Class Properties for Security Review in an Object-Capability Subset of Java (Short Paper)

Adrian Mettler David Wagner
Department of Electrical Engineering and Computer Science
University of California, Berkeley, USA
{amettler, daw}@cs.berkeley.edu

ABSTRACT

Joe-E is a subset of the Java language, with additional restrictions enforced by a static source-code verifier. We explore several semantic properties of classes relating to immutability and object identity that can be declared by the programmer and are checked by the Joe-E verifier. We present the simple, modular analyses we use to verify these properties and describe how they are useful in performing security reviews of applications.

1. INTRODUCTION

In designing code for security review, it is helpful to make it easy to characterize the properties of objects in a program. A sufficiently rich type system can help with this goal: programmers can document the properties of objects through annotations on their types, and an appropriately constructed static code verifier can check that these properties will hold for all instances of these types. Once a desired property has been statically verified to hold for all instances of a particular type, a code reviewer can use this property in reasoning about the structure or behavior of the associated type.

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

The alternative is to run the verifier on all code, and to verify properties of classes when any of their supertypes has an annotation declaring that property. In other words, the annotation used to declare and verify properties of a class must be inherited by subclasses. In Java, this can be accomplished by using interface types as annotations. We define a *marker interface* for each semantic property we want to verify. Then, a class may implement a marker interface to declare that it satisfies the property associated with that interface, and the verifier checks that the associated properties hold for every class that implements the marker interface.

In Joe-E, we use marker interfaces to verify a number of interesting class-based properties, generally relating to immutability and object identity. We have identified simple source code analyses that can be used to verify these properties statically.

Immutability has a number of applications to supporting security reasoning. It is safe to share immutable objects, without risk that the recipient might be able to modify something we are relying upon. Immutability is also useful for defending against time-of-check-to-time-of-use attacks, as the recipient of an immutable object knows that it will not change. This application requires a subtly stronger immutability guarantee than previous work on verifying immutability for Java has enforced; we explain why and how we enforce the stronger notion in Joe-E.

Due to Java's == operator, which tests pointer equality, every object has an unforgeable identity in addition to its contents. Identity tests mean that any object can be used as a *token*, serving as an unforgeable proof of authorization to perform some action. For example, an object can store a reference to a particular token in a private variable and then only perform certain actions when the same token is passed as a method argument. In programs that make use of tokens, it would be helpful if the type system could help us reason about how tokens can propagate in a program. Joe-E defines a marker interface that ensures that an object is free of authority-bearing tokens, and thus can be passed to untrusted code without accidentally conveying any tokens.

Additionally, the presence of pervasive object identity in Java means that no reference type can be a true value type. To address this limitation, Joe-E restricts the use of == and `!=`, hiding the object identity of some types. Joe-E's support for value types prevents errors where their object identity is accidentally used and strengthens the applicability of Joe-E's determinism guarantees.

2. JOE-E AND MARKER INTERFACES

Joe-E [7] is a subset of Java, designed to make security review of applications easier. We wish to make it feasible for reviewers of an application to formulate application-specific security properties and convince themselves that a program satisfies these security properties. We support a number of patterns of reasoning that reviewers can use to soundly check that a program has a desired security property. These patterns are based on capability-style reasoning about the propagation of references in a program, and thus are only valid in an object-capability language. Joe-E imposes a number of restrictions on Java language features so that the resulting smaller language is a deterministic object-capability language.

As a deterministic object-capability language, Joe-E provides two linguistic security properties:

- **Object-capability paradigm.** In Joe-E, all *authority* is embodied by object references, which serve as capabilities. Authority refers to any effects that running code can have other than to perform side-effect-free computations or to throw a virtual machine error due to resource exhaustion. Authority includes not only effects on external resources such as files or network sockets, but also on mutable data structures that are shared with other parts of the program.

  As an object-capability language, Joe-E ensures that authority is granted only via object references. It ensures that the global scope, the state reachable from static fields in Java, does not contain any authority and that static methods and constructors do not grant authority to their callers.

- **Determinism.** Joe-E ensures that code's behavior depends deterministically on only its inputs, unless it is explicitly given access to nondeterministic language features or external resources.

The additional linguistic restrictions imposed upon Joe-E code fall into two categories:

- Restrictions placed on only those classes that implement certain *marker interfaces*. These interfaces act as inheritable annotations indicating that a particular class should be verified to have a specific property.

- Restrictions that apply to all Joe-E code. This includes limitations on the use of various language features, such as native methods and the `finally` keyword, and limitations on which standard Java libraries can be used by Joe-E code.

This separation ensures that the restrictions can be enforced in a modular analysis. Much like the Java compiler, to analyze a particular file, the Joe-E verifier needs the source code of only that one file; it examines interface-level details of other classes but not their code. This means that Joe-E verification scales well (roughly linearly) to large code bases.

2.1 Marker Interfaces

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

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

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

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

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

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

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

3. IMMUTABILITY

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

Immutability is helpful for reasoning about the correctness and robustness of code, for a number of well-known reasons. One particular reason motivates our strong immutability requirement: if an object is immutable, code that makes use of it does not need to defend against modifications to that object by other code in the system. For instance, this eliminates the possibility of time-of-check-to-time-of-use attacks on those values. Also, immutability facilitates verification of functional purity (determinism and side-effect freeness) of methods [5].

In contrast with other type systems for immutability such as Immutability Generic Java [14], in which references to read-only or immutable objects are subjected to additional type checks, in Joe-E we only support class immutability, i.e., classes with no mutable state. This is less expressive, but we have found it to be sufficient for new code. Class types that implement the `Immutable` marker interface are verified to be immutable, and those that do not are not.

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

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

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

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

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

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

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

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

3.1 Ensuring Final Means Final

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

To address this problem and ensure that `Immutable` objects are truly immutable, Joe-E prevents Joe-E code from reading a field before it has been initialized. We ensure this by preventing the `this` pointer from escaping from any constructor[1], enforced as follows:

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

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

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

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

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

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

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

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

Figure 1: A locked box class. Once the content is stored in the box, it can only be retrieved again given the key object.

Java has a similar weakness with static final fields: if there is a circular dependency in classes' static initializer logic, it is possible for code executing during static initialization of these classes to see uninitialized values for their fields. Joe-E does not currently address this problem, which does not technically violate our object immutability guarantees, as it only affects static fields. However, if an immutable object reads a value from a static field when an instance method is invoked, it may return different results on different invocations, which appears the same as a change in state, so we would still like to close this loophole.

4. IDENTITY-BASED AUTHORITY

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

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

We can reason more precisely if we verify and rely on certain behaviors of shared objects in the reference graph. Consider the locked box class presented in Figure 1. (This is an adaptation of a construction [9, §6] that dates back to 1973.) The box's constructor accepts an object to store in the box and a key object. The key must be presented to extract the object from the box. The class `Token` is an empty class used here solely for pointer comparison.

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

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

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

4.1 Power and Tokens

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

As another example, consider the currency system depicted in Figure 2. Each Currency object corresponds to a different currency, and is used to ensure that money is not accidentally transmuted from one currency to another. Additionally, it gives its holder the ability to mint virtual coins in that currency, even though the class itself has no fields or methods. A Purse object is used by a client, such as an object representing a player of an online game, to hold a number of units of a particular currency. If a client wants to transfer some money to another object, he first constructs a new, empty purse of the same currency using the unary constructor, then transfers some money into this new purse from his primary purse using the takeFrom method. The new purse can then be passed to the recipient, who can add the balance from it to her own main purse also using takeFrom. The Purse class can be reviewed to verify that currency cannot be created without the use of the Currency token; this reduces the portion of the program that would have to be reviewed to ensure that there are no bugs that might allow money to be created from nothing.

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

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

- Conservatively assume that any authority made available by the object identity of non-tokens is available everywhere in the program.

```
public final class Currency extends Token { }

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

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

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

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

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

Figure 2: A secure abstraction that supports flexible use of currencies.

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

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

To encourage using tokens solely for their object identity, and not as containers for other capabilities, the class Token is declared to implement Immutable. Thus, tokens (including subtypes of Token) cannot contain mutable objects. Token also implements Equatable so tokens can be compared for identity (see Section 5).

4.2 Powerless

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

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

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

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

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

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

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

In Joe-E, in order to achieve the principle of least authority, we restrict global (static) fields to only contain `Powerless` objects. In addition to ensuring that the global scope does not contain any mutable state, this ensures authority-bearing tokens are not made globally available. A bigger concern is accidental escalation or malicious transmission of privileges due to an exception being thrown and caught. As exceptions are often hard to predict and reason about when performing a code review, in Joe-E we require all subtypes of `java.lang.Throwable` to be powerless in order to ensure that they cannot be used to transmit authority in unexpected ways [7, § 4.3]. This allows us to limit our examination to the more explicit mechanisms for reference propagation when reasoning about how objects can communicate with the rest of a program.

5. SELFLESS AND EQUATABLE

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

Consider, for instance, Figure 3, where a buggy method compares two strings using `==` instead of `equals()`. This breaks the abstraction that strings should be value types. Java has no way to enforce that code treats `String` as a value type; in contrast, Joe-E hides the object identity of `String`s and similar library classes

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

Figure 3: A method that violates the intuitive expectation that `String` is a value type. This code is probably a bug (and would not be allowed in Joe-E).

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

Figure 4: The bug in Figure 3 might not be detected by testing (due to automatic interning of string literals), but might trigger in practice.

from Joe-E code, making these library classes true value types. Such abstraction-violating bugs can sometimes be tricky to detect. For example, in Figure 4, testing code that makes use of literal (and thus automatically-interned) strings will fail to replicate the incorrect behavior of the `isYes()` method when called with non-interned strings from user input that have the same contents.

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

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

A class may be marked as being *equatable* by implementing the `Equatable` marker interface. Joe-E code is allowed to compare two references using `==` and `!=` if at least one of the references' declared types is equatable, or if either reference is `null`. Specifically, we ensure that, for every use of the binary operators `==` and `!=`, the type resolved for at least one of the two operands must be either the null type, a primitive type, or a reference type that implements Equatable in the overlay type relation.[2] All other uses of == and != are prohibited.

For instance, the buggy code in Figure 3 would not be allowed in Joe-E, because String is not an equatable type, and thus the use of == found there would result in a verification-time error.

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

1. All instance fields of C must be final and may not be transient.
2. C must not be equatable.
3. The object identity of instances of the class must not be visible. This can be satisfied by one of:
   (a) C's superclass is a selfless type, or
   (b) C's superclass is java.lang.Object, C overrides equals() and hashCode(), and C doesn't call super.equals(). (No Joe-E class is allowed to call Object's version of hashCode(), even on itself, due to its exposure of nondeterminism.)

Joe-E provides several guarantees about selfless objects:

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

Selfless classes are useful for constructing serialization code. To serialize a selfless object, we only need to serialize its contents (and possibly their identity); we do not need to record its own identity. This makes it easier to ensure that the result of serializing and deserializing an object is indistinguishable from the original. For instance, the Waterken server [7, § 7.1] uses Joe-E's Selfless interface to ensure the correctness of a performance optimization: when serializing a non-selfless type, we must maintain a unique serialized version per instance, whereas for selfless types, it is safe to make multiple copies of a single instance if that improves performance.

A *deep selfless* class C must satisfy the requirements for a selfless class. Also, all instance fields, all local variables of enclosing scopes observable by C, and all enclosing classes (if C or any superclass is a non-static inner class) must be deep selfless. Joe-E does not yet implement deep selfless, but it would be a straightforward addition. A deep selfless class is also powerless (but not necessarily vice versa).

One feature of Joe-E is that it makes it easy to verify when methods are definitely deterministic [5]. By deterministic, we mean that two successful invocations of the method with equivalent arguments will always yield equivalent results. The notion of "equivalence" depends upon the type of the object. If any arguments are not deep selfless, equivalence must take into account object identity: for instance, two immutable objects are equivalent if they have equivalent contents as well as the same object identity (or more generally, the same set of aliasing relationships to other objects in the method's scope, including other arguments and global variables). Put another way, if all we know about the method's arguments is that they are immutable, then we cannot rule out the possibility that the method's behavior and return value might depend upon the identity of its arguments. This notion of determinism is often weaker than we might prefer.

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

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

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

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

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

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

6. RELATED WORK

Our use of an augmented overlay type system follows E [8], which also provides a mechanism for indicating that Java classes "honorarily" satisfy similar object properties, though unlike our approach, in E these properties are not represented by types in the Java type system. The work that most closely resembles our use of marker interfaces is the Auditors framework for E, which uses runtime introspection of an object's AST to verify annotated properties such as immutability and selflessness [13]. In contrast to that work, we verify similar semantic properties on a per-class basis in a class-based language.

In this paper, marker interfaces' semantic restrictions apply on a per-type basis. For example, specific classes in the Java type system are considered immutable; standard Java type safety and final field enforcement ensures that objects of such classes are never mutated after construction. An alternative, at least for immutability, is to use an extended type system that treats some references or instances as read-only while allowing others to be mutated. The C++ const qualifier for pointers is the most well-known example

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

of this. It is a compilation error to assign the fields of, or invoke a non-`const` method on, a `const` reference. Its use in preventing side effects is limited because the restrictions are not transitive; it is possible to modify an object contained in a field reached via a `const` pointer. A transitive analogue of this was introduced by the KeyKOS operating system [6] as a "sensory key"; such a key prohibits writes and also causes all keys retrieved through it to be sensory. This concept is also found in the type system of a few programming languages to improve reasoning about immutability and side effects. Such types allow for documentation and modular checking of effect restrictions on a per-function basis.

The Javari [12] type system provides similar qualifiers for Java; its `readonly` qualifier serves as a transitive, sound version of `C++`'s const. Like C++, Javari provides a way for fields of a class to be declared as exempt from the `readonly` restrictions. This is potentially problematic where the immutability of an object of untrusted type is necessary to guarantee a security property. In addition to a sound, transitive version of `const` with no escape clauses for mutable fields, the D language [2] provides an instance-immutability qualifier `invariant` that can be used to ensure that a specific instance of an object will never be modified. This has the benefit of allowing the same type to be used in immutable and non-immutable forms. Immutability-Generic Java [14] supports both transitive-const (`ReadOnly`) and instance-immutability (`Immutable`) annotations, as an extension to the Java type system. Like Javari, they provide a mechanism for a class to declare fields as exempt from the read-only restrictions, which we would consider a soundness hole. Also in contrast to our work, they do not consider the escape of a `ReadOnly` reference to a partially-constructed object to be a violation of their immutability property. Pluggable type system frameworks like JavaCop [1] and the Checker Framework [10] can also be used to define type checks that enforce semantic properties such as immutability.

Previous work has addressed the problem of partially-constructed objects in the context of non-null types for instance fields in object-oriented imperative languages such as Java and C#. A number of papers have presented type systems to properly type partially-constructed object instances. Raw types [3] indicate which levels of the type hierarchy have not yet performed their share of an object's initialization. Delayed types [4] are associated with a lexical scope in which the object is not fully initialized, but ensure that all associated objects are fully initialized before the scope is exited. Masked types [11] explicitly specify which fields of an object have not yet been initialized, and thus cannot yet be used. These approaches would provide a more precise way to address the use case of wanting to pass a reference to an object under construction to other objects, allowing for more complex initialization patterns, e.g., creation of circular verifiably-immutable data structures.

7. CONCLUSIONS

We have identified a number of useful semantic properties of classes that support reasoning about applications. We provide simple type checking rules for ensuring these properties of programs written in a subset of Java. These class properties are instrumental to Joe-E's support for the design and construction of software to support security review.

8. ACKNOWLEDGEMENTS

We would like to thank our anonymous reviewers for their helpful feedback and our shepherds, Úlfar Erlingsson and Todd Millstein, for guidance in refocusing and substantially improving this paper.

This work was supported by the National Science Foundation under grants CNS-0716715, CCF-0424422, and CCF-0430585 and by a Siebel Scholarship. 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.

9. REFERENCES

[1] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *OOPSLA '06: 21st ACM Conference on Object-Oriented Programming Systems and Applications*, pages 57–74, Portland, Oregon, USA, 2006.

[2] W. Bright. D language 2.0. `http://www.digitalmars.com/d/2.0/`.

[3] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA '03: 18th ACM Conference on Object-Oriented Programming Systems and Applications*, pages 302–312, Anaheim, California, USA, 2003.

[4] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA '07: 22nd ACM Conference on Object-Oriented Programming Systems and Applications*, pages 337–350, Montréal, Québec, Canada, 2007.

[5] M. Finifter, A. Mettler, and D. Wagner. Verifiable functional purity in Java. In *ACM Computer and Communications Security (CCS 2008)*, pages 161–173, Arlington, VA, USA, October 27–31 2008.

[6] N. Hardy. KeyKOS architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, 1985.

[7] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *Network and Distributed Systems Symposium (NDSS 2010)*, pages 357–374, San Diego, CA, USA, February 28–March 3 2010.

[8] 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.

[9] J. H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.

[10] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for java. In *ISSTA '08: 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, 2008.

[11] X. Qi and A. C. Myers. Masked types for sound object initialization. In *OOPSLA '09: 24th ACM Conference on Object-oriented Programming Systems and Applications*, pages 53–65, Savannah, Georgia, USA, 2009.

[12] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA '05: 20th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.

[13] K.-P. Yee and M. Miller. Auditors: An extensible, dynamic code verification mechanism, 2003. `http://www.erights.org/elang/kernel/auditors/index.html`.

[14] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. D. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE 2007: 11th European Software Engineering Conference and 15th ACM Symposium on the Foundations of Software Engineering*, pages 75–84, Dubrovnik, Croatia, September 5–7, 2007.