Enforcing Secure Object Initialization in Java

Laurent Hubert[1], Thomas Jensen[2], Vincent Monfort[2], and David Pichardie[2]

[1] CNRS/IRISA, France
[2] INRIA Rennes - Bretagne Atlantique/IRISA, France

Abstract. Sun and the CERT recommend for secure Java development to *not allow partially initialized objects to be accessed*. The CERT considers the severity of the risks taken by not following this recommendation as *high*. The solution currently used to enforce object initialization is to implement a coding pattern proposed by Sun, which is not formally checked. We propose a modular type system to formally specify the initialization policy of libraries or programs and a type checker to statically check at load time that all loaded classes respect the policy. This allows to prove the absence of bugs which have allowed some famous privilege escalations in Java. Our experimental results show that our safe default policy allows to prove 91% of classes of `java.lang`, `java.security` and `javax.security` safe without any annotation and by adding 57 simple annotations we proved all classes but four safe. The type system and its soundness theorem have been formalized and machine checked using Coq.

1 Introduction

The initialization of an information system is usually a critical phase where essential defense mechanisms are being installed and a coherent state is being set up. In object-oriented software, granting access to partially initialized objects is consequently a delicate operation that should be avoided or at least closely monitored. Indeed, the CERT recommendation for secure Java development [2] clearly requires to *not allow partially initialized objects to be accessed* (guideline OBJ04-J). The CERT has assessed the risk if this recommendation is not followed and has considered the severity as *high* and the likelihood as *probable*. They consider this recommendation as a first priority on a scale of three levels.

The Java language and the Java Byte Code Verifier (BCV) enforce some properties on object initialization, *e.g.* about the order in which constructors of an object may be executed, but they do not directly enforce the CERT recommendation. Instead, Sun provides a guideline that enforces the recommendation. Conversely, failing to apply this guidelines may silently lead to security breaches. In fact, a famous attack [4] used a partially initialized class loader for privilege elevation.

We propose a twofold solution: (i) a modular type system which allows to express the initialization policy of a library or program, *i.e.* which methods may access partially initialized objects and which may not; and (ii) a type checker,

which can be integrated into the BCV, to statically check the program at load time. To validate our approach, we have *formalized* our type system, *machine checked* its soundness proof using the Coq proof assistant, and *experimentally validated* our solution on a large number of classes from Sun's Java Runtime Environment (JRE).

Section 3 overviews object initialization in Java and its impacts on security. Section 4 then informally presents our type system, which is then formally described in Section 5. Section 6 finally presents the experimental results we obtained on Sun's JRE.

2 Related Work

Object initialization has been studied from different points of view. Freund and Mitchell [7] have proposed a type system that formalizes and enforces the initialization properties ensured by the BCV, which are not sufficient to ensure that no partially initialized object is accessed. Unlike local variables, instance fields have a default value (**`null`**, **`false`** or `0`) which may be then replaced by the program. The challenge is then to check that the default value has been replaced before the first access to the field (*e.g.* to ensure that all field reads return a non-null value). This is has been studied in its general form by Fähndrich and Xia [6], and Qi and Myers [9]. Those works are focused on enforcing invariants on fields and finely tracks the different fields of an object. They also try to follow the objects after their construction to have more information on initialized fields. This is an overkill in our context. Unkel and Lam studied another property of object initialization: stationary fields [12]. A field may be stationary if all its reads return the same value. There analysis also track fields of objects and not the different initialization of an object. In contrast to our analysis, they stop to track any object stored into the heap.
Other work have targeted the order in which methods are called. It has been studied in the context of rare events (*e.g.* to detect anomaly, including intrusions). We refer the interested reader to the survey of Chandola *et al.* [3]. They are mainly interested in the order in which methods are called but not about the initialization status of arguments. While we guarantee that a method taking a fully initialized receiver is called after its constructor, this policy cannot be locally expressed with an order on method calls as the methods (constructors) which needs to be called on a object to initialize it depends on the dynamic type of the object.

3 Context Overview

Fig. 1 is an extract of class `ClassLoader` of SUN's JRE as it was before 1997. The security policy which needs to be ensured is that `resolveClass`, a security sensitive method, may be called only if the security check l. 5 has succeeded. To ensure this security property, this code relies on the properties enforced on object initialization by the BCV.

```
1 public abstract class ClassLoader {
2   private ClassLoader parent;
3   protected ClassLoader() {
4     SecurityManager sm = System.getSecurityManager();
5     if (sm != null) {sm.checkCreateClassLoader();}
6     this.parent = ClassLoader.getSystemClassLoader();
7   }
8   protected final native void resolveClass(Class c);
9 }
```

Fig. 1. Extract of the ClassLoader of Sun's JRE

Standard Java Object Construction. In Java, objects are initialized by calling a class-specific constructor which is supposed to establish an invariant on the newly created object. The BCV enforces two properties related to these constructors. These two properties are necessary but, as we shall see, not completely sufficient to avoid security problems due to object initialization.

Property 1. Before accessing an object, (i) a constructor of its dynamic type has been called and (ii) each constructor either calls another constructor of the same class or a constructor of the super-class on the object under construction, except for `java.lang.Object` *which has no super-class.*

This implies that at least one constructor of C and of each super-class of C is called: it is not possible to bypass a level of constructor. To deal with exceptional behaviour during object construction, the BCV enforces another property — concisely described in *The Java Language Specification* [8], Section 12.5, or implied by the type system described in the JSR202 [1]).

Property 2. If one constructor finishes abruptly, then the whole construction of the object finishes abruptly.

Thus, if the construction of an object finishes normally, then all constructors called on this object have finished normally. Failure to implement this verification properly led to a famous attack [4] in which it was exploited that if code such as `try {super();} catch(Throwable e){}` in a constructor is not rejected by the BCV, then malicious classes can create security-critical classes such as class loaders.

Attack on the Class Loader and the Patch from Sun. However, even with these two properties enforced, it is not guaranteed that uninitialized objects cannot be used. In Fig. 1, if the check fails, the method `checkCreateClassLoader` throws an exception and therefore terminates the construction of the object, but the garbage collector then call a `finalize()` method, which is an instance method and has the object to be collected as receiver (cf. Section 12.6 of [8]).

An attacker could code another class that extends `ClassLoader` and has a `finalize()` method. If run in a right-restricted context, *e.g.* an applet, the constructor of `ClassLoader` fails and the garbage collector then call the attacker's

```
1  public abstract class ClassLoader {
2    private volatile boolean initialized;
3    private ClassLoader parent;
4    protected ClassLoader() {
5      SecurityManager sm = System.getSecurityManager();
6      if (sm != null) {sm.checkCreateClassLoader();}
7      this.parent = ClassLoader.getSystemClassLoader();
8      this.initialized = true;}
9    private void check() {
10     if (!initialized) {
11       throw new SecurityException(
12         "ClassLoader␣object␣not␣initialized");}}
13   protected final void resolveClass(Class c){
14     this.check();
15     this.resolveClass0(c);}
16   private native void resolveClass0(Class c);
17 }
```

Fig. 2. Extract of the ClassLoader of Sun's JRE

`finalize` method. The attacker can therefore call the `resolveClass` method on it, bypassing the security check in the constructor and breaking the security of Java.

The initialization policy enforced the BCV is in fact too weak: when a method is called on an object, there is no guarantee that the construction of an object has been successfully run. An ad-hoc solution to this problem is proposed by SUN [11] in its Guideline 4-3 *Defend against partially initialized instances of non-final classes*: adding a special Boolean field to each class for which the developer wants to ensure it has been sufficiently initialized. This field, set to **`false`** by default, should be private and should be set to **`true`** at the end of the constructor. Then, every method that relies on the invariant established by the constructor must test whether this field is set to **`true`** and fail otherwise. If `initialized` is **`true`**, the construction of the object up to the initialization of `initialized` has succeeded. Checking if `initialized` is **`true`** allows to ensure that sensitive code is only executed on classes that have been initialized up to the constructor of the current class. Fig. 2 shows the same extract as in Fig. 1 but with the needed instrumentation (this is the current implementation as of JRE 1.6.0_16).

Although there are some exceptions and some methods are designed to access partially initialized objects (for example to initialize the object), most methods should not access partially initialized objects. Following the remediation solution proposed in the CERT's recommendation or Sun's guideline 4-3, a field should be added to almost every class and most methods should start by checking this field. This is resource consuming and error prone because it relies on the programmer to keep track of what is the semantic invariant, without providing the adequate automated software development tools. It may therefore lead not to

functional bugs but to security breaches, which are harder to detect. In spite of being known since 1997, this pattern is not always correctly applied to all places where it should be. This has lead to security breaches, see *e.g.*, the Secunia Advisory SA10056 [10].

4 The Right Way: A Type System

We propose a twofold solution: first, a way to specify the security policy which is simple and modular, yet more expressive than a single Boolean field; second, a modular type checker, which could be integrated into the BCV, to check that the whole program respects the policy.

4.1 Specifying an Initialization Policy with Annotations

We rely on Java annotations and on one instruction to specify our initialization policy. We herein give the grammar of the annotations we use.

```
V_ANNOT ::= @Init | @Raw | @Raw(CLASS)
R_ANNOT ::= @Pre(V_ANNOT) | @Post(V_ANNOT)
```

We introduce two main annotations: `@Init`, which specifies that a reference can only point to a fully initialized object or the **`null`** constant, and `@Raw`, which specifies that a reference may point to a partially initialized object. A third annotation, `@Raw(CLASS)`, allows to precise that the object may be partially initialized but that all constructors up to and including the constructor of `CLASS` must have been fully executed. *E.g.*, when one checks that `initialized` contains **`true`** in `ClassLoader.resolveClass`, one checks that the receiver has the type `@Raw(ClassLoader)`. The annotations produced by the `V_ANNOT` rule are used for fields, method arguments and return values. In the Java language, instance methods implicitly take another argument: a receiver — reachable through variable **`this`**. We introduce a `@Pre` annotation to specify the type of the receiver at the beginning of the method. Some methods, usually called from constructors, are meant to initialize their receiver. We have therefore added the possibility to express this by adding a `@Post` annotation for the type of the receiver at the end of the method. These annotations take as argument an initialization level produced by the rule `V_ANNOT`.

Fig. 3 shows an example of `@Raw` annotations. Class `Ex1A` has an instance field `f`, a constructor and a getter `getF`. This getter requires the object to be initialized at least up to `Ex1A` as it accesses a field initialized in its constructor. The constructor of `Ex1B` uses this getter, but the object is not yet completely initialized: it has the type `Raw(Ex1A)` as it has finished the constructor of `Ex1A` but not yet the constructor `Ex1B`. If the getter had been annotated with `@Init` it would not have been possible to use it in the constructor of `Ex1B`.

Another part of the security policy is the `SetInit` instruction, which mimics the instruction **`this`**`.initialized = `**`true`** in Sun's guideline. It is implicitly put at the end of every constructor but it can be explicitly placed before. It

```
1 class Ex1A {                          9 class Ex1B extends Ex1A{
2     private Object f;                10     Ex1B(){
3     Ex1A(Object o){                  11         super();
4         securityManagerCheck()       12         ... = this.getF();
5         this.f = o;}                 13     }
6     @Pre(@Raw(Ex1A))                 14 }
7     getF(){return this.f;}
8 }
```

Fig. 3. Motivations for `Raw(CLASS)` annotations

```
1 public C() {
2   ...
3   securityManagerCheck(); // perform dynamic security checks
4   SetInit;                // declare the object initialized up C
5   Global.register(this);  // the object is used with a method
6 }                         // that only accept Raw(C) parameters
```

Fig. 4. An Example with `SetInit`

declares that the current object has completed its initialization up to the current class. Note that the object is not yet considered fully initialized as it might be called as a parent constructor in a subclass. The instruction can be used, as in Fig.4, in a constructor after checking some properties and before calling some other method.

Fig. 5 shows class `ClassLoader` with its policy specification. The policy ensured by the current implementation of Sun is slightly weaker: it does not ensure that the receiver is fully initialized when invoking `resolveClass` but simply checks that the constructor of `ClassLoader` has been fully run. On this example, we can see that the constructor has the annotations `@Pre(@Raw)`, meaning that the receiver may be completely uninitialized at the beginning, and `@Post(@Raw(ClassLoader))`, meaning that, on normal return of the method, at least one constructor for each parent class of `ClassLoader` and a constructor of `ClassLoader` have been fully executed.

We define as default values the most precise type that may be use in each context. This gives a *safe by default* policy and lowers the burden of annotating a program.

- Fields, method parameters and return values are fully initialized objects (written `@Init`).
- Constructors take a receivers uninitialized at the beginning (`@Pre(@Raw)`) and initialized up-to the current class at the end (written `@Post(@Raw(C))` if in the class `C`).
- Other methods take a receiver fully initialized (`@Pre(@Init)`).
- Except for constructors, method receivers have the same type at the end as at beginning of the method (written `@Post(A)` if the method has the annotation `@Pre(A)`).

```
1  public abstract class ClassLoader {
2    @Init private ClassLoader parent;
3    @Pre(@Raw) @Post(@Raw(ClassLoader))
4    protected ClassLoader() {
5      SecurityManager sm = System.getSecurityManager();
6      if (sm != null) {sm.checkCreateClassLoader();}
7      this.parent = ClassLoader.getSystemClassLoader();
8    }
9    @Pre(@Init) @Post(@Init)
10   protected final native void resolveClass(@Init Class c);
11 }
```

Fig. 5. Extract of the ClassLoader of Sun's JRE

If we remove from Fig. 5 the default annotations, we obtain the original code in Fig. 1. It shows that despite choosing the strictest (and safest) initialization policy as default, the annotation burden can be kept low.

4.2 Checking the Initialization Policy

We have chosen static type checking for at least two reasons. Static type checking allows for more performances (except for some rare cases), as the complexity of static type checking is linear in the *code size*, whereas the complexity of dynamic type checking is linear in the *execution time*. Static type checking also improves reliability of the code: if a code passes the type checking, then the code is correct with respect to its policy, whereas the dynamic type checking only ensures the correction of a particular execution.

Reflection in Java allows to retrieve code from the network or to dynamically generates code. Thus, the whole code may not be available before actually executing the program. Instead, code is made available class by class, and checked by the BCV at linking time, before the first execution of each method. As the whole program is not available, the type checking must be modular: there must be enough information in a method to decide if this method is correct and, if an incorrect method is found, there must exist a safe procedure to end the program (usually throwing an exception), *i.e.* it must not be too late.

To a have a modular type checker while keeping our security policy simple, method parameters, respectively return values, need to be contra-variant, respectively co-variant, *i.e.* the policy of the overriding methods needs to be at least as general as the policy of the overridden method. Note that this is not surprising: the same applies in the Java language (although Java imposes the invariance of method parameters instead of the more general contra-variance), and when a method call is found in a method, it allows to rely on the policy of the resolved method (as all the method which may actually be called cannot be known before the whole program is loaded).

$$
\begin{array}{rl}
 & x, y, r \in \mathit{Var} \quad f \in \mathit{Field} \quad e \in \mathit{Exc} \quad i \in \mathcal{L} = \mathbb{N} \\
p \in \mathit{Prog} ::= & \{\, \mathsf{classes} \in \mathcal{P}(\mathit{Class}),\ \mathsf{main} \in \mathit{Class}, \\
 & \quad \mathsf{fields} \in \mathit{Field} \to \mathit{Type},\ \mathsf{lookup} \in \mathit{Class} \to \mathit{Meth} \rightharpoonup \mathit{Meth}\} \\
c \in \mathit{Class} ::= & \{\mathsf{super} \in \mathit{Class}_\perp,\ \mathsf{methods} \in \mathcal{P}(\mathit{Meth}),\ \mathsf{init} \in \mathit{Meth}\} \\
m \in \mathit{Meth} ::= & \{\, \mathsf{instrs} \in \mathit{Instr}\ \mathit{array},\ \mathsf{handler} \in \mathcal{L} \to \mathit{Exc} \to \mathcal{L}_\perp, \\
 & \quad \mathsf{pre} \in \mathit{Type},\ \mathsf{post} \in \mathit{Type},\ \mathsf{argtype} \in \mathit{Type},\ \mathsf{rettype} \in \mathit{Type}\} \\
\tau \in \mathit{Type} ::= & \mathit{Init} \mid \mathit{Raw}(c) \mid \mathit{Raw}^\perp \\
e \in \mathit{Expr} ::= & \mathit{null} \mid x \mid e.f \\
\mathit{ins} \in \mathit{Instr} ::= & x \leftarrow e \mid x.f \leftarrow y \mid x \leftarrow \mathit{new}\ c(y) \mid \mathit{if}\ (\star)\ \mathit{jmp} \mid \\
 & \mathit{super}(y) \mid x \leftarrow r.m(y) \mid \mathit{return}\ x \mid \mathit{SetInit}
\end{array}
$$

Fig. 6. Language Syntax.

5 Formal Study of the Type System

The purpose of this work is to provide a type system that enforces at load time an important security property. The semantic soundness of such mechanism is hence crucial for the global security of the Java platform. In this section, we formally define the type system and prove its soundness with respect to an operational semantics. All the results of this section have been machine-checked with the Coq proof assistant[3].

Syntax. Our language is a simple language in-between Java source and Java bytecode. Our goal was to have a language close enough to the bytecode in order to easily obtain, from the specification, a naive implementation at the bytecode level while keeping a language easy to reason with. It is based on the decompiled language from Demange *et al.* [5] that provides a stack-less representation of Java bytecode programs. Fig. 6 shows the syntax of the language. A program is a record that handles a set of classes, a main class, a type annotation for each field and a lookup operator. This operator is used do determine during a virtual call the method ($p.\mathsf{lookup}\ c\ m$) (if any) that is the first overriding version of a method m in the ancestor classes of the class c. A class is composed of a super class (if any), a set of method and a special constructor method init. A method handles an array of instructions, a handler function such that ($m.\mathsf{handler}\ i\ e$) is the program point (if any) in the method m where the control flows after an exception e has been thrown at point i. Each method handles also four initialization types for the initial value of the variable `this` ($m.\mathsf{pre}$), its final value ($m.\mathsf{post}$), the type of its formal parameter[4] ($m.\mathsf{argtype}$) and the type of its return value ($m.\mathsf{rettype}$). The only expressions are the *null* constant, local variables and field reads. For this analysis, arithmetic needs not to be taken into account. We only manipulate objects. The instructions are the assignment to a local variable or to a field, object creation (new)[5], (non-deterministic) conditional jump, super constructor

[3] The development can be downloaded at http://www.irisa.fr/celtique/ext/rawtypes/

[4] For the sake of simplicity, each method has a unique formal parameter *arg*.

[5] Here, the same instruction allocates the object and calls the constructor. At bytecode level this gives raise to two separated instructions in the program (allocation and

$$
\begin{array}{rll}
\overline{Exc} \ni \overline{e} & ::= e \mid \bot & \text{(exception flag)} \\
\mathbb{L} \ni l & & \text{(location)} \\
\mathbb{V} \ni v & ::= l \mid null & \text{(value)} \\
\mathbb{M} = Var \to \mathbb{V} \ni \rho & & \text{(local variables)} \\
\mathbb{O} = Class \times Class_\bot \times (Field \to \mathbb{V}) \ni o & ::= [c, c_{init}, o] & \text{(object)} \\
\mathbb{H} = \mathbb{L} \to \mathbb{O}_\bot \ni \sigma & & \text{(heap)} \\
CS \ni cs & ::= (m, i, l, \rho, r) :: cs \mid \varepsilon & \text{(call stack)} \\
\mathbb{S} = Meth \times \mathcal{L} \times \mathbb{M} \times \mathbb{H} \times CS \times \overline{Exc} \ni st & ::= \langle m, i, \rho, \sigma, cs \rangle_{\overline{e}} & \text{(state)}
\end{array}
$$

Fig. 7. Semantic Domains.

call, virtual method call, return, and a special instruction that we introduce for explicit object initialization: *SetInit*.

Semantic Domains. Fig. 7 shows the concrete domain used to model the program states. The state is composed of the current method m, the current program point i in m (the index of the next instruction to be executed in $m.\mathsf{instrs}$), a function for local variables, a heap, a call stack and an exception flag. The heap is a partial function which associates to a location an object $[c, c_{init}, o]$ with c its type, c_{init} its current initialization level and o a map from field to value (in the sequel o is sometimes confused with the object itself). An initialization $c_{init} \in Class$ means that each constructors of c_{init} and its super-classes have been called on the object and have returned without abrupt termination. The exception flag is used to handle exceptions: a state $\langle \cdots \rangle_e$ with $e \in Exc$ is reached after an exception e has been thrown. The execution then looks for a handler in the current method and if necessary in the methods of the current call stack. When equal to $\bot$, the flag is omitted (normal state). The call stack records the program points of the pending calls together with their local environments and the variable that will be assigned with the result of the call.

Initialization Types. We can distinguish three different kinds of initialization types. Given a heap σ we define a value type judgment $h \vdash v : \tau$ between values and types with the following rules.

$$
\frac{}{\sigma \vdash null : \tau} \qquad \frac{}{\sigma \vdash l : Raw^\bot} \qquad \frac{\sigma(l) = [c_{dyn}, c_{init}, o] \quad \forall c', c_{dyn} \preceq c' \wedge c \preceq c' \Rightarrow c_{init} \preceq c'}{\sigma \vdash l : Raw(c)} \qquad \frac{\sigma(l) = [c, c, o]}{\sigma \vdash l : Init}
$$

The relation $\preceq$ here denotes the reflexive transitive closure of the relation induced by the *super* element of each class. $Raw^\bot$ denotes a reference to an object which may be completely uninitialized (at the very beginning of each constructor). *Init* denotes a reference to an object which has been completely initialized. Between those two "extreme" types, a value may be typed as $Raw(c)$ if at least one

later constructor invocation) but the intermediate representation generator [5] on which we rely is able to recover such construct.

$$\frac{m.\mathsf{instrs}[i] = x \leftarrow \mathit{new}\ c(y) \quad x \neq \mathit{this} \quad \mathit{Alloc}(\sigma, c, l, \sigma') \quad \sigma' \vdash \rho(y) : c.\mathsf{init.argtype}}{\langle m, i, \rho, \sigma, cs\rangle \Rightarrow \langle c.\mathsf{init}, 0, [\cdot \mapsto \mathit{null}][\mathit{this} \mapsto l][\mathit{arg} \mapsto \rho(y)], \sigma', (m, i, \rho, x) :: cs\rangle}$$

$$\frac{m.\mathsf{instrs}[i] = \mathit{SetInit} \quad m = c.\mathsf{init} \quad \rho(\mathit{this}) = l \quad \mathit{SetInit}(\sigma, c, l, \sigma')}{\langle m, i, \rho, \sigma, cs\rangle \Rightarrow \langle m, i{+}1, \rho, \sigma', cs\rangle}$$

$$\frac{\begin{array}{c} m.\mathsf{instrs}[i] = \mathit{return}\ x \quad \rho(\mathit{this}) = l \quad ((\forall c,\ m \neq c.\mathsf{init}) \Rightarrow \sigma = \sigma') \\ (\forall c,\ m = c.\mathsf{init} \Rightarrow \mathit{SetInit}(\sigma, c, l, \sigma') \wedge x = \mathit{this}) \end{array}}{\langle m, i, \rho, \sigma, (m', i', \rho', r) :: cs\rangle \Rightarrow \langle m', i'{+}1, \rho'[r \mapsto \rho(x)], \sigma', cs\rangle}$$

Fig. 8. Operational Semantics (excerpt).

constructor of c and of each parent of c has been executed on all objects that may be reference from this value. We can derive from this definition the sub-typing relation $\mathit{Init} \sqsubseteq \mathit{Raw}(c) \sqsubseteq \mathit{Raw}(c') \sqsubseteq \mathit{Raw}^{\perp}$ if $c \preceq c'$. It satisfies the important monotony property

$$\forall \sigma \in \mathbb{H}, \forall v \in \mathbb{V}, \forall \tau_1, \tau_2 \in \mathit{Type},\ \tau_1 \sqsubseteq \tau_2 \wedge \sigma \vdash v : \tau_1 \Rightarrow \sigma \vdash v : \tau_2$$

Note that the sub-typing judgment is disconnected from the static type of object. In a first approach, we could expect to manipulate a pair (c, τ) with c the static type of an object and τ its initialization type and consider equivalent both types $(c, \mathit{Raw}(c))$ and (c, Init). Such a choice would however impact deeply on the standard dynamic mechanism of a JVM: each dynamic cast from A to B (or a virtual call on a receiver) would requires to check that an object has not only an initialization level set up to A but also set up to B.

Operational Semantics. We define the operational semantics of our language as a small-step transition relation over program states. A fixed program p is implicit in the rest of this section. Fig. 8 presents some selected rules for this relation. The rule for the *new* instruction includes both the allocation and the call to the constructor. We use the auxiliary predicate $\mathit{Alloc}(\sigma, c, l, \sigma')$ which allocate a fresh location l in heap σ with type c, initialization type equals to $\perp$ and all fields set equal to *null*. The constraint $\sigma' \vdash \rho(y) : c.\mathsf{init.argtype}$ explicitly asks the caller of the constructor to give a correct argument with respect to the policy of the constructor. Each call rules of the semantics have similar constraints. The execution is hence stuck when an attempt is made to call a method with badly typed parameters. The *SetInit* instruction updates the initialization level of the object in *this*. It relies on the predicate $\mathit{SetInit}(\sigma, c, l, \sigma')$ which specifies that σ' is a copy of σ where the object at location l has now the initialization tag set to c if the previous initialization was $c.\mathit{super}$. It forces the current object (**`this`**) to be considered as initialized up to the current class (*i.e.* as if the constructor of the current class had returned, but not necessarily the constructors of the subsequent classes). This may be used in the constructor, once all fields that need to be initialized have been initialized and if some method requiring a non-raw object needs to be called. Note that this instruction is really sensitive: using this instruction too early in a constructor may break the security of

the application. The *return* instruction uses the same predicate when invoked in a constructor. For convenience we requires each constructor to end with a *return this* instruction.

Typing Judgment. Each instruction *ins* of a method m is attached a typing rule (given in Fig. 9) $m \vdash ins : L \to L'$ that constraint the type of variable before (L) and after (L') the execution of *ins*.

Expression typing

$$\frac{}{L \vdash e.f : (p.\mathsf{fields}\ f)} \qquad \frac{}{L \vdash x : L(x)} \qquad \frac{}{L \vdash null : Init}$$

Instruction typing

$$\frac{L \vdash e : \tau \quad x \neq this}{m \vdash x \leftarrow e : L \to L[x \mapsto \tau]} \qquad \frac{L(y) \sqsubseteq (p.\mathsf{fields}\ f)}{m \vdash x.f \leftarrow y : L \to L} \qquad \frac{}{\Gamma, m \vdash if\ \star\ jmp : L \to L}$$

$$\frac{L(this) \sqsubseteq m.\mathsf{post} \quad L(x) \sqsubseteq m.\mathsf{rettype} \quad (\forall c,\ m = c.\mathsf{init} \Rightarrow L(this) \sqsubseteq Raw(c.\mathsf{super}))}{m \vdash return\ x : L \to L}$$

$$\frac{L(y) \sqsubseteq c.\mathsf{init.argtype}}{m \vdash x \leftarrow new\ c(y) : L \to L[x \mapsto Init]} \qquad \frac{c' = c.super \quad L(y) \sqsubseteq c'.\mathsf{init.argtype}}{c.\mathsf{init} \vdash super(y) : L \to L[this \mapsto Raw(c')]}$$

$$\frac{L(r) \sqsubseteq m'.\mathsf{pre} \quad L(y) \sqsubseteq m'.\mathsf{argtype} \quad L' = L[r \mapsto m'.\mathsf{post}][x \mapsto m'.\mathsf{rettype}]}{m \vdash x \leftarrow r.m'(y) : L \to L'} \qquad \frac{L(this) \sqsubseteq Raw(c.\mathsf{super}) \quad L' = L[this \mapsto Raw(c)]}{c.\mathsf{init} \vdash SetInit : L \to L'}$$

Fig. 9. Flow sensitive type system

Definition 1 (Well-typed Method). *A method m is* well-typed *if there exists flow sensitive variable types $L \in \mathcal{L} \to Var \to Type$ such that*

- $m.\mathsf{pre} \sqsubseteq L(0, this)$ *and* $m.\mathsf{argtype} \sqsubseteq L(0, arg)$,
- *for all instruction ins at point i in m and every successor j of i, there exists a map of variable types $L' \in Var \to Type$ such that $L' \sqsubseteq L(j)$ and the typing judgment $m \vdash ins : L(i) \to L'$ holds. If i is in the handler j of an exception e (*i.e $(m.\mathsf{handler}\ i\ e = j)$*) then $L(i) \sqsubseteq L(j)$.*

The typability of a method can be decided by turning the set of typing rules into a standard dataflow problem. The approach is standard [7] and not formalized here.

Definition 2 (Well-typed Program). *A program p is* well-typed *if all its methods are well-typed and the following constraints holds:*

1. *for every method m that is overridden by a method m' (*i.e *there exists c, such that $(p.\mathsf{lookup}\ c\ m = m')$),*

$$m.\mathsf{pre} \sqsubseteq m'.\mathsf{pre} \ \wedge m.\mathsf{argtype} \sqsubseteq m'.\mathsf{argtype} \wedge$$
$$m.\mathsf{post} \sqsupseteq m'.\mathsf{post} \wedge\ m.\mathsf{rettype} \sqsupseteq m'.\mathsf{rettype}$$

2. *in each method, every first point, jump target and handler point contain an instruction and every instruction (except return) has a next instruction,*
3. *the default constructor* $c.\mathsf{init}$ *of each class* c *is unique.*

In this definition only point 1 is really specific to the current type system. The other points are necessary to established the progress theorem of the next section.

Type Soundness. We rely on an auxiliary notion of well-formed states that capture the semantics constraints enforce by the type system. A state $\langle m, i, \rho, \sigma, cs\rangle$ is *well-formed* (wf) if there exists a type annotation $L_p \in (\mathit{Meth} \times \mathcal{L}) \to (\mathit{Var} \to \mathit{Type})$ such that

$$\begin{array}{ll} \forall l \in \mathcal{L}, \forall o \in \mathbb{O}, \sigma(l) = o \Rightarrow \sigma \vdash o(f) : (p.\mathsf{fields}\ f) & \text{(wf. heap)} \\ \forall x \in \mathit{Var}, \sigma \vdash \rho(x) : L_p[m, i](x) & \text{(wf. local variables)} \\ \forall (m', i', \rho', r) \in cs,\ \forall x, \sigma \vdash \rho'(x) : L_p[m', i'](x) & \text{(wf. call stack)} \end{array}$$

Given a well-typed program p we then establish two key theorems. First, any valid transition from a well-formed state leads to another well-formed state (*preservation*) and then, from every well-formed state there exists at least a transition (*progress*). As a consequence we can establish that starting from an initial state (which is always well-formed) the execution is never stuck, except on final configuration. This ensures that all initialization constraints given in the operational semantics are satisfied without requiring any dynamic verification.

Limitations. The proposed language has some limitations compared to the Java (bytecode) language. Static fields and arithmetic have not been introduced but are handled by our implementation and do not add particular difficulties. Arrays have not been introduced in the language neither. Our implementation conservatively handles arrays by allowing only writes of *Init* references in arrays. Although this approach seems correct it has not been proved and it is not flexible enough (cf. Section 6). Multi-threading as also been left out of the current formalization but we conjecture the soundness result still holds with respect to the Java Memory Model because of the flow insensitive abstraction made on the heap. As for the BCV, native methods may brake the type system. It is their responsibility to respect the policy expressed in the program.

6 A Case Study: Sun's JRE

In order to show that our type system allows to verify legacy code with only a few annotations, we implemented a standalone prototype, handling the full Java bytecode, and we tested all classes of packages `java.lang`, `java.security` and `javax.security` of the JRE1.6.0_20.

348 classes out of 381 were proven safe *w.r.t.* the default policy without any modification. By either specifying the actual policy when the default policy was too strict, or by adding cast instructions (see below) when the type system was

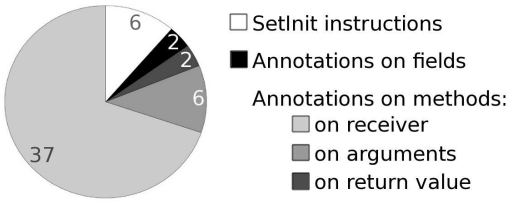

Fig. 10. Distribution of the 47 annotations and 6 instructions added to successfully type the three packages of the JRE.

not precise enough, we were able to verify 377 classes, that is to say 99% of classes. We discuss below the 4 remaining classes that are not yet proven correct by our analysis. The modifications represent only 55 source lines of code out of 131,486 for the three packages studied. Moreover most code modifications are to express the actual initialization policy, which means existing code can be proven safe. Only 45 methods out of 3,859 (1.1%) and 2 fields out of 1,524 were annotated. Last but not least, the execution of the type checker takes less than 20 seconds for the packages studied.

Adapting the security policy. Fig. 10 details the annotations and the `SetInit` added to specify the security policy. In the runtime library, a usual pattern consists in calling methods that initialize fields during construction of the object. In that case, a simple annotation `@Pre(@Raw(super(C)))` on methods of class `C` is necessary. These cases represent the majority of the 37 annotations on method receivers. 6 annotations on method arguments are used, notably for some methods of `java.lang.SecurityManager` which check permissions on an object during its initialization. The instruction `SetInit` is used when a constructor initializes all the fields of the receiver and then call methods on the receiver that are not part of the initialization. In that case the method called need at least a `Raw(C)` level of initialization and the `SetInit` instruction allows to express that the constructor finished the minimum initialization of the receiver. Only 6 `SetInit` intructions are necessary.

Cast instructions. Such a static and modular type checking introduces some necessary loss of precision — which cannot be completely avoided because of computability issues. To be able to use our type system on legacy code without deep modifications, we introduce two dynamic cast operators: `(Init)` and `(Raw)`. The instruction `y = (Init)x;` allows to dynamically check that `x` points to a fully initialized object: if the object is fully initialized, then this is a simple assignation to `y`, otherwise it throws an exception. As explained in Section 3, the invariant needed is often weaker and the correctness of a method may only need a $Raw(c)$ reference. `y = (Raw(C))x` dynamically checks that `x` points to an object which is initialized up to the constructor of class `C`.

Only 4 cast instructions are necessary. There are needed in two particular cases. First, when a field must be annotated, but annotation on fields were only necessary on two fields — they imply the use of 3 `(Init)` cast instructions. The

second case is on a receiver in a `finalize()` method that checks that some fields are initialized, thereby checking that the object was `Raw(C)` but the type system could not infer this information. The later case implies to use the unique `(Raw(C))` instruction added.

Remaining classes. Finally, only 4 classes are not well-typed after the previous modifications. Indeed the compiler generates some code to compile inner classes and part of this code needs annotations in 3 classes. These cases could be handled by doing significant changes on the code, by adding new annotations dedicated to inner classes or by annotating directly the bytecode. The one class remaining is not typable because of the limited precision of our analysis on arrays: one can only store `@Init` values in arrays. To check this later class, our type system needs to be extended to handle arrays more precisely but this is left for future work.

Special case of finalize methods. As previously exposed, `finalize()` methods may be invoked on a completely uninitialized receiver. Therefore, we study the case of `finalize()` methods in the packages `java.*` and `javax.*`. In the classes of those packages there are 28 `finalize()` methods and only 12 succeed to be well-typed with our default annotation values. These are either empty or do not use their receiver at all. For the last 16 classes, the necessary modifications are either the use of cast instructions when the code's logic guarantees the success of cast, or the addition of `@Pre(@Raw)` annotations on methods called on the receiver. In that case, it is important to verify that the code of any called method is defensive enough. Therefore, the type system forced us to pay attention to the cases that could lead to security breaches or crashes at run time for `finalize()` methods. After a meticulous checking of the code we added the necessary annotations and cast instructions that allowed to verify the 28 classes.

7 Conclusion and Future Work

We have proposed herein a solution to enforce a secure initialization of objects in Java. The solution is composed of a modular type system which allows to manage uninitialized objects safely when necessary, and of a modular type checker which can be integrated into the BCV to statically check a program at load time. The type system has been formalized and proved sound, and the type-checker prototype has been experimentally validated on more than 300 classes of the Java runtime library.

The experimental results point out that our default annotations minimize the user intervention needed to type a program and allows to focus on the few classes where the security policy needs to be stated explicitly. The possible adaptation of the security policy on critical cases allows to easily prevent security breaches and can, in addition, ensure some finer initialization properties whose violation could lead the program to crash. On one hand, results show that such a static and modular type checking allows to prove in an efficient way the absence of

bugs. On the other hand, rare cases necessitate the introduction of dynamic features and analysis to be extended to analyze more precisely arrays. With such an extension, the checker would be able to prove more classes correct, but this is left for future work.

On the formalization side, an obvious extension is to establish the soundness of the approach in presence of multi-threading. We conjecture the soundness result still holds with respect to the Java Memory Model because of the flow insensitive abstraction made on the heap.

The prototype and the Coq formalization and proofs can be downloaded from http://www.irisa.fr/celtique/ext/rawtypes/.

Acknowledgment. This work was partly supported by the Région Bretagne and by the ANSSI (JavaSec project, see http://www.ssi.gouv.fr/site_article226.html).

References

[1] A. Buckley. JSR 202: Java^{TM}class file specification update, December 2006. http://jcp.org/en/jsr/detail?id=202.

[2] The CERT Sun Microsystems secure coding standard for Java, February 2010. https://www.securecoding.cert.org/confluence/display/java/.

[3] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Survey*, 41(3), 2009.

[4] D. Dean, E.W. Felten, and D.S. Wallach. Java security: From HotJava to Netscape and beyond. *IEEE Symposium on Security and Privacy*, pages 190–200, 1996.

[5] Delphine Demange, Thomas Jensen, and David Pichardie. A provably correct stackless intermediate representation for java bytecode. Research Report RR-7021, INRIA, 2009. http://hal.inria.fr/inria-00414099/en/.

[6] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *Proc. of OOPSLA*, pages 337–350. ACM, 2007.

[7] S. N. Freund and J. C. Mitchell. A type system for the Java bytecode language and verifier. *J. Autom. Reasoning*, 30(3-4):271–321, 2003.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification (3rd Edition)*. The Java. Addison Wesley, 3rd edition edition, 2005.

[9] Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In *POPL*, pages 53–65. ACM, 2009.

[10] Secunia Advisory SA10056: Sun JRE and SDK untrusted applet privilege escalation vulnerability. Web, October 2003. http://secunia.com/advisories/10056/.

[11] Sun. Secure coding guidelines for the Java programming language, version 3.0. Technical report, Oracle, 2010. http://java.sun.com/security/seccodeguide.html.

[12] Christopher Unkel and Monica S. Lam. Automatic inference of stationary fields: a generalization of Java's final fields. In *Proc. of POPL*, pages 183–195, New York, NY, USA, 2008. ACM.