two properties in the context of our system are provided in Section 4.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2008 ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

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 their correctness is verified as needed by this check.

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 [16]. 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. No additional restrictions are placed on the *body* of a pure method, allowing wide flexibility in how it is implemented provided that it exposes a pure interface.

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 verifiably pure, 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 show how Joe-E enforces determinism and thus 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.

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.

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 failover to a backup system that is receiving the same stream of input events.

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 , $\text{Decode}(\text{Encode}(x))$ should yield some output x' that is functionally equivalent to x . To ensure the original x will be recoverable in the future, we’d like this 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 [11]. 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.

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 with², 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` [5] 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

²Untrusted 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 [19], but that is beyond the scope of this paper.

remotely 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.

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" [16, § 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 [5]. 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.

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. 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) [5]. Verifiable purity would enable developers to check that this discipline was followed correctly and preserve it under maintenance.

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 [13]. Since JML specifications can be compiled to assertions and checked at runtime [9], 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.

3. REQUIREMENTS

Our approach to purity is based on leveraging the properties of a *deterministic object-capability language*, i.e., an object-capability language without any nondeterministic constructs.

An object-capability language [16] 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 particular, references serve as capabilities, and capabilities can be granted only by explicitly passing references. For instance, the global scope must not contain any capabilities.

In a deterministic object-capability language, the observable global state must never change and must be the same on every execution. 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 scope are globals and arguments, any variation in the method’s behavior can always be attributed to differences in its arguments.

3.1 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. 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.

4. DEFINITIONS

We are interested in whether a method is *observationally* functionally pure. There is no need to place any particular restriction on what the function does internally, as long as no external observer can observe any deviation from functional behavior. In the context of an imperative programming language, purity can be factored into two separable but related requirements. First, the method must not have any side effects; its only observable result must be its return value. Secondly, its behavior must be a deterministic function of its arguments: it must perform the “same” computation every time that it is invoked with the “same” arguments. Below we formalize these requirements more carefully, in the context of a deterministic object-capability language.

4.1 Side-effect freeness

A method is *side-effect free* if the only objects that the method ever modifies are those 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. Objects the method mutates will never be reachable from outside the method (except via the method’s return value), as it would be necessary to modify external objects in order for them to obtain references to the new objects.

4.2 Determinism

At a high level, a deterministic function should always give the same result when passed the same arguments, regardless of when and where it is invoked in a program. This applies even on different executions of the program, provided that the source code and libraries are not changed. Our approach to determinism requires the use of a strong notion of “sameness”: a program written within the language should not be able to distinguish between (i.e., behave differently when given) argument sets that we consider the same. A formal definition of equivalence that meets this requirement for deterministic object-capability languages is given below (in § 4.3).

We will consider a method M to be *deterministic in its arguments* if any two calls to that method with equivalent sets of named arguments A, A' will always yield a pair of return values r, r' such that the sets of references $A \cup \{r\}$ and $A' \cup \{r'\}$ are equivalent as defined below. (The original arguments must be included because aliasing relationships between the arguments and the return value are visible to the caller of the method.) The two calls to the method can occur at any point in any execution of the program.

For instance methods, the implicit `this` parameter is considered an argument, hence the behavior of an instance method is permitted to depend upon the state of the object on which the method is invoked.

4.3 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.

Many object-oriented languages include both *reference types* and *value types*. Objects of reference types 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 canonical null object belonging to every 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 reference 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 *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 referenced object 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 objects reachable from the set A and from global variables. See Figure 1 for an example of an object graph.

5. PURE METHODS

Languages that meet our requirements (§ 3) make verification of purity easy. If all parameters to the method (including the implicit `this` parameter) are statically declared to be of an immutable type, then the method will be pure. For instance, consider a method with the following type signature:

```
static String extractAddress(String message)
```

This example implements Bernstein’s address-extraction interface (see § 2.3). We can see that all of its parameters have an immutable type, so any implementation of this method must be pure. For instance methods, because the `this` parameter is also considered an argument, we must also check that the method’s declaring class is immutable.

When the purity condition is met, the method will have access to no external resources or shared mutable objects and thus cannot cause any visible side effects. Also, the only objects it can observe are those that are reachable from its arguments (which are immutable, if the arguments are) or those that it creates. Consequently, the method’s behavior can only depend on the pre-state of objects that are reachable from its arguments.

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. 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.

6. IMPLEMENTATION

The object-capability language Joe-E is specified in a technical report by Mettler et al. [14]. Joe-E is a subset of Java; the Joe-E

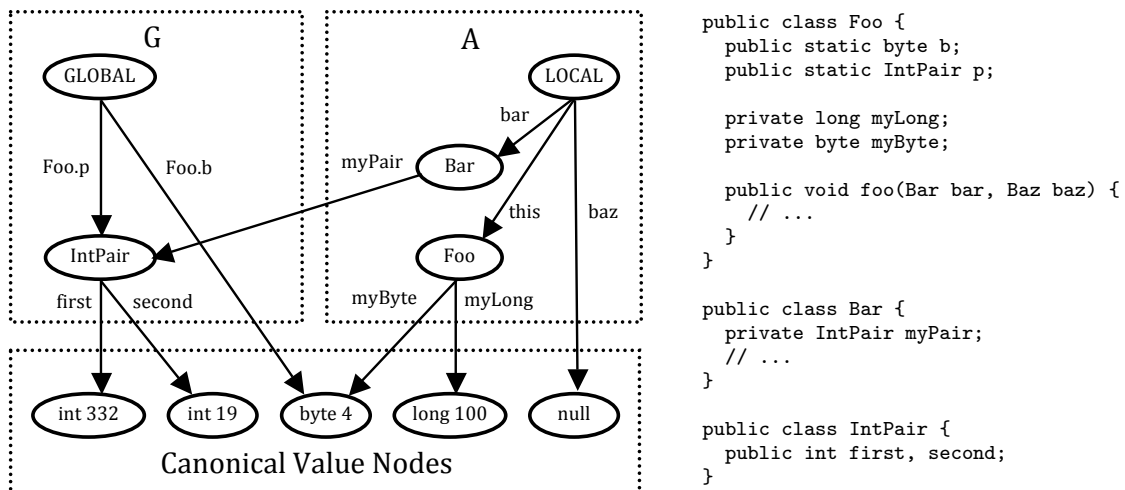


Figure 1: An example of an object graph and corresponding Java class definitions.

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. In this paper, we present how Joe-E satisfies the requirements articulated earlier (§ 3), eliminating many barriers to reasoning about purity that we would face if we worked with Java code.

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 7 in the Appendix for examples.

6.1.1 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 2, `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). 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. This approach makes our definition of determinism much 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

```

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

```

Figure 2: This method is not deterministic.

```

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

```

Figure 3: This 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")`.

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.

6.1.2 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 throws a `VirtualMachineError`. 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 4.

This limits the guarantees we can provide: we cannot promise

```

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

```

Figure 4: The JVM throws a `StackOverflowError` when the available stack space is exhausted, so this recursive method’s return value is nondeterministic.

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.

We mitigate this shortcoming by ensuring that if the JVM does throw a `VirtualMachineError`, then the program will terminate immediately—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 a `VirtualMachineError` occurs. See Appendix A for details.

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 [14]. The most relevant is that constructors must not call 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 with a common supertype. 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; for instance, `ByteArray` holds an immutable sequence of bytes.

6.3 Verifying Purity

With our extensions, Joe-E makes it easy to reason about purity. 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

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

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

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 1 for the three applications we analyze. (We used the Eclipse Metrics Plugin [21] for all code metrics.)

7.1 AES library

7.1.1 Motivation

We started with an open-source AES implementation written in Java [6]. 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 2.2.

7.1.2 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 `ImmutableArrays`, to meet Joe-E’s requirement that all global 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.

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

```
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.

7.2 Voting machine

7.2.1 Motivation

Next, we examined the serialization and deserialization code from an experimental voting machine implementation [20]. 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 2.2. This ensures that all votes that are successfully recorded will be read back correctly during vote tallying.

7.2.2 Changes to the codebase

Nearly all changes made were simple changes that involved changing the type of an array from a standard Java array type to a Joe-E immutable array type. Another common change was implementing the `Immutable` interface in classes that were already observationally immutable (which required nothing more than adding "implements `Immutable`" to the end of 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 original signature of the method that we wished to make verifiably pure was the method to check the serialization:

```
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.

7.3 HTML parser

Our third application, an HTML parser [18], 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 [1].

7.3.1 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 ParseException
```

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

7.3.2 Mutable static state

We removed several instances of mutable static state in 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; }
```

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.

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

Figure 5: 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.

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()`. Although we decided to remove debugging information for simplicity, a suitable solution for retaining it would be to return from the top-level `parse` method an object containing both the parse tree and the desired debugging information. 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.

7.3.3 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 § 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.

7.3.4 Immutable classes

After the 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. The code was refactored to construct and use a `Lexer` inside the top-level `parse` method. A typical use of the `Parser` class can be seen in Figure 5.

Making the classes that implemented the `NodeFilter` interface immutable was straightforward, except in the case of the `IsEqualFilter` class. This required significant effort due to the fact that the `IsEqualFilter`, 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 become immutable proved difficult due to a nonstandard construction pattern. To allow for the creation of custom parsers that recognize varying sets of HTML

tags³, a prototype construction pattern was used: a set of `Node` prototypes would be registered before beginning to parse. 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.

The code was refactored 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.

An additional step necessary in realizing immutable `Node` classes was the splitting of the `Page` class, which conflated two distinct purposes into a single class. Before refactoring, the `Page` class 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 functionality precludes making the class immutable. 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, an immutable class called `PageInfo` was created to hold this information, which can then be extracted from the mutable `Page` class. Now, when the `Lexer` creates a `Node` by the `Lexer`, it obtains the `PageInfo` from the `Page` and passes it to the `Node`'s constructor.

As demonstrated 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. In particular, there cannot be any cycles in immutable data structures, which must be constructed from the bottom up.

We also changed the `parse` method to return an immutable data structure, of type `ImmutableNodeList`. This is not necessary for purity, but it aids the creation of other pure methods that use the object returned by the `parse` method, since they can directly use the returned parse tree as a parameter. Had a non-immutable structure been returned by the `parse` method, the caller of another pure method (for example `extractLinks(ImmutableNodeList)`) would have to create an immutable copy before calling the second method.

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

³For 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 2: The number of pure and impure methods and constructors in the Waterken Server.

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 split the two into separate classes.

7.5 Waterken Server

The Waterken server is an extensible web server designed for building distributed web services [10]. 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 implementing 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 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.

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.

9. RELATED WORK

Object-capability languages have a long history [16]. Most object-capability 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” [23]. The auditor verifies this property using runtime introspection on an object; in contrast, we verify it statically for individual methods. Additionally, E’s `Functional` auditor requires pure methods to have an immutable return type, which we do not.

We were inspired by the notion of “environment-freeness” [20], 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.

Jif [17] extends Java with label-based information flow checking. 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.

In contrast, while our approach does not allow for the rich policies expressible in Jif, it is better suited to enforcing determinism, as we can prevent information flow from any non-constant state, not just data that is tagged with a principal.

Most exploration of purity in the context of languages such as Java has focused on side-effect freeness rather than determinism. We suspect this is the result of interest primarily in using pure methods for specifications in frameworks like JML. While determinism is also necessary for such specifications to be statically well-defined, a condition with side effects has the larger problem that enabling assertions changes the semantics of the program.

Spec# [4] and JML [7, 13] 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 [13]. They support annotating methods with the `pure` attribute, but pureness as defined for JML includes only side-effect freeness and not determinism. Also, JML does not check that a method declared pure is actually pure. Chase [8] is a static checker for JML's *assignable* clause that can be used to verify side-effect freeness of a method by verifying that no variable is assignable from the method. Unlike our approach, Chase does not consider aliasing.

In strictly functional languages, like Haskell, all functions are pure. Methods that would normally cause side effects instead return a monadic type that lists these effects [22]. The effects can be ignored by the caller if it prefers to do so, but syntactic sugar makes it easy to chain monads, giving a program that reads similarly to an imperative one. In contrast, we allow a programmer to introduce purely functional code in an imperative language. While less disciplined than monadic programming style, our approach permits one to focus on obtaining purity where needed, while allowing the rest of the program to be written in an imperative style, reducing the changes needed for existing code and programming patterns.

Other systems have mixed imperative and functional programming styles to varying degrees. The Eiffel language [15] 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 [12] and SPARK [3] 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 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.

10. CONCLUSIONS

Verifiable purity is useful for verifying many kinds of high-level security properties. We showed that an appropriately-designed programming language can greatly simplify the task of writing pure code and verifying that it is pure. By leveraging object-capabilities, this can be done in the framework of a primarily imperative language. Thanks to this approach, Joe-E allows programmers to write verifiably pure code within a conventional imperative language.

11. REFERENCES

- [1] HTML4 Test Suite.
<http://www.w3.org/MarkUp/Test/HTML401/current/>.
- [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] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [4] 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.
- [5] 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.
- [6] L. Brown. AEScalc. <http://www.unsw.adfa.edu.au/~lpb/src/AEScalc/AEScalc.jar>.
- [7] 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.
- [8] N. Cataño and M. Huisman. CHASE: A static checker for JML's assignable clause. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 26–40, London, UK, 2003. Springer-Verlag.
- [9] Y. Cheon and G. Leavens. A runtime assertion checker for the Java Modeling Language, 2002.
- [10] T. Close and S. Butler. Waterken server.
<http://waterken.sourceforge.net/>.
- [11] 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.
- [12] 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.
- [13] G. Leavens and Y. Cheon. Design by contract with JML, 2003.
- [14] A. Mettler and D. Wagner. The Joe-E language specification (draft). Technical Report UCB/EECS-2006-26, EECS Department, University of California, Berkeley, March 17, 2006.
- [15] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, Englewood Cliffs, NJ, USA, 1992.
- [16] 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.
- [17] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.
- [18] D. Oswald, S. Raha, I. Macfarlane, and D. Walters. HTMLParser 1.6.
<http://htmlparser.sourceforge.net/>.
- [19] A. Rudys and D. S. Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security*, 5(2), May 2002.
- [20] N. K. Sastry. *Verifying Security Properties in Electronic Voting Machines*. PhD thesis, University of California at Berkeley, 2007.
- [21] F. Sauer. Eclipse metrics plugin 1.3.6.

```

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

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

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

```

Figure 6: finally clauses expose nondeterminism.

```

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

```

Figure 7: Several methods from the Java library are nondeterministic and callable by all Java code. These are not exposed to Joe-E code.

<http://metrics.sourceforge.net/>.

- [22] 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, Albuquerque, New Mexico, 1992.
- [23] K.-P. Yee and M. Miller. Auditors: An extensible, dynamic code verification mechanism, 2003. <http://www.erights.org/elang/kernel/auditors/index.html>.

APPENDIX

A. FINALLY CLAUSES

Figure 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 doubles until triggering an `OutOfMemoryError`. This causes the `finally` clause to execute, which will then throw 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.