The Joe-E Language Specification, Version 1.1

Adrian Mettler David Wagner
{amettler,daw}@cs.berkeley.edu

September 18, 2009

1 Introduction

We describe the Joe-E language, a capability-based subset of Java intended to make it easier to build secure systems. The goal of object capability languages is to support the Principle of Least Authority (POLA), so that each object naturally receives the least privilege (i.e., least authority) needed to do its job. Thus, we hope that Joe-E will support secure programming while remaining familiar to Java programmers everywhere.

2 Goals

We have several goals for the Joe-E language:

- **Be familiar to Java programmers.** To minimize the barriers to adoption of Joe-E, the syntax and semantics of Joe-E should be familiar to Java programmers. We also want Joe-E programmers to be able to use all of their existing tools for editing, compiling, executing, debugging, profiling, and reasoning about Java code.

  We accomplish this by defining Joe-E as a subset of Java. In general:

  > *Subsetting Principle:* Any valid Joe-E program should also be a valid Java program, with identical semantics.

  This preserves the semantics Java programmers will expect, which are critical to keeping the adoption costs manageable. Also, it means all of today's Java tools (IDEs, debuggers, profilers, static analyzers, theorem provers, etc.) will apply to Joe-E code.

  In this document, we define the Joe-E language by specifying constraints that will be verified by a Joe-E verifier. These checks may be performed at the source level, or possibly upon bytecodes produced by a compliant compiler.

- **Include as much of Java as possible.** Some Java constructs must be omitted from Joe-E, because they are incompatible with capability programming. However, we ideally want to retain as much of Java's expressiveness as possible.

  > *Maximal Subset Principle:* Choose the largest subset of Java that is compatible with secure capability programming. Forbid only language constructs that render capability-style programming or reasoning impossible or error-prone.

  Thus, Joe-E will permit construction of secure programs, but it will not guarantee that programs in this subset will be secure.

- **Enable capability-style reasoning about Joe-E code.** To ensure that objects receive the least authority needed, the language must allow capability-style reasoning. Capability-style reasoning involves thinking about the directed graph of object references: if object O has a reference to O', then we draw an edge $O \rightarrow O'$ in the graph. To characterize how O can causally affect the outside world (its authority), we examine the set of capabilities that O might obtain. An upper bound for this set is the bidirectional transitive closure of this graph, i.e., the set of objects reachable from O via backward and forward edges in the reference graph. If the code is constructed appropriately, it is often possible to improve upon this approximation by verifying that the program semantics for some methods prevent the transitive closure worst-case from occurring; it should be easy to write code like this (e.g., facets), and it should be easy to verify the correctness of this code through purely local reasoning.

  Note that this style of reasoning assumes that references are unforgeable and that references are the only thing that convey authority—in particular, that there is no ambient authority.

- **Unforgeable references.** It must be impossible to manufacture a reference to an arbitrary object. Java's memory-safety ensures that objects can only obtain references through specific controlled mechanisms. Specifically, references held by an object O can originate only in the following ways.

  1. **Endowment:** When O is instantiated, it receives as its birthright any references passed to its constructor and (if it is a non-static inner class) anything visible as part of its lexical scope (including a reference to itself, via `this`);
  2. **Parenthood:** When O creates a new object, it receives a reference to that object.
  3. **Introduction:** References can be introduced to an object in a variety of ways, namely:
     (a) By field access: If O has a reference to O', then O can obtain any references stored in the accessible fields of O'.
     (b) By field mutation: If O has a field accessible to another object, O receives any references stored into this field by that object.
     (c) By invoking: When O calls a method on some other object, O receives any reference returned or thrown by that method.
     (d) By being invoked: When a method is called on O, O receives any references passed as the arguments to this method call.

- **No ambient authority.** A precondition for capability-style reasoning is that the only way for an object O to affect the outside world is through the references it possesses. To ensure least authority and support capability-style reasoning, the references that O possesses should be limited by lexical scoping rules: e.g., limited to its fields, with no "global variables" allowed. For instance, `java.lang.System.out` violates this principle, because it is available to every object and allows causally affecting the outside world.

  Avoiding ambient authority is important for POLA. It should be easy to limit the amount of authority given to an object to only those capabilities necessary to perform its function, and to do so in a way that is foolproof and easy for an human auditor to recognize. In particular, the rule should be that no object receives a capability unless it has been explicitly granted that capability in one of the ways described above. Authority-bearing variables that are defined in a location distant from some code that can use them should be avoided, because they violate this rule. In addition to this static (code-based) constraint, sharing of state between objects associated with different flows of control or protection domains should be avoided as much as possible, even if they share some source code. In general, since ambient authority is available to all objects, ambient authority is incompatible with POLA and must be avoided at all costs.

- **Support local reasoning about Joe-E programs.** A closely related goal is that it should be possible to reason about Joe-E programs through only local analysis. Suppose we are given a Joe-E program, consisting of many objects. If we are given the code to one object O, we would like to be

able to reason about the capabilities that O might have access to and might pass on to others. It should be possible to reason about this just by looking at the code of O and the objects it interacts with (possibly continuing transitively as far as is needed).

For instance, suppose method `C.m()` creates a new object T. If T never escapes from the method, then we would like to be able to conclude that no other object can affect T. Moreover, we would like to be able to verify, just by looking at the code of the class `C`, that T cannot escape. This is an example of local reasoning, a kind of reasoning about composition, and the language should support this kind of reasoning. Such reasoning might assume nothing about the rest of the program, other than that it is valid Joe-E code; the benefit is that a programmer does not have to keep the entire program or system in his head in order to reason about local properties of his code.

As another important example of this, it must be possible (preferably with very little effort) to bound the capabilities granted to any object O. This should be possible even when the code of object O has not been analyzed; typically, it is done by reasoning about the capabilities that are passed to O when it is created and thereafter. This task allows for the creation of systems that are secure in the face of unknown, possibly adversarial code.

- **Unforgeable types.** If we invoke a method on object O, we may be relying on it to behave as we expect. Therefore, we may need some way to verify that O is the correct entity before we invoke method calls on it. In the E language, this is accomplished with guards, auditors, and introspection; in Joe-E, we use types. Some classes are part of the base system and hence trustworthy; other classes are provided by the programmer and thus trusted. Joe-E programs should be able to check that the type of O is the expected one before invoking method calls on O. This ability is provided by Java's type system.

  Without this feature, masquerading attacks are a serious risk. Suppose that method `m()` accepts an object O and uses it to perform various operations. If `m()` does not validate the type of O, then an attacker might be able to call `m()` and pass a malicious object O_M. For instance, O_M might undetectably emulate the expected behavior while simultaneously observing information that `m()` passes to its argument, leaking these secrets back to the attacker. Or, O_M might behave in unexpected ways: e.g., if `m()` expects its argument to behave like an `Integer`, then `m()` might be surprised if two attempts to read its contents return different results (this can lead to TOCTTOU attacks, for instance). It is essential for there to be a way to avoid this kind of masquerade attack.

- **Permit use of capability discipline.** Capability discipline refers to a set of guidelines for writing programs in a way that maximizes the chances that the program will be secure and respect POLA. Suppose I want to give Alice access to a file on the filesystem. I can give her a reference to a `File` object F. Of course, in doing so, I have given Alice all of the authority that can be invoked through the interface of this object. If every `File` object has a `formatHardDrive()` method that erases the entire hard disk, then in giving Alice access to F, I have also given her the ability to re-format the hard disk, a violation of POLA. In this case, we say that F fails to respect capability discipline.

  It is a goal of Joe-E that it should be natural and easy to build classes that respect capability discipline. It is not a goal of Joe-E to somehow guarantee that every Joe-E class will respect capability discipline; capability discipline requires knowledge of application semantics and the desired security policy, and hence cannot be enforced at the language level.

  Our general stance in defining Joe-E is include everything that is not outright incompatible with capability-style reasoning (see the Maximal Subsetting Principle). One can identify many syntactic source code patterns that are suspicious and often correspond to violation of capability discipline (e.g., `public` fields), but that are not inherently incompatible with security. We do not attempt to forbid such syntax in the Joe-E language. One might build a separate "capability-style lint" to check for suspect language constructs that are risky but not necessarily incompatible with capability reasoning; however, such considerations are out of scope for this document.

- **Provide a set of capability-friendly base classes.** Joe-E should provide a set of library classes that enable programming in the capability style. At a minimum, this collection should contain the minimum necessary to build useful Joe-E programs. These base classes should respect capability discipline and should be constructed to maximize the likelihood that programs built using the base classes will respect POLA and will be secure.

3 Definitions

The Joe-E Language is a subset of the Java source language as defined in the Java Language Specification, 3rd Edition.

3.1 Power and Tokens

The Joe-E language is designed to facilitate reasoning about a conservative approximation to authority that we denote *power*. In order to simplify reasoning, we consider the object identity only of a specific subclass of objects we call *tokens*, and conservatively assume that any authority made available from the object identity of other objects is available everywhere in the program.

A class is a *token class* if it is the class `org.joe_e.Token` or any of its subclasses. An instance of any such class is a token.

3.2 The Overlay Type System

Joe-E defines interfaces (most with no members) that are used as inherited annotations on Java classes. These are called *marker interfaces* and are used to indicate properties of importance to the Joe-E language and verifier. In many cases, it would be appropriate for classes in the standard Java libraries to implement these interfaces. However, we cannot modify the existing Java class libraries. Instead, Joe-E defines an extended type system consisting of additional interface-implementation relationships overlaid on top of the Java type system.

The *base type system* is the type system defined by the Java language; it defines certain subtyping relationships. In addition, Joe-E provides a way to declare a Java library class to *honorarily implement* a specified interface, and this defines some additional subtyping relationships. The *overlay type system* is defined as the union of the subtyping relationships from these two sources, along with their transitive closure. Note that since honorary components of the type system only add interfaces to classes, the overlay type system will be a consistent, "legal" typing relation (no circular subtyping, etc).

All type checking performed as part of the standard Java compilation process and JVM runtime enforcement uses the base type system, as required to preserve Java semantics. However, Joe-E's additional restrictions are defined in terms of the overlay type system.

3.3 Compliance and Deeming

Each marker interface is associated with a set of restrictions that must hold for all classes implementing that interface. The obligation to satisfy these restrictions is automatically inherited by any subclasses of a class declared to implement the interface, as these subclasses also implement the associated interface. The inheritance of restrictions is necessary to conclude that any object assignable to a variable of some type fulfills that type's restrictions.

These restrictions are automatically checked as described in Section 4; the decision procedure used is sound but not complete. Because there will be classes that satisfy the contracts associated with a marker interface, but which cannot be automatically verified to do so, a Joe-E implementation may *deem* certain classes to satisfy the interface, exempting them from automated verification. Such deemings must be made with care, and are restricted to classes in the Java and Joe-E libraries that have been manually verified to satisfy the deemed interfaces. Note that being deemed to implement an interface is distinct from being

declared to implement that interface honorarily. Neither implies the other: A class can explicitly implement an interface but not be automatically verifiable to meet its contract and thus require deeming. On the other hand, a class that implements an interface honorarily might automatically be verified to satisfy its requirements.

Each class considered to satisfy a marker interface without requiring automatic verification must be individually deemed to implement that interface; deeming decisions are not automatically inherited. This is because a subclass may fail to meet some obligations that its superclass was manually verified to maintain.

The following invariants will hold for the set of honorary interface implementations and deemings made in a Joe-E implementation:

1. A class can only be deemed to implement an interface I if it implements I in the overlay type system (i.e., either in the base type system or honorarily.)

2. If classes C and D are both library classes, if C honorarily implements an interface I, and class D extends C, D must also honorarily implement I.

3.4 Immutable Types

A type T is *immutable* if and only if it implements the marker interface `org.joe_e.Immutable` according to the overlay type system. The (empty) `org.joe_e.Immutable` interface must be provided by the Joe-E implementation.

The intuition behind an immutable object is that such an object cannot be changed (mutated) in any observable way, nor can any objects reachable by following the fields of the immutable object. The contents of an immutable objects' fields and any objects reachable from an immutable object must not change once the object is constructed. With the exception of library classes explicitly deemed to implement `Immutable`, an immutable class must satisfy additional linguistic restrictions enforced by the verifier (§4.4) to ensure this property. Library classes that cannot be automatically verified and are deemed immutable must be carefully manually verified to expose no possibility for modification of their contents.

Note that immutability does not place any restrictions on any local variables defined within the immutable class. It also says nothing about the mutability of the arguments passed to methods. It only applies to the values stored in and objects reachable from the immutable class's fields.

3.5 Powerless Types

A type T is *powerless* if and only if it implements the marker interface `org.joe_e.Powerless` according to the overlay type system. The `org.joe_e.Powerless` interface must be provided by the Joe-E implementation, and it must be declared to extend the `org.joe_e.Immutable` interface. This ensures that every powerless type is also immutable (but not necessarily vice versa). A *powerful* type is one that is not powerless.

A Joe-E implementation must ensure that the following types honorarily implement `org.joe_e.Powerless`:

- the primitive scalar types (`boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, and `double`)
- `java.lang.Throwable`
- `java.lang.Enum`.

(These types are designated powerless using the honorary mechanism, since they cannot be declared to implement this interface explicitly.)

A Joe-E implementation may declare specific additional types from the standard Java libraries and from the Joe-E library to implement `org.joe_e.Powerless`, so long as their behavior can be verified (manually, in the case of classes deemed `Powerless`) to be that of a transitively immutable object that does not reveal the identity of any tokens.

Rationale: A powerless object conveys no inherent or identity-based power and thus can be excluded from the object reference graph without loss of soundness. Any authority granted to the holder of the 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.) 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 anyone.

The `Powerless` interface can be used to assert that all instances of a user class are powerless. This assertion is checked at compile time by the Joe-E verifier, as described in Section 4.4.

An immutable object contains no mutable state and has no references to mutable state. A reference to an immutable object cannot be used to enact any state change visible to another entity that holds a reference of the same object. An immutable object conveys no power except for the unforgeable identity of any tokens it may contain. This potential form of power 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. The exceptions are tokens and objects containing tokens.

3.6 Selfless Types

A type T is *selfless* if and only if it implements the marker interface `org.joe_e.Selfless` in the overlay type system. The `org.joe_e.Selfless` interface must be provided by the Joe-E implementation and must declare a `hashCode()` method.

A Joe-E implementation may declare specific types from the standard Java libraries and from the Joe-E library to implement `org.joe_e.Selfless`. Such classes must not inherit Object's `hashCode()` method; it must be overriden either in the class itself or in some parent class. The behavior of an instance of the class must be verified (manually, in the case of classes deemed `Selfless`) to conceal object identity.

Rationale: An instance of a selfless type does not have visible object identity. Any two instances of the same selfless type with indistinguishable field values will be programmatically identical. A (shallow) clone of any selfless object will thus be indistinguishable from the original. One requirement for this condition to hold is that the selfless object be shallowly immutable: the fields of selfless objects cannot be allowed to change, otherwise the indistinguishability between instances is lost if one of them is changed. The objects pointed to by its reference-typed fields do not have this restriction, however, as any change made to them would be equally visible from all objects that reference them.

This is useful for ensuring that serialization and deserialization of objects is an identity operation. A selfless object can be serialized without having to worry about maintaining its identity. We anticipate that this will make serialization easier to perform using unprivileged code. The requirement that `hashCode()` be overridden is included to allow for hash codes to be taken when the concrete type for an object is not known, but it is known that the object is selfless.

The `Selfless` interface can be used to assert that all instances of a user class can be considered selfless. This assertion is checked at compile time by the Joe-E verifier, as described in Section 4.5. Also see the following section on `Equatable`.

3.7 Equatable Types

A type T is *equatable* if and only if it implements the marker interface `org.joe_e.Equatable` in the overlay type system. The (empty) `org.joe_e.Equatable` interface must be provided by the Joe-E implementation.

An instance of an equatable type allows comparison with another object using the `==` and `!=` operators. In the case of object types, this is pointer comparison, which is disallowed unless one of the objects being compared is equatable or null.

The following types are honorarily declared to implement `org.joe_e.Equatable`:

- the primitive scalar types (`boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, and `double`)
- all array types
- `java.lang.Enum`

Rationale: Primitive types are equatable because they have no object identity to expose; `==` simply compares their scalar values. Arrays are equatable because the `equals()` method on arrays already does a pointer-identity comparison. Enumerations, which are the only types that Java allows to extend `Enum`, are not constructed like ordinary objects; instead a fixed number of instances are constructed when the class is initialized and are made available to everyone as static members. `Enum`'s implementation of `equals()` is `final` and is equivalent to `==`, so making the class equatable is harmless, and only serves to make enumerations more convenient to use.

The `Equatable` interface can be used to allow pointer equality comparisons on objects of a type (See §4.9). This prevents the type or any of its subtypes from being declared selfless (§4.5).

4 Restrictions on Joe-E classes

4.1 Threads

For now, we are restricting Joe-E to the single-threaded subset of Java. This is reflected in taming decisions and the use of finalizers; see Sections 4.11 and 5. It is not necessary to prohibit the use of the `synchronized` keyword; since monitor locks are reentrant in Java, it simply becomes meaningless if no threads are created aside from the primary application thread.

Rationale: Unrestricted synchronization could allow a set of threads otherwise effectively isolated from each other to communicate, as every object becomes a communications channel through its associated lock. In the absence of such primitives, one can provide stricter guarantees of confinement, as immutable objects do not provide a communications channel. Any type of mutable shared state between threads allows observation of the nondeterminism inherent in thread scheduling, and would greatly complicate our efforts to make nondeterministic behavior require a capability.

4.2 Overlay Type System

Any class which implements an interface in the overlay type system must implement the type in the base type system if able to do so. For example, every user-defined exception type must implement `Powerless` in the base type system.

Rationale: This allows for efficiently determining whether an object implements a type in the overlay type system. The set of types that have honorary implements relationships is limited to types in the libraries, and traversal of type heirarchies is not necessary at runtime.

4.3 Static Fields

All static fields must be declared `final` and be of a powerless type. (Enumeration instances always satisfy this restriction, since java.lang.Enum is honorarily Powerless.)

Rationale: A reference to an object of any powerful type may convey authority. A mutable field conveys the authority to change its value. A public static variable in either of these categories provides ambient authority. While less obvious, this reasoning also applies to private static fields. In general, since any piece of code can create an object of the type in question, the object thus created has privileged access to the private static state. It is possible that this could be used to modify non-final static fields. Similarly, if the static field is of a powerful type, a newly-created object of the corresponding type could provide the authority to utilize this capability. This is a likely source of ambient authority which could be difficult to spot if non-powerless static fields were allowed. Fortunately, simple alternatives to the use of static fields nearly always exist.

4.4 Immutable and Powerless

If a class C implements `org.joe_e.Immutable` in the overlay type system, the following properties must hold:

1. every instance field f of C is both declared `final` and of an immutable 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 is immutable
3. any local variables in the scope of C that are observable by C are immutable.

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

If C is a library class that does not meet the requirements above, it can be manually deemed to be immutable. In this case, neither C nor any other library class may expose via its taming-allowed members any mutable state C may contain. Mutability must be hidden from subclasses if their creation is not prevented by the class being final or having no accessible constructor.

If a class C implements `org.joe_e.Powerless` in the overlay type system, the following properties must hold:

1. every instance field f of C is both declared `final` and of a 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 is powerless
3. any local variables in the scope of C that are observable by C are powerless
4. C is not a subclass of `org.joe_e.Token`.

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

If C is a library class that does not meet the requirements above, it can be manually deemed to be powerless. In this case, neither C nor any other library class may expose via its taming-allowed members any mutable state or tokens that C may contain. Mutability and tokens must be hidden from subclasses if their creation is not prevented by the class being final or having no accessible constructor.

The `org.joe_e.Token` class must be provided by the Joe-E implementation, and neither `org.joe_e.Token` itself nor its sole superclass `java.lang.Object` may implement `org.joe_e.Powerless`.

For the restrictions above, "every instance field of C" includes fields of any superclasses (whether accessible to C or not; this includes, for instance, all private fields of all superclasses).

A local variable v is *observable* by a class C if any of the following are true:

1. v is referenced by any code belonging to class C
2. an instance of D is constructed by any code in class C and v is observable to class D
3. D is the superclass of C and v is observable to class D

(This corresponds to the synthetic member fields the Java compiler must add to C in order to allow it, one of its superclasses, or another class constructed by it (possibly transitively) to make use of v.)

4.5 Selfless

The following restrictions are imposed on a class C that implements `Selfless`:

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

(b) C's superclass is `java.lang.Object`, C overrides `equals()`, and C doesn't call `super.equals()`.

For the first restriction above, "all instance fields of C" includes fields of any superclasses, whether accessible to C or not.

Note that no explicit checks are necessary to guarantee that a Selfless class provides a deterministic `hashCode()` implementation, just the general checks to ensure that interfaces are satisfied with methods that are not banned via taming (see §5)

4.6 Final Fields

Java guarantees that the final fields of an object are only uninitialized while the initialization of the object is taking place, and have all been initialized when the constructor exits. An object's initialization includes the execution non-static initializer blocks and initialization expressions of non-static fields; these are executed in lexical order before the body of hte invoked constructor is run, but after any superconstructor. Any expression within a constructor, non-static initializer, or non-static field initializer expression is executed during initialization.

The initialization process can call arbitrary methods during its execution, making its fields visible in an uninitialized state to those methods. In these methods (possibly from other classes) default values that may later change are visible for the object's final fields. This can allow for time-of-check-to-time-of-use attacks as the values of these fields can be observed to change. In order for the final qualifier on fields to ensure that the field value will not change, allowing Joe-E programs to trust in the finality of fields, Joe-E prevents any code other than the constructor itself from viewing an object in a partially-constructed state. This is accomplished by placing the following restrictions on instance initialization code, to ensure that a reference to the partially-constructed object cannot escape during its execution:

1. Instance initialization can't call any instance methods on the object being constructed. These are calls that resolve to instance methods (defined or inherited) that are invoked implicitly or explicitly on `this`. This includes supermethod invocations (those of the form `super.f()`).

2. Initialization cannot call the constructor of any non-static inner class of itself, i.e. any anonymous class or non-static member class that is defined within itself or any of its 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 can't make any references to the `this` pointer corresponding to 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.

4.7 Throwables

Note that making `java.lang.Throwable` a powerless type (see § 3.5) ensures that no `Throwable` can contain any powerful capabilities or any object of type `org.joe_e.Token` or any of its sub- or superclasses.

Rationale: Exceptions can implicitly communicate capabilities across security boundaries. This propagation can be hard to reason about, because the exceptional flow might not be immediately apparent in the source code. To see how this can cause unpleasant surprises, suppose Alice calls Bob. Bob has some special capability that she lacks, and Bob wants to avoid leaking this to her. At some point, Bob might need to invoke Chuck to perform some operation, passing this capability to Chuck. If (unbeknownst to Bob) Chuck can throw an exception that Bob doesn't catch, this exception might propagate to Alice. If this exception contains Bob's precious capability, this might cause Bob's capability to leak to Alice, against Bob's wishes. Example:

```
class E extends RuntimeException {
    public Object obj;
    public E(Object o) { obj = o; }
}
class Bob {
    // a capability, intended to be closely held
    private Capability cap;
    ...
    void m() {
        new Chuck().f(cap);
    }
}
class Chuck {
    void f(Capability cap) {
        ... do some work ...
        throw new E(cap);
    }
}
class Alice {
    void attack() {
        Bob bob = ...;
        try {
            bob.m();
        } catch (E e) {
            Capability stolencap = (Capability) e.obj;
            doSomethingEvil(stolencap);
        }
    }
}
```

The problem is that it is hard to tell, just by looking at the code of `Bob`, that Bob's private capability can leak to the caller of `m()`. This is a barrier to local reasoning about the flow of capabilities. By requiring that all throwables be powerless, we ensure that exceptions cannot convey authority or communicate capabilities across security boundaries.

4.8 Try-Catch-Finally Clauses

A `catch` clause is not allowed to specify a type of `java.lang.Throwable`, `java.lang.Error`, or any subtype of `java.lang.Error`. In addition, `finally` clauses are not permitted.

Rationale: These restrictions exist to deny Joe-E programs access to nondeterminism. Also, they provide a way to throw an exception that Joe-E code cannot catch.

If any source of nondeterminism is considered a capability, one can reason that any method called on an immutable object (provided it doesn't include an authority-bearing argument) will deterministically return the same result every time it is invoked. Unfortunately, the unmodified Java language provides ambient access to nondeterminism via virtual machine errors (subtypes of `java.lang.VirtualMachineError`), which can be caught by any class. In particular, a class can behave nondeterministically based on the amount of memory available by keeping track of how much memory it must allocate before it receives and recovers from an out-of-memory error. Alternately, one can determine the amount of stack space available as in the following example.

```
class IntException extends Exception implements Powerless {
    final int number;
```

```
    IntException(int n) { number = n; }
}
class Nondeterministic {
    static void f(int[] count) {
        count[0]++;
        f(count);
    }
    static void xx() throws IntException {
        int[] max = {0};
        try {
            f(max);
        } catch (StackOverflowError) {
            throw new IntException(max[0]);
        }
    }
    static int nondet() {
        try {
            xx();
            return 0;
        } catch (IntException ie) {
            return ie.number;
        }
    }
}
```

We take the approach that the core problem is that the nondeterministic virtual-machine errors such as `StackOverflowError` are visible to user code. If these errors were instead irrecoverable and caused immediate shutdown of the virtual machine, the program's view would be deterministic. Joe-E enforces the restrictions on exception handling listed above to prevent such errors from being recoverable by Joe-E code.

Finally clauses are forbidden because they can be used to "catch" any type of throwable and prevent the propagation of the original exception by throwing a new exception that masks it. For instance, in the example above, the `catch (StackOverflowError)` clause could be replaced by a `finally` clause without changing its behavior. This demonstrates that it is not enough to restrict the types of exceptions that can be caught using `catch` clauses; we must also restrict `finally` clauses as well. The cleanest solution we have been able to find involves forbidding `finally` clauses entirely.

In addition, if Joe-E code were able to catch errors, such as `StackOverflowError`, this would make it unreasonably difficult to build secure abstractions that can maintain their own object invariants in the face of hostile clients. For instance, a malicious client could arrange to use almost all available stack space, then invoke the object-under-attack. This might cause a `StackOverflowError` to be thrown at an unpredictable point in the code of the object-under-attack, possibly at a time when the object's invariants are (temporarily) violated. If the caller can catch this error and then make additional method invocations on the object-under-attack, the caller will be able to interact with the object after its object invariants have been invalidated. This may enable subtle, hard-to-predict attacks. It is unreasonable to expect programmers to write code in a way that defends against such attacks. Therefore, forbidding Joe-E code from catching errors makes it easier to preserve encapsulation and write code that can defend itself against hostile clients.

These rules suffice to deny Joe-E programs access to nondeterministic errors and also allow a Joe-E method to throw an exception that its invoker cannot catch. This is useful as it allows a way to guarantee that clients of a class cannot view inconsistent state, even if the class enters a condition from which it cannot recover to a consistent state. An `Error` that propagates to top level will by default halt the virtual machine, but in transactional Joe-E systems, it could roll back to a previous checkpoint.

Technically, the prohibition on `finally` clauses does not limit expressivity: there are several possible workarounds that allow one to achieve the same semantics or similar semantics, by transforming the code to

eliminate the `finally` keyword, though these transformations may increase code size or harm code clarity. There are several possible goals we might have for such a transformation:

1. No code duplication: The transformed version of the code should not require duplicating any of the original code (e.g., the code originally in the `finally` block).
2. No effect on callers: The transformed version of the code should not require us to modify the signature of the containing method—in particular, it should not require any changes to the list of exceptions thrown.
3. Equivalent semantics: The semantics of the transformed example should be exactly identical to the original code, except that if an error is thrown no Joe-E code should execute after the error is thrown.
4. Clarity: The transformed code should be as pleasant to read and write as possible. The transformed code should be as concise and clear as possible.

We list next several transformations that meet different subsets of these goals to varying degrees.

We describe first a general semantics-preserving transformation. For instance, suppose we have the following Java code:

```
void foo() {
   try {
      dosomething();
   } finally {
      cleanup();
   }
}
```

This code is illegal in Joe-E, but it can be transformed to comply with the Joe-E prohibition on `finally`, as follows:

```
void foo() {
  try {
    dosomething();
  } finally {
    cleanup();
  }
}
```

$\longrightarrow$

```
void foo() {
  RuntimeException e = null;
  try {
    dosomething();
  } catch (RuntimeException re) { e = re; }
  cleanup();
  if (e != null) { throw e; }
}
```

It is also possible to handle an arbitrary `try-catch-finally` block, by introducing a nested try block:

```
void foo() {
  try {
    dosomething();
  } catch (ParseException pe) {
    handle(pe);
  } finally {
    cleanup();
  }
}
```

$\longrightarrow$

```
void foo() {
  RuntimeException e = null;
  try {
    try {
      dosomething();
    } catch (ParseException pe) {
      handle(pe);
    }
  } catch (RuntimeException re) {
    e = re;
  }
  cleanup();
  if (e ! null) { throw e; }
}
```

The nested try block is unnecessary if the catch clause performs any needed cleanup itself (i.e., if the code in the finally block is redundant in this case). If the original try block might throw a checked exception, which also appears in the signature of the containing method, then we can add an extra catch clause to the transformed version, as follows:

```
void foo() throws IOException {
  InputStream in = ...;
  try {
    use(in);
  } finally {
    in.close();
  }
}
```

$\longrightarrow$

```
void foo() throws IOException {
  InputStream in = ...;
  Exception e = null;
  try {
    use(in);
  } catch (IOException ie) {
    e = ie;
  } catch (RuntimeException re) {
    e = re;
  }
  in.close();
  if (e instanceof IOException) {
    throw (IOException)e;
  } else if (e instanceof RuntimeException) {
    throw (RuntimeException)e;
  }
}
```

Of course, if the original code contains both a try block that might throw a checked exception (which appears in the signature of the containing method) and a catch clause for some other exception, it is possible to combine the two ideas listed above. This transformation preserves the type signature of the containing method and does not introduce any code duplication. This provides a way to handle arbitrary `try-catch-finally` sequences, transforming away the `finally` clause, while satisfying goals 1–3. We will let the reader judge whether the resulting code is sufficiently clear.

If goal 2 is considered unimportant, the above transformation can be simplified a bit:

```
void foo() throws IOException {
  InputStream in = ...;
  try {
    use(in);
  } finally {
    in.close();
  }
}
```

$\longrightarrow$

```
void foo() throws Exception {
  InputStream in = ...;
  Exception e = null;
  try {
    use(in);
  } catch (Exception e2) { e = e2; }
  in.close();
  if (e != null) { throw e; }
}
```

This transformation may be slightly clearer. It also achieves all of goals 1–3 in some special cases, e.g., where the try block can not throw any checked exception, or where the containing method's signature already contains a `throws Exception` clause.

Tyler Close has reported that he applied a related transformation when developing the Waterken server:

```
void foo() throws IOException {
  InputStream in = ...;
  try {
    use(in);
  } finally {
    in.close();
  }
}
```

⟶

```
void foo() throws Exception {
  InputStream in = ...;
  try {
    use(in);
  } catch (Exception e) {
    try { in.close(); }
    catch (Exception e2) {}
    throw e;
  }
  in.close();
}
```

This transformation makes no attempt to achieve goals 1 or 2: it duplicates the finally clause, and if the original method was declared to throw any checked exceptions, it changes the method signature by modifying the `throws` clause. It also does not preserve the exact semantics of the original code (goal 3). However, the transformed code on the right is possibly better than the original: if the finally clause (the `in.close()` operation) throws an exception of its own while it is executing, that may mask the original exception; the replacement code avoids masking the original exception. Tyler reports that, in his experience, it was not a problem to use the replacement code shown above, instead of using `finally` clauses.

In summary, there are a rich variety of workarounds that allow one to write useful code without using `finally` clauses. The existence of alternative coding patterns exist helped to support our decision to prohibit `finally` clauses.

We also made an attempt to evaluate how often `finally` clauses would be used in existing Java. We examined the 126 code samples found in *Effective Java*, a popular book on Java programming, and found that 2 of them used a `finally` clause. Both of these had a similar flavor to the `InputStream` examples above, where the code allocates a resource and then uses the `finally` clause to clean up after the resource is no longer used. They both also had the property that the `finally` clause is a single line and that there probably is no benefit to masking the original exception if the `finally` clause throws an exception—hence the transformation used in Waterken might be applicable to both of those examples as well.

4.9 Object Identity

The `==` and `!=` operators can only be applied to:

- two values of primitive type (this includes the case where one of these values is a boxed type that will be auto-unboxed);
- any object being compared with `null`; or
- two references, one or more of which is declared to be of a type that implements `org.joe_e.Equatable` in the overlay type system.

Any other use of `==` or `!=` is a compile-time error.

Rationale: The ability to uniquely identify an immutable object independent of the value that it contains can imbue an object that otherwise contains "just data" with a form of authority. For example, a locked box class can recognize whether it has been supplied the right key by keeping a private reference to the key object used only to test if a supplied key is the same one. An object used for this purpose must be an instance of `org.joe_e.Token` or one of its subclasses. A reference to an enumeration type does not convey any authority, not even by its identity, since all such objects are global, universally exported via static fields. For such values, `==` and it's "safer" possibly-selfless alternative, `equals()`, are equivalent.

4.10 Parameterized Types and Heap Pollution

The Java language only guarantees the safety of type parameters up to erasure. For backwards compatibility reasons, it is possible for a variable containing an instance of a parameterized type to have qualifiers that do

not match the runtime instance. This results from the erasure of the qualifiers at compile time and is known as "heap pollution". The consequence of this is that we must be cautious about trusing the syntactic type of expressions as inferred by the Java type checker when they get their type from a type parameter.

For checks as to whether an instance field implements a marker interface such as Powerless, the erased type of the field is used. In particular, this means that a Joe-E Powerless class with an instance field of parameterized type must declare the type parameter with `Powerless` as its first type bound. (Subsequent type bounds are not reflected in the erased type of the field.)

For explicit method invocations, expressions whose type derives from a type parameter do not require special handling because the call is compiled to reference the method determined by the expression's static type.

The implicit method call resulting from string conversion, however, causes problems. The Java Language spec is vague about how string conversion works; it only says that it is "as if by an invocation of the `toString` method of the referenced object" but does not specify what happens if the type of the object is not as expected. Both compilers tested treated this case as different from an explicit invocation of `toString()` on the expression; the call succeeded without error even if the actual type of the object was not as expected. (This would be consistent with compiling string conversion to always invoke `toString()` on `Object`; equivalent to the source construction `((Object) x).toString()` for reference types.) In order to prevent this anomaly providing a way around taming policy for `toString()`, the verifier uses the erased type of expressions subject to implicit string conversion when checking the implicit `toString()` call. A verification failures resulting from this can be fixed by inserting an explicit cast to the expected type of the expression.

It is unclear exactly when a particular expression may have the wrong type due to heap pollution. While it would be easy to just insert an implicit cast to the expected type as soon as a value of a parameterized type occurs in an expression with an inferred concrete type, it appears that compiler implementations instead do nothing to ameliorate the effects of heap pollution and instead rely on the error being caught by the runtime checks of the JVM. These appear to occur when the value is assigned to another variable, either implicitly or explicitly. It appears that a ?: expression always results in a temporary variable used to store the result, which is typed to correspond with the expected type of the expression. This means that only a type parameter literal, a field whose type is a type parameter, or a return value that is a type parameter might be corrupted; though it may be surrounded by any number of parentheses. When checking string conversions, the Joe-E verifier currently assumes that the type-checker-inferred type is correct for all other cases. Unfortunately, it is unclear whether this assumption is guaranteed to hold.

The enhanced for loop does not have this problem, as it is defined to be exactly equivalent to an idiom that invokes `iterator()` on the collection expression without any casts, which will ensure that the invocation will fail with a class cast exception if the object is of a different type than that inferred by the type checker.

4.11 Finalizers

Custom finalizers (methods that override `java.lang.Object`'s default `finalize()` method) are not allowed.

Rationale: When the garbage collector invokes `finalize()` methods, it does so at unpredictable times and likely in a separate Java thread. Allowing user code to be executed in this manner would provide opportunities for nondeterminism and would violate reasoning based on single-thread semantics for Joe-E programs.

Also, `finalize()` can be used to get access to an incompletely-initialized object instance that should not be available. For instance, consider:

```
public class Uninstantiable {
    public Uninstantiable() {
        throw new SecurityException("not allowed");
    }
}
```

It should be impossible to obtain a reference to an `Uninstantiable` object. Unfortunately, `finalize()` allows an attacker to gain access to such an object:

```
public class Thief extends Uninstantiable {
    public static Uninstantiable u = null;
    public void finalize() {
        this.u = this;
    }
    public static Uninstantiable make() {
        new Thief();
        // ... wait a while for GC and finalizers to execute
        return u;
    }
}
```

The ability to gain access to an object whose constructor has completed unsuccessfully is surprising. It also violates the semantics of `final` fields. Consider this variation:

```
public class Uninstantiable2 {
    public final int i;
    public Uninstantiable2() {
        die();
        i = 42;
    }
    private static void die() {
        throw new SecurityException("not allowed");
    }
}
```

Now an attacker can use the same trick to gain access to an incompletely-constructed `Uninstantiable` instance whose field `i` has the value 0.

Custom finalizers are thus prohibited.

4.12 Serialization

Custom serialization behavior of Joe-E classes (methods with signatures `readObject(java.io.ObjectInputStream)` or `writeObject(java.io.ObjectOutputStream)` is similarly prohibited.

Rationale: Serialization methods are expected to have properties that are not verified and which are difficult to verify automatically. The default serialization behavior provides that the reconstituted object's field values are equivalent to the original, but this equivalence can be violated with custom serialization behavior. This could allow an object, for example, to "remember" when it is serialized. In order to support the development of persistence mechanisms that provide a stricter serialization consistency guarantee, for example ensuring an object's behavior is independent of whether it has been serialized and revived, Joe-E code must be prevented from writing its own serialization methods.

4.13 Native Methods

Native methods are forbidden.

Rationale: Native methods bypass the memory- and type-safety checks of the Java language that are necessary to ensure the unforgeability of capabilities. Therefore, they are forbidden.

4.14 Library Protection

Optional: A Joe-E implementation may impose restrictions to ensure that Joe-E user code does not declare itself to be a member of a library package. These checks are required if any user code is loaded using the same classloader as any library code not protected by the classloader (e.g. by package sealing).

These checks are not required for a standard Java environment, at least not for Java classpath code. The standard classpath libraries are loaded by the native primordial class loader, while all user code is loaded with a Java-language class loader that extends `java.lang.ClassLoader`. These checks may be necessitated if a Joe-E implementation uses the same classloader for Joe-E user code and the Joe-E library.

5 Taming

Code that is written entirely in Joe-E in accordance with the rules above is verifiably capability-safe. While the ability to use Java syntax and the Java toolchain would add some value even if all code must be written from the ground up in Joe-E, this is not enough. Much of the utility of the Java language comes from its libraries. Technically, it's not possible to write a Java program at all without using at least one library class: `java.lang.Object`.

Unfortunately, many library classes (including `Object`) were not designed with capability-security in mind, and so contain some methods that expose nondeterminism or allow access to privileged operations without requiring a capability. Allowing indiscriminate use of the Java class library can thus subvert some of Joe-E's linguistic restrictions. Especially problematic are reflective facilities that could violate the intended security discipline of classes defined in Joe-E.

5.1 Policy

Since the library can't (and shouldn't) be completely prohibited, nor can it all be allowed, we use a *taming* mechanism to allow some library facilities and disable others. This takes the form of a whitelist of classes and class members (library methods and fields) that are allowed to be named and invoked by Joe-E code. Any class or method not in the whitelist cannot be referenced; any Joe-E program should be able to compile against a hypothetical version of the library in which the method did not exist. The taming-related checks in the verifier, detailed below, ensure that no method or field that is "tamed away" can be directly invoked by Joe-E code. Note that other methods in the Java or Joe-E library may be allowed to access the member in question, by a mechanism yet to be formally specified but roughly analagous to deeming for marker interfaces. This essentially makes it a separate policy decision to determine whether their access is safe. It could be that they only provide a safe subset of the functionality that would be exposed by enabling the member directly. An example of this are wrappers provided in the Joe-E library which provide a mediated, capability-compatible interface to underlying Java library classes that are not capability-safe.

A comprehensive approach to philosophy and specific policies for taming the Java class libraries is forthcoming. Some notes follow.

Nondeterminism means that `java.lang.Object`'s `hashCode()` and and `toString()` implementations must be disabled. `Selfless` indicates that a class has a safe `hashCode()` method, but we may also want a marker interface (or a reflective method?) to allow for printing out object's string representations for objects that are safe to convert to strings.

We will need to tame `java.lang.Throwable` to ensure it meets the semantics of a powerless class, since Section 3.5 requires it to be honorarily powerless. We may also need to tame some existing exception classes defined in the standard Java libraries and then explicitly deem them to implement `org.joe_e.Powerless`, so that their behavior will indeed be consistent with what one would expect of a powerless type (since all throwables are required to be powerless).

A number of invariants must hold in the taming decisions in order for them to be consistent and enforceable.

- If a class is disabled, all subtypes of that class must also be disabled. Otherwise the "tamed" type hierarchy has holes in it, which may cause problems.
- If a method is enabled, all methods that override that method must also be enabled. Otherwise one could simply upcast to the supertype and call the method anyway. This is a result of the static nature of the taming enforcement; if the method dispatch was mediated dynamically (as it is in E-on-Java), a policy that allows a method in the superclass but not in the subclass would in fact be enforceable. Note that this requirement is of particular concern for non-final methods of class `Object`: if any class provides a capability-unsafe version of these methods, we must either disable the methods from `Object` or ensure (through other taming decisions) that no instance of that class is ever obtainable by Joe-E code.

Any Serializable library class must be tamed such that the obervable behavior of a serialized and revived instance is indistinguishable from the original with the possible exception of object identity.

Classes implementing the Iterable interface as well as non-final classes providing methods named `next()` or `hasNext()` raise special issues for taming. If the implementation provided of either of these methods is determined to be unsafe, Joe-E code must be prevented from ever obtaining an instance of the class, as it could otherwise declare an `iterator()` method that returns it.

Classes that call hashCode(), toString(), or Class.getName() should be carefully examined to ensure that their behavior does not visibly depend on the result of these method calls, which can be nondeterministic.

Any method providing information about real time must be suppressed or require a capability.

Methods dependent on the current locale should be suppressed.

The behavior of a program can be dependent on the current version of Java (or at least any things that change between versions, e.g. the set of characters defined by the Unicode standard), the current taming decisions, whether assertions are enabled. Policy may change on these if there is a good reason.

5.2 Mechanism

Taming policy is enforced by ensuring that no identifier in the program resolves to a type or member that is disabled. Potentially disabled class types can appear in:

1. the formal argument types of a method or constructor
2. the formal bounds of a type variable
3. the actual type arguments to a generic type, method, or constructor
4. class instance creation expressions, including anonymous class definitions.
5. `extends` and `throws` clauses
6. field and local variable declarations
7. parameters to `catch` clauses
8. class import declarations
9. return types
10. cast expressions

Potentially disabled members (fields, methods, and constructors) can appear in:

1. field and superfield accesses
2. method and supermethod invocations
3. superconstructor invocations

4. class instance creation expressions, including anonymous class definitions.

5. static import declarations

The last three of the locations for disabled classes are benign, as creating an instance of the class requires the use of a constructor, all of which are disabled for a disabled class. We don't see any reason to allow these uses, however, so it is simpler to prohibit any mention of the type of a disabled class. This approach likely to avoid confusion by consistently indicating that the type is disabled. Static import declarations of disabled fields and methods could also be allowed without violating soundness, but are prohibited for the same reason.

In addition to explicit expressions that resolve to a disabled member, it is also necessary to check for implicit invocation of disabled members in the following circumstances:

1. implicit superconstructor invocations, either as the default superconstructor for a constructor that does not specify one, or if no explicit constructor is provided.

2. implicit calls to `toString()` resulting from string conversion in assertion expressions and when evaluating `+` and `+=` expressions.

3. implicit calls to `iterator()` resulting from the use of enhanced for loops. The verifier does *not* check the implicit `next` and `hasNext()` calls performed on the iterator itself, since the concrete type of the iterator is not available to the verifier. This has the unfortunate effect of requiring special attention be made when taming if any class provides unsafe versions of these methods. Joe-E code must be prevented from obtaining a reference to any such class, as it could otherwise declare an `iterator()` method that returns it. This open-ended requirement is tractable because (a) most (possibly all) iterators are safe and (b) any unsafe iterator is likely to be obtainable in a limited number of ways, all of which can be disabled.

Another wrinkle with taming verification is that Java's interface implementation checks are unaware of taming decisions. This can be a problem when one calls the method on an object that is known only to belong to the interface. In this case, the verifier is only able to resolve the method to the interface rather than a concrete implementation, and so it will be unable to tell if the concrete implementation is supposed to be tamed away. In order to prevent a loss of soundness in this case while still allowing methods to be called in this manner, Joe-E requires that classes only implement interfaces consistent with taming decisions. Specifically, it requires that each method defined in an interface implemented by a class resolves to a concrete implementation that isn't disabled by taming.

6 Malicious bytecode

This specification requires that Joe-E programs be written in source form, compiled using a Java compiler, passed through the Joe-E verifier, and then executed with a JVM. In particular, we require that every classfile be produced by a correctly operating Java compiler; no part of a Joe-E program is permitted to contain bytecode obtained from untrusted sources or generated by hand.

Rationale: Java compilers perform many checks that are not duplicated by the JVM. These checks include[1]: exception safety; access control for inner classes; isolation of user code from classpath code; in-range checks for short types. Since these checks are performed only by the compiler, malicious bytecode can evade these checks. This means that if any part of the program includes bytecode not produced by a legitimate Java compiler, Joe-E programmers will be unable to rely upon exception safety, access control for inner classes, etc.

In principle, one could duplicate all these checks in the Joe-E verifier, and then raw bytecode would not need to be forbidden since programmers could rely upon these language features even in the presence of malicious bytecode. However, the benefit of adding this into the verifier seems to be questionable given the costs, so we omit this (for now, anyway).

[1] See Appendix A for further discussion.

7 Work in Progress

Further documentation will be forthcoming about the taming framework and philosophy as it is developed, as well as documentation on taming decisions.

This specification currently does not prevent the reading of an uninitialized value for a static final field; this protection may be provided in the future.

A Issues with malicious bytecode

If we were to accept (possibly maliciously constructed) bytecode from an unknown source and link it into our program (after checking that it is accepted by the Joe-E verifier), there would be many pitfalls to worry about. These include:

- Java's visibility modifiers (private, protected, etc.) can be subverted in the presence of inner classes. The Java compiler inserts synthetic getter/setter methods; while the compiler checks that Java source cannot use these synthetic methods, the JVM does not, so malicious bytecode could exploit them to to gain access to private state. Consequently, Java programmers cannot count on visibility modifiers on inner classes, in the presence of malicious bytecode.

  Joe-E code compiled from source can ensure that these protections are enforced. A more sophisticated implementation of the Joe-E verifier would be able to make the same guarantees for arbitrary bytecode. Extension of the verifier to handle arbitrary bytecode is nontrivial but quite feasible, and may be considered for a version subsequent to the first prototype.

- Exception-safety can be subverted in the presence of malicious code. The Java compiler checks a property that one might call *exception-safety*: if the program calls a method `m()`, and if `m()` terminates abruptly, it can only do so with an exception that is (1) (a subclass of) a class declared in `m()`'s clause; (2) (a subclass of) a `RuntimeException`; or, (3) (a subclass of) an `Error`. This property is enforced only by the Java compiler, and not by the JVM. Consequently, maliciously constructed Java bytecode can throw checked exceptions not declared in its `throws` clause.

  Just as in Java, Joe-E programmers cannot rely on exception-safety in the presence of malicious bytecode. While it would be possible to re-implement the exception-safety checks in the Joe-E verifier, it seems this would require more effort than it is worth.

- Final fields can be assigned to multiple times, in the presence of malicious bytecode. The Java compiler performs so-called "definite assignment" checks to ensure that every final field is assigned to exactly once by the time the constructor exits, and is never assigned after that point. This check is not necessarily duplicated by the JVM; consequently, malicious bytecode might be able to modify `final` fields, subvert our immutability analysis, and potentially violate other security properties.

- Since generic types are implemented by erasure, malicious bytecode can partially subvert type-checking for type parameters. The correct use of type parameters is only checked at compile time, and then is erased. Consequently, though the JVM does check basic type system, the JVM does not and cannot check that type parameters are used correctly. Further details: `http://portal.acm.org/citation.cfm?id=997144` (§5).

- It seems the intuitive range limits for short types can possibly be violated by malicious bytecode. Consider the following method:

```
boolean isByte(byte b) {
    return (-128 <= bb) && (bb <= 127);
}
```

One might naively expect that `isByte()` can only return true. However, it's not clear that this is actually guaranteed, in the presence of malicious bytecode. The JVM stores `byte` values in 32-bit registers, and handles them almost exactly the same way as `int` values. No range checks are performed. It may be possible for malicious bytecode to pass a full 32-bit value to `isByte()`, causing the method to return false. Similar risks may apply to `boolean`, `byte`, `short`, and `char`. We have not yet evaluated this risk. References: `http://groups.yahoo.com/group/java-spec-report/message/756?threaded=1` `http://archives.java.sun.com/cgi-bin/wa?A2=ind9802&L=java-security&P=2691`

- It might be possible to subvert some of the access checks for `protected` members. Let `x` be a protected non-static field of class `C`, and let `S` be a subclass of `C` from a different package. The Java compiler enforces a special check that `S` can access `obj.x` only if `obj`'s class is in the inheritance subtree rooted at `S` (and not, for instance, if `obj` is of class `C`). This check is a little bit tricky in the presence of overriding, and there have been reports that malicious bytecode can bypass this check. We haven't investigated this in any detail. `http://www.kestrel.edu/home/people/coglio/ftjp04.pdf`

- There may also be some potential danger spots with monitors. What if malicious bytecode fails to follow a stack discipline with its `monitorenter`/`monitorexit` instructions? What if the bytecode fails to call `monitorexit` before returning? What a security-critical library acquires a lock and calls our malicious bytecode, which then uses `monitorexit` to release the library's lock? It is unclear what may be possible here.

- The `ACC_SUPER` bit may allow to subvert the semantics of overridden methods. Suppose class `C` defines a method `m()`, which is overridden by some subclass `S`. With current Java semantics, holding a reference to `S`, you aren't supposed to be able to invoke the body of `C.m()` on this object. However, malicious bytecode could reset the `ACC_SUPER` bit in its classfile and get access to the hidden method, so that it can get an instance of `S` to execute the code specified in `C.m()`. If `S` was relying on overriding to prevent access to `C`'s implementation of `m()`, then malicious bytecode could falsify this assumption.

- The Java language requires that the first line of any construct must either call the superclass constructor or must call some other constructor for the same class. The Java compiler will insert an implicit call to the superclass constructor if this rule is not followed. In contrast, the JVML bytecode verifier does not check this rule, so a constructor written in JVML bytecode could violate this rule.

We make no claims that this is the complete list of pitfalls associated with malicious bytecode. The simplest way to avoid these pitfalls is to compile everything from source using a trusted Java compiler and refrain from accepting raw bytecode from unknown sources; this is exactly what we require when writing Joe-E programs.