Securing Class Initialization

Keiko Nakata[1] and Andrei Sabelfeld[2]

[1] Institute of Cybernetics, Tallinn University of Technology, Tallinn, Estonia
[2] Chalmers University of Technology, Gothenburg, Sweden

Abstract. Language-based information-flow security is concerned with specifying and enforcing security policies for information flow via language constructs. Although much progress has been made on understanding information flow in object-oriented programs, the impact of class initialization on information flow has been so far largely unexplored. This paper turns the spotlight on security implications of class initialization. We discuss the subtleties of information propagation when classes are initialized and propose a formalization that illustrates how to track information flow in presence of class initialization by a type-and-effect system for a simple language. We show how to extend the formalization to a language with exception handling.

1 Introduction

Language-based concepts and techniques are becoming increasingly popular in the context of security [Koz99,SMH00,WAF00,SM03,Ler03,MSL+08,Cro09,Fac09] because they provide an appropriate level of abstraction for specifying and enforcing application and language-sensitive security policies. Popular examples include Java stack inspection [WAF00], which enforces a stack-based access-control discipline, and Java bytecode verification [Ler03], which traverses bytecode and verifies its type safety, as well as web language-based mechanisms such as Caja [MSL+08], ADsafe [Cro09], and FBJS [Fac09], which use program transformation and language subsets in order to enforce sandboxing and separation properties.

Language-based information-flow security [SM03] is concerned with specifying and enforcing security policies for information flow via language constructs. There has been much recent progress on understanding information flow in languages of increasing complexity [SM03], and, consequently, information-flow security tools for languages such as Java, ML, and Ada have emerged [MZZ+10,Sim03,Sys10]. In particular, information flow in object-oriented languages has been an area of intensive development [Mye99,BS99,BCG+02,ABF03,BFLM05,BN05,ABB06,Nau06,BRN06,HS09]. However, it is surprising that the impact of class initialization, being an important aspect of object-oriented programs, has received scarce attention in the context of security. In a language like Java, class initialization is *lazy*: classes are loaded as they are first used. This introduces challenges for information-flow tracking, in particular when class initialization may trigger initialization of other classes, which, for example, may include superclasses. Additional complexity is introduced by exceptions raised during initialization. Exceptions may be exploited to leak secret information.

Because of its power, Java's class loading mechanism [LB98] is a target for our model. A class is loaded, linked and initialized lazily on demand when the class is actively used for the first time [LY99] [3]. Moreover the programmer may define application-specific loading policies. Class loading constitutes one of the most compelling features of the Java platform.

This paper turns the spotlight on security implications of class initialization (and loading and liking, which are prerequisites for initialization). We discuss the subtleties of information propagation when classes are initialized. The key issue is that class initialization may perform side effects (such as opening a file or updating the memory). The side effects may be exploited by the attacker who may deduce from these side effects which classes have been (not) initialized, which is sometimes sufficient to learn secret information.

We propose a formalization that illustrates how to track information flow in presence of class initialization by a type-and-effect system for a simple language. By ensuring that classes may not be initialized inside conditionals and loops that branch on secret data, the type-and-effect system guarantees security in a form of *noninterference* [GM82]. We show how to extend the formalization to a language with exception handling. The only approach we are aware of that actually considers class initialization in the context of information-flow security is Jif [Mye99,MZZ+10]. However, Jif's restrictions on code initialization are rather severe: this code is restricted to simple constant manipulation that may not raise any exceptions. Our treatment of class initialization is more liberal than Jif's and yet we demonstrate that it is secure. We argue that this liberty is sometimes desired in scenarios such as server-side code.

2 Background

This section presents informal considerations that lead up to a formalization in the following sections. For illustration purposes, we use a simple subset of Java with classes that contain static fields. We assume variables and class fields are partitioned into *high* (secret) and *low* (public). We assume that l and h are typical low and high variables, respectively. The security goal is to prevent programs from leaking initial values of secret data into final values of public data. The *context* corresponds to a body of a conditional or loop. We say that the context is *high* if the guard depends on a secret (i.e., contains a secret variable or field) and *low* otherwise.

Consider the following two class definitions for class names C and D with low fields g and f, respectively:

$$\begin{array}{l}\mathsf{class}\ C\ \{\ g = 1\ \}\\ \mathsf{class}\ D\ \{\ f = 1/C.g\ \}\end{array}$$

Certainly the above definitions may be considered secure since no high data is involved. However, an attempt to instantiate an object of D may lead to an information leak:

$$\begin{array}{ll}P_0: & C.g := 0;\\ & \mathsf{if}\ h = 0\ \mathsf{then\ new}\ D\ \mathsf{else\ skip}\end{array}$$

[3] The JVM specification permits the large flexibility as to the timing of loading and linking. But these activities must *appear* as if they happen on the class's (or interface's) first active use.

Indeed, the above program results in an error only when the high variable h initially contains 0, in which case the class D is initialized. Note that in the terminology we have introduced, the initialization occurs in high context. The attacker learns about the secret value of h by observing the termination behavior.

It is illustrative to compare the above program that leaks through termination behavior with the following one that does not:

$$\begin{array}{ll} P_1: & \mathsf{new}\ D; \\ & C.g := 0; \\ & \mathsf{if}\ h = 0\ \mathsf{then}\ \mathsf{new}\ D\ \mathsf{else}\ \mathsf{skip} \end{array}$$

In this latter program, D is initialized before the assignment. More importantly, D has been initialized before it is used in high context: the second use does not incur any initialization activities.

In Java, when initialization of a class has completed abnormally by throwing some exception, the class is marked as erroneous. Initialization of a class in an erroneous state is not possible [LY99, Ch. 2] [4]. This makes initialization failure persistent in the sense that when initialization of a class failed on it first (active) use, then it will fail on the second use irrespective of the state in which the second initialization is attempted [5]. Catching initialization errors introduces a delicate scenario of information leaks. For instance, consider the following program:

$$\begin{array}{ll} P_2: & C.g := 0; \\ & \mathsf{if}\ h = 0\ \mathsf{then}\ (\mathsf{try}\ \mathsf{new}\ D\ \mathsf{catch}\ \mathsf{skip})\ \mathsf{else}\ \mathsf{skip}; \\ & C.g := 1; \\ & \mathsf{new}\ D \end{array}$$

The above program again results in an error only when the high variable h initially contains 0. The next variation of the example shows how to exploit this flow so that the resulting program always terminates normally and reflects the initial value of h in the final value of l. Standard security type systems (e.g., [Mye99,PS03,HS06,AS09]) allow liberate handling of exceptions raised by expressions that are independent of secret data, as long as these expressions are used in public context. Since seemingly neither class definitions of C nor D involves high variables, one may be tempted to consider possible errors caused by initializing D as low. However, the following program illustrates the subtlety of the problem:

$$\begin{array}{ll} P_3: & C.g := 0; \\ & \mathsf{if}\ h = 0\ \mathsf{then}\ (\mathsf{try}\ \mathsf{new}\ D\ \mathsf{catch}\ \mathsf{skip})\ \mathsf{else}\ \mathsf{skip}; \\ & C.g := 1; \\ & \mathsf{try}\ \mathsf{new}\ D; l := 1\ \mathsf{catch}\ l := 0 \end{array}$$

The above program successfully terminates irrespective of the initial value at h, and the final value at l indicates whether h was 0 or not.

[4] To be precise, the *Class* object representing the class is labeled as erroneous.

[5] Initialization may recover for instance by resorting to garbage collection. But normally a class is eligible for unloading when the running application has no reference to the class.

Security might be compromised within correct, i.e., error-free, programs. Moreover class hierarchy also may impact on the security: before a class is initialized, its superclasses are initialized. For instance, consider class definitions below, involving only low fields.

$$\begin{array}{l}\mathsf{class}\ C_0\ \{\ g = 1\ \}\\ \mathsf{class}\ D_0\ \{\ f = C_0.g{+}{+}\ \}\\ \mathsf{class}\ D_1\ \mathsf{extends}\ D_0\ \{\}\end{array}$$

The next program leaks the initial value of h into the final value of $C_0.g$.

$$\begin{array}{rl}P_4: & \mathsf{new}\ C_0;\\ & \mathsf{if}\ h = 0\ \mathsf{then\ new}\ D_1\ \mathsf{else\ skip}\end{array}$$

Combining the two previous programs yields a scenario, where class hierarchy and persistence of initialization failure cooperate to leak information:

$$\begin{array}{l}\mathsf{class}\ C_0\ \{\ g = 1\ \}\\ \mathsf{class}\ D_0\ \{\ f = 1/C_0.g\ \}\\ \mathsf{class}\ D_1\ \mathsf{extends}\ D_0\ \{\}\end{array}$$

$$\begin{array}{rl}P_5: & C_0.g := 0;\\ & \mathsf{if}\ h = 0\ \mathsf{then}\ (\mathsf{try\ new}\ D_1\ \mathsf{catch\ skip})\ \mathsf{else\ skip};\\ & C_0.g := 1;\\ & \mathsf{try\ new}\ D_0; l := 1\ \mathsf{catch}\ l := 0\end{array}$$

Again the resulting program always terminates normally and reflects secret input values in the public results.

The bottom line is that class initialization may perform side effects, causing information to leak. One rather conservative approach to securing class initialization is to eliminate any possibilities of side effects during initialization and disallow errors due to initialization to be caught, an approach taken in Jif [Mye99,MZZ+10]. This approach rules out, among other, read and write access to instance as well as static fields, method calls and object creation during initialization. For example, a static field of a reference type may only be initialized to null, which would exclude some standard Java APIs [Sun], such as *(java.lang.)Boolean* and *String*, etc. Indeed Jif restricts (class) field initializers to simple constant manipulation that may not raise any exceptions. While it is rarely good practice to catch initialization errors within ordinary methods, such as methods in libraries, there are several scenarios where it is good practice to catch them, such as in server applications to avoid crashing the entire system due to third party applications or to log messages. For instance, Fortress, the primary product from the Excalibur software project [Exc], catches *LinkageError* and rethrows an object of a subclass of *Exception*, which may in turn be caught and logged.

This paper goes ahead to propose and formalize a different approach: we restrict class initialization in high contexts but allows side effects during initialization. Section 4 develops a type-and-effect system for a simple language, defined in Section 3. The type-and-effect system ensures a class has been initialized before it is used in high contexts and, as we show, guarantees information-flow security. Moreover, Section 5 shows how to scale our approach, when initialization errors are permitted to be caught. Section 6 discusses related work, and Section 7 concludes.

3 Language

We define the language for our formal study by the following syntax:

$$
\begin{array}{lll}
\textit{Expressions} & & e ::= n \mid x \mid e_0 \ op \ e_1 \mid C.f \\
\textit{Statements} & & s ::= \mathsf{skip} \mid s_0; s_1 \mid x := e \mid C.f := e \mid \mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \\
& & \quad \mid \mathsf{while}\ e\ \mathsf{do}\ s_t \\
\textit{Class definitions} & & CL ::= \mathsf{class}\ C\ \{f_0 = e_0,\ \ldots, f_k = e_k\}
\end{array}
$$

Metavariables x, n, C and f range over variables, integers, class names and field names, respectively. We assume given a binary partial operator op on integers, i.e., op may signal an error. We write $n_0 \ op \ n_1 = \bullet$ when op signals an error on operands n_0 and n_1. A class definition $\mathsf{class}\ C\ \{f_0 = e_0,\ \ldots, f_k = e_k\}$ declares a class name C consisting of the (static) fields f_i's with e_i's being initializing expressions. Then a class table CT is a (finite) mapping from class names to class definitions. A program is a pair (CT, s) of a class table and a statement. To lighten the notation, we assume a fixed class table CT hereafter.

A *state* (or *store*), ranged over by σ, maps variables to integers and class names to *abstract class objects*. An abstract class object, or simply class object, denotes the loaded status of a class name C in a state σ: C is uninitialized in σ when $\sigma(C) = \circ$; initialization is in progress or has been successfully completed when $\sigma(C) = \{f_0 = n_0, \ldots, f_k = n_k\}$, where f_i's are the fields of the class C; C has failed to initialize when $\sigma(C) = \bullet$. We write $\mathit{uninitialized}(\sigma)$ to denote the set of uninitialized classes in σ, i.e., $\{C \mid \sigma(C) = \circ\}$, $\mathit{initialized}(\sigma)$ the set of classes of which initialization is in progress or has been completed in σ, i.e., $\{C \mid \sigma(C) = \{f_1 = n_1, \ldots, f_k = n_k\}$ for some $f_1, \ldots, f_k, n_1, \ldots, n_k$, and $\mathit{failed}(\sigma)$ the set of classes that have failed to initialize, i.e., $\{C \mid \sigma(C) = \bullet\}$. When the context ensures C is in $\mathit{initialized}(\sigma)$, we may write $\sigma(C.f)$ to denote f-field of the class object for C in σ, i.e., $\sigma(C.f) = n$ where $\sigma(C) = \{\ldots, f = n, \ldots\}$, and $\sigma[C.f \mapsto n]$ to denote the update of f-field of the class object for C in σ by n.

Evaluation of expressions is given in Fig. 1. The relation $(\sigma, e) \downarrow (\sigma', n)$ states that the expression e in the state σ evaluates to the result n with the state being σ'. The relation $(\sigma, e) \uparrow \sigma'$ states that evaluating the expression e in the state σ fails in the state σ', signaling an error.

The inference rules in Fig. 1 are straightforward except those for reading from a field of a class. Both read and write access to a field of a class C triggers initialization of C. Class initializer $\rho(C, \sigma)$, to be defined below, initializes the class C in the state σ. If the initialization is in progress or has been successfully completed, written $\rho(\sigma, C) \downarrow \sigma'$, σ' contains a class object for C and evaluation of $C.f$ returns f-field of the class object. If the initialization fails, written $\rho(\sigma, C) \uparrow \sigma'$, so does the evaluation of $C.f$.

The small-step operational semantics for the statements is given in Fig. 2. The one-step reduction relation $\langle \sigma, s \rangle \to \langle \sigma', s' \rangle$ states that in the state σ the statement s one-step reduces to s' with the next state being σ'. The relation $\langle \sigma, s \rangle \to \langle \sigma', \bullet \rangle$ states that the statement s in the state σ signals an error in the state σ'. We write $\langle \sigma, s \rangle \downarrow \sigma'$, stating that s in the initial state σ successfully terminates in the final state σ'. Or, $\langle \sigma, s \rangle \downarrow \sigma'$ if $\langle \sigma, s \rangle \to^* \langle \sigma', \mathsf{skip} \rangle$, where $\to^*$ denotes the reflexive and transitive closure of $\to$.

$$\frac{}{(\sigma, n) \downarrow (\sigma, n)} \quad \frac{}{(\sigma, x) \downarrow (\sigma, \sigma(x))}$$

$$\frac{(\sigma, e_0) \downarrow (\sigma', n_0) \quad (\sigma', e_1) \downarrow (\sigma'', n_1) \quad n_0 \ op \ n_1 = n}{(\sigma, e_0 \ op \ e_1) \downarrow (\sigma'', n)}$$

$$\frac{(\sigma, e_0) \uparrow \sigma'}{(\sigma, e_0 \ op \ e_1) \uparrow \sigma'} \quad \frac{(\sigma, e_0) \downarrow (\sigma', n_0) \quad (\sigma', e_1) \uparrow \sigma''}{(\sigma, e_0 \ op \ e_1) \uparrow \sigma''}$$

$$\frac{(\sigma, e_0) \downarrow (\sigma', n_0) \quad (\sigma', e_1) \downarrow (\sigma'', n_1) \quad n_0 \ op \ n_1 = \bullet}{(\sigma, e_0 \ op \ e_1) \uparrow \sigma''}$$

$$\frac{\rho(\sigma, C) \downarrow \sigma'}{(\sigma, C.f) \downarrow (\sigma', \sigma'(C.f))} \quad \frac{\rho(\sigma, C) \uparrow \sigma'}{(\sigma, C.f) \uparrow \sigma'}$$

Fig. 1. Expression evaluation

Similarly we write $\langle \sigma, s \rangle \uparrow \sigma$, stating that s in the initial state σ abnormally terminates at the state σ'. Or, $\langle \sigma, s \rangle \uparrow \sigma'$ if $\langle \sigma, s \rangle \rightarrow^* \langle \sigma', \bullet \rangle$.

The inference rules for the one-step reduction relation are again straightforward. Assignment to a field of a class triggers initialization of the class, and thus may fail if the initialization fails.

Finally Fig. 3 defines the class initializer. If initialization of the class has been initiated and has not failed, i.e., $\sigma(C) = \{f_0 = n_0, \ldots, f_n = n_k\}$, then the initializer immediately returns. This covers both the cases that initialization is in progress and that initialization has been successfully completed. If initialization has previously failed, therefore the class object is in erroneous state, i.e., $\sigma(C) = \bullet$, then initialization is not possible. Otherwise, the initialization is initiated: the fields are first set to default values, namely 0, then updated according to their initializing expressions. The initialization may fail, in which case the class object is marked erroneous.

4 Specifying and Enforcing Security

We now introduce a security condition for our language and then develop a security type system for statically guaranteeing this condition.

4.1 Security condition

As before, we assume a simple security lattice [Den76] consisting of only two levels low (public) and $high$ (secret), with $low \sqsubset high$. Metavariables ℓ and pc range over security levels. Then a *security environment* Γ is a finite mapping from variables and pairs (C, f) of a class name and a field name of the class to their security levels. We extend Γ to expressions by assuming an expression is mapped to the least upper bound of the security levels that occur in it. Again, for the sake of notational simplicity, we assume a fixed security environment Γ in what follows.

$$\frac{(\sigma, e) \downarrow (\sigma', n)}{\langle \sigma, x := e \rangle \to \langle \sigma'[x \mapsto n], \mathsf{skip} \rangle}$$

$$\frac{(\sigma, e) \downarrow (\sigma', n) \quad \rho(\sigma', C) \downarrow \sigma''}{\langle \sigma, C.f := e \rangle \to \langle \sigma''[C.f \mapsto n], \mathsf{skip} \rangle}$$

$$\frac{(\sigma, e) \uparrow \sigma'}{\langle \sigma, C.f := e \rangle \to \langle \sigma', \bullet \rangle} \qquad \frac{(\sigma, e) \downarrow (\sigma', n) \quad \rho(\sigma', C) \uparrow \sigma''}{\langle \sigma, C.f := e \rangle \to \langle \sigma'', \bullet \rangle}$$

$$\frac{}{\langle \sigma, \mathsf{skip}; s \rangle \to \langle \sigma, s \rangle}$$

$$\frac{\langle \sigma, s_0 \rangle \to \langle \sigma', s_0' \rangle}{\langle \sigma, s_0; s_1 \rangle \to \langle \sigma', s_0'; s_1 \rangle} \qquad \frac{\langle \sigma, s_0 \rangle \to \langle \sigma', \bullet \rangle}{\langle \sigma, s_0; s_1 \rangle \to \langle \sigma', \bullet \rangle}$$

$$\frac{(\sigma, e) \downarrow (\sigma', n) \quad n \neq 0}{\langle \sigma, \mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rangle \to \langle \sigma', s_t \rangle} \qquad \frac{(\sigma, e) \downarrow (\sigma', 0)}{\langle \sigma, \mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rangle \to \langle \sigma', s_f \rangle}$$

$$\frac{(\sigma, e) \downarrow (\sigma', n) \quad n \neq 0}{\langle \sigma, \mathsf{while}\ e\ \mathsf{do}\ s_t \rangle \to \langle \sigma', s_t; \mathsf{while}\ e\ \mathsf{do}\ s_t \rangle} \qquad \frac{(\sigma, e) \downarrow (\sigma', 0)}{\langle \sigma, \mathsf{while}\ e\ \mathsf{do}\ s_t \rangle \to \langle \sigma', \mathsf{skip} \rangle}$$

$$\frac{(\sigma, e) \uparrow \sigma'}{\langle \sigma, Q[e] \rangle \to \langle \sigma', \bullet \rangle}$$

where $Q ::= x := [] \mid \mathsf{if}\ []\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \mid \mathsf{while}\ []\ \mathsf{do}\ s_t$

Fig. 2. Operational semantics for statements

$$\frac{\sigma(C) = \{f_0 = n_0, \ldots, f_k = n_k\}}{\rho(\sigma, C) \downarrow \sigma} \qquad \frac{\sigma(C) = \bullet}{\rho(C, \sigma) \uparrow \sigma}$$

$$\frac{\begin{array}{c}\sigma(C) = \circ \quad CT(C) = \mathsf{class}\ C\ \{f_0 = e_0, \ldots, f_k = e_k\} \\ \langle \sigma[C \mapsto \{f_0 = 0, \ldots, f_k = 0\}], C.f_0 := e_0; \ldots; C.f_k := e_k \rangle \downarrow \sigma'\end{array}}{\rho(\sigma, C) \downarrow \sigma'}$$

$$\frac{\begin{array}{c}\sigma(C) = \circ \quad CT(C) = \mathsf{class}\ C\ \{f_0 = e_0, \ldots, f_k = e_k\} \\ \langle \sigma[C \mapsto \{f_0 = 0, \ldots, f_k = 0\}], C.f_0 := e_0; \ldots; C.f_k := e_k \rangle \uparrow \sigma'\end{array}}{\rho(C, \sigma) \uparrow \sigma'[C \mapsto \bullet]}$$

Fig. 3. Class initialization

Two states σ and σ' are *low-equivalent*, written $\sigma =_{low} \sigma'$, if they agree on low variables and fields and on class objects. Formally, $\sigma =_{low} \sigma'$ if the following three conditions hold:

- for any x such that $\Gamma(x) = low$, $\sigma(x) = \sigma'(x)$;
- $uninitialized(\sigma) = uninitialized(\sigma')$, $initialized(\sigma) = initialized(\sigma')$, and also $failed(\sigma) = failed(\sigma')$;
- for any C and f such that $\Gamma(C, f) = low$ and C is in $initialized(\sigma)$, $\sigma(C.f) = \sigma'(C.f)$.

We adopt a commonly-used baseline policy of *termination-insensitive noninterference* [VSI96,SM03,PS03]. Intuitively, a program satisfies noninterference if for any two initial memories that agree on public data, whenever the program runs that start in these memories terminate, then these runs result in the memories that also agree on public data. This policy is an appropriate fit for batchjob programs, where leaks due to (non)termination are ignored because they may leak at most one bit per execution [AHSS08].

Definition 1. *A statement s satisfies termination-insensitive noninterference if, for any low-equivalent states σ_0 and σ_1, $\langle \sigma_0, s\rangle \downarrow \sigma_0'$ and $\langle \sigma_1, s\rangle \downarrow \sigma_1'$ imply $\sigma_0' =_{low} \sigma_1'$.*

4.2 Type system

Class initialization potentially performs low side effects. Therefore, we are going to prohibit class initialization in high contexts. Our type system essentially performs a must-analysis: a class must have been initialized before it is used in high contexts.

Fig. 4 presents typing rules for expressions. The judgment $pc \vdash e : \delta \hookrightarrow \delta'$ states that the expression e is typable at the security level pc with the effect type $\delta \hookrightarrow \delta'$. The pretype δ and the posttype δ', respectively, represent the classes that must have been initialized before and after the expression is evaluated. Effectively $pc \vdash e : \delta \hookrightarrow \delta'$ is read that the class names that occur in e must be in δ when pc is $high$, and δ' is the union of the class names that occur in e and δ. The type system may be made more permissive by computing, for each class C, the set of the classes that are necessarily initialized during the initialization of C.

Fig. 5 presents typing rules for statements. The judgment $pc \vdash s : \delta \hookrightarrow \delta'$ is read similarly to that for expressions. For example, the rule for assignment prevents explicit flows (such as $l := h$), *implicit* flows [DD77] via control structure (as in if $h = 0$ then $l := 0$ else $l := 1$) in a standard fashion [DD77,VSI96,SM03]. In addition, the effect type information is propagated from the expression to command level, and class initialization in high contexts is ruled out. For the posttype of if-statement, we take the intersection of the posttypes of the branches, collecting classes that must have been initialized irrespective of which branch is taken. Similarly, the posttype of while-statement only includes classes initialized by evaluating the boolean guard, since the loop-body might not be executed at all.

To state soundness with respect to the security condition, we must ensure that the type environment Γ is consistent with the class table CT.

$$\frac{}{pc \vdash n : \delta \hookrightarrow \delta} \quad \frac{}{pc \vdash x : \delta \hookrightarrow \delta}$$

$$\frac{pc \vdash e_0 : \delta \hookrightarrow \delta_0 \quad pc \vdash e_1 : \delta \hookrightarrow \delta_1}{pc \vdash e_0 \; op \; e_1 : \delta \hookrightarrow \delta_0 \cup \delta_1}$$

$$\frac{}{low \vdash C.f : \delta \hookrightarrow \delta \cup \{C\}} \quad \frac{C \in \delta}{high \vdash C.f : \delta \hookrightarrow \delta}$$

Fig. 4. Typing of expressions

Definition 2. *A type environment Γ is well-formed with respect to a class table CT if, for any class name C and a field f of C, $\Gamma(e) \sqsubseteq \Gamma(C, f)$ where $CT(C) =$ $\mathsf{class}\ C\ \{\ldots, f = e, \ldots\}$.*

The existence of a well-formed environment ensures typability of the class table.

The type system is sound with respect to termination-insensitive noninterference:

Lemma 1. *Suppose Γ is well-formed with respect to CT. If $pc \vdash s : \delta \hookrightarrow \delta'$ and $\sigma_0 =_{low} \sigma_1$ and $\delta \subset initialized(\sigma_0) \cup failed(\sigma_0)$ and $\langle \sigma_0, s \rangle \downarrow \sigma_0'$ and $\langle \sigma_1, s \rangle \downarrow \sigma_1'$, then $\delta' \subseteq initialized(\sigma_0') \cup failed(\sigma_0')$ and $\sigma_0' =_{low} \sigma_1'$.*

Corollary 1. *If $pc \vdash s : \emptyset \hookrightarrow \delta$ then s satisfies noninterference.*

$$\frac{}{pc \vdash \mathsf{skip} : \delta \hookrightarrow \delta} \quad \frac{pc \vdash e : \delta \hookrightarrow \delta' \quad \Gamma(e) \sqsubseteq \Gamma(x) \quad pc \sqsubseteq \Gamma(x)}{pc \vdash x := e : \delta \hookrightarrow \delta'}$$

$$\frac{low \vdash e : \delta \hookrightarrow \delta' \quad \Gamma(e) \sqsubseteq \Gamma(C.f)}{low \vdash C.f := e : \delta \hookrightarrow \delta' \cup \{C\}}$$

$$\frac{C \in \delta \quad high \vdash e : \delta \hookrightarrow \delta' \quad \Gamma(e) \sqsubseteq \Gamma(C.f) \quad high \sqsubseteq \Gamma(C.f)}{high \vdash C.f := e : \delta \hookrightarrow \delta'}$$

$$\frac{pc \vdash s_0 : \delta_0 \hookrightarrow \delta_1 \quad pc \vdash s_1 : \delta_1 \hookrightarrow \delta_2}{pc \vdash s_0; s_1 : \delta_0 \hookrightarrow \delta_2}$$

$$\frac{pc \vdash e : \delta \hookrightarrow \delta' \quad pc \sqcup \Gamma(e) \vdash s_t : \delta' \hookrightarrow \delta_0 \quad pc \sqcup \Gamma(e) \vdash s_f : \delta' \hookrightarrow \delta_1}{pc \vdash \mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : \delta \hookrightarrow \delta_0 \cap \delta_1}$$

$$\frac{pc \vdash e : \delta \hookrightarrow \delta' \quad pc \sqcup \Gamma(e) \vdash s_t : \delta' \hookrightarrow \delta''}{pc \vdash \mathsf{while}\ e\ \mathsf{do}\ s_t : \delta \hookrightarrow \delta'}$$

Fig. 5. Typing of statements

5 Exception Handling

$$\frac{}{\langle \sigma, \mathsf{try\ skip\ catch}\ s\rangle \to \langle \sigma, \mathsf{skip}\rangle}$$

$$\frac{\langle \sigma, s_0\rangle \to \langle \sigma', s_0'\rangle}{\langle \sigma, \mathsf{try}\ s_0\ \mathsf{catch}\ s_1\rangle \to \langle \sigma', \mathsf{try}\ s_0'\ \mathsf{catch}\ s_1\rangle}$$

$$\frac{\langle \sigma, s_0\rangle \to \langle \sigma', \bullet\rangle}{\langle \sigma, \mathsf{try}\ s_0\ \mathsf{catch}\ s_1\rangle \to \langle \sigma', s_1\rangle}$$

Fig. 6. Operational semantics for exception handling

$$\frac{}{pc \vdash n : \delta \hookrightarrow \delta :: low} \quad \frac{}{pc \vdash x : \delta \hookrightarrow \delta :: low}$$

$$\frac{pc \vdash e_0 : \delta \hookrightarrow \delta_0 :: \ell_0 \quad pc \vdash e_1 : \delta \hookrightarrow \delta_1 :: \ell_1}{pc \vdash e_0\ op\ e_1 : \delta \hookrightarrow \delta_0 \cup \delta_1 :: pc \sqcup \ell_0 \sqcup \ell_1 \sqcup \Gamma(e_0) \sqcup \Gamma(e_1)}$$

$$\frac{}{low \vdash C.f : \delta \hookrightarrow \delta \cup \{C\} :: \Gamma(C)}$$

$$\frac{C \in \delta}{high \vdash C.f : \delta \hookrightarrow \delta :: high}$$

Fig. 7. Typing of expressions for exception handling

This section extends our system with an exception handling mechanism. The Java virtual machine throws an object that is an instance of a subclass of *LinkageError* when a loading, linkage, preparation, verification or initialization error occurs [GJSB96, Ch. 11]. *LinkageError* is a subclass of *Error*, rather than *Exception*. *Error* is designed in principle to indicate serious problems, and ordinary applications, such as library programs, are not expected to catch $Error$. However, as we argued in Section 2, there are several scenarios where catching *Error* is desirable such as in server applications to avoid crashing the entire system or to log messages. Therefore, we are motivated to develop a security type system which allows errors due to class initialization to be caught, while rejecting attacks that leak information through exception handling.

5.1 Operational semantics

We extend the syntax of the statements with $\mathsf{try}\ s_0\ \mathsf{catch}\ s_1$, whose operational semantics is given in Fig. 6.

$$\frac{}{pc \vdash \mathsf{skip} : \delta \hookrightarrow \delta :: low}$$

$$\frac{pc \vdash e : \delta \hookrightarrow \delta' :: \ell \quad \Gamma(e) \sqsubseteq \Gamma(x) \quad pc \sqsubseteq \Gamma(x)}{pc \vdash x := e : \delta \hookrightarrow \delta :: \ell}$$

$$\frac{low \vdash e : \delta \hookrightarrow \delta' :: \ell \quad \Gamma(e) \sqsubseteq \Gamma(C.f)}{low \vdash C.f := e : \delta \hookrightarrow \delta' \cup \{C\} :: \ell \sqcup \Gamma(C)}$$

$$\frac{C \in \delta \quad high \vdash e : \delta \hookrightarrow \delta' :: \ell \quad \Gamma(e) \sqsubseteq \Gamma(C.f) \quad high \sqsubseteq \Gamma(C.f)}{high \vdash C.f := e : \delta \hookrightarrow \delta' :: high}$$

$$\frac{pc \vdash s_0 : \delta_0 \hookrightarrow \delta_1 :: \ell_0 \quad pc \sqcup \ell_0 \vdash s_1 : \delta_1 \hookrightarrow \delta_2 :: \ell_1}{pc \vdash s_0; s_1 : \delta_0 \hookrightarrow \delta_2 :: \ell_0 \sqcup \ell_1}$$

$$\frac{\begin{array}{c} pc \vdash e : \delta \hookrightarrow \delta' :: \ell \\ \Gamma(e) \sqcup \ell \sqcup pc \vdash s_t : \delta' \hookrightarrow \delta_0 :: \ell_0 \quad \Gamma(e) \sqcup \ell \sqcup pc \vdash s_f : \delta' \hookrightarrow \delta_1 :: \ell_1 \end{array}}{pc \vdash \mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : \delta \hookrightarrow \delta_0 \cap \delta_1 :: \ell \sqcup \ell_0 \sqcup \ell_1}$$

$$\frac{pc \vdash e : \delta \hookrightarrow \delta' :: \ell \quad \Gamma(e) \sqcup \ell \sqcup pc \vdash s_t : \delta' \hookrightarrow \delta'' :: \ell'}{pc \vdash \mathsf{while}\ e\ \mathsf{do}\ s_t : \delta \hookrightarrow \delta' :: \ell \sqcup \ell'}$$

$$\frac{pc \vdash s_0 : \delta \hookrightarrow \delta' :: \ell_0 \quad pc \sqcup \ell_0 \vdash s_1 : \delta \hookrightarrow \delta' :: \ell_1}{pc \vdash \mathsf{try}\ s_0\ \mathsf{catch}\ s_1 : \delta \hookrightarrow \delta' :: \ell_1}$$

Fig. 8. Typing of statements for exception handling

5.2 Type system

Fig. 7 and Fig. 8 give typing rules for expressions and statements, respectively. We extend the type environment Γ to map class names to the security levels of the exceptions that may be raised during initialization. We have to adjust the definition of well-formed type environments:

Definition 3. *A type environment Γ is well-formed with respect to a class table CT if, for any class name C and a field name f of C such that $CT(C) = \mathsf{class}\ C\ \{\ldots, f = e, \ldots\}$, we have*

- $\Gamma(e) \sqsubseteq \Gamma(C, f)$*;*
- *if* $low \vdash e : \delta \hookrightarrow \delta' :: \ell$ *then* $\ell \sqsubseteq \Gamma(C)$*.*

We note that initialization failure of a class having high fields may be low and a class having only low fields may be high, as the following example illustrates:

$$\begin{array}{l} \mathsf{class}\ C_0\ \{\ g = 4,\ f = 1\ op\ 0\ \} \\ \mathsf{class}\ C_1\ \{\ g = 1\ op\ 0,\ f = 1\ \} \\ \mathsf{class}\ C_2\ \{\ f = C_1.f\ \} \end{array}$$

A type environment Γ such that $\Gamma(C_0) = \Gamma(C_0, f) = \Gamma(C_1, f) = \Gamma(C_2, f) = low$ and $\Gamma(C_1) = \Gamma(C_2) = \Gamma(C_0, g) = \Gamma(C_1, g) = high$ is well-formed with respect to the class table corresponding to the above class definitions.

Having noticed the above subtlety, the typing rules in Fig. 7 and Fig. 8 are straightforward adaptation from type systems that track information flow in the presence of exceptions (e.g., [Mye99,PS03,HS06,AS09]). The new form of judgment $pc \vdash e : \delta \hookrightarrow \delta' :: \ell$ for expressions (resp. $pc \vdash s : \delta \hookrightarrow \delta' :: \ell$ for statements) indicates the level ℓ of an exception that the expression e (resp. the statement s) may throw in the context pc.

For expression typing, $pc \vdash e : \delta \hookrightarrow \delta' :: low$ is derivable if either e does not throw an exception, or else pc is low and for any class C that occurs in e, $\Gamma(C) = low$ and, for any subexpression e' of e that is an operand of op, $\Gamma(e') = low$. Suppose $\Gamma(x) = high$, for instance, then $high \vdash x : \emptyset \hookrightarrow \emptyset :: low$ and $low \vdash x\ op\ n : \emptyset \hookrightarrow \emptyset :: high$ are derived. Notice that the security level of x is propagated to the security level of an exception only if the value of x may affect whether or not an exception is thrown.

We look at the inference rules for typing of statements. Since skip does not throw an exception, its exception level is low. The exception level of an assignment $x := e$ is that of the expression e at pc. For the exception level of an assignment to a field of a class, $C.f := e$ at low program context, we take the upper bound of the exception level of e at low and $\Gamma(C)$. More importantly, the exception level of $C.f := e$ in high program context is necessarily $high$, even if e does not throw an exception. This is because initialization of C may be triggered *and* fail, if it has previously failed, as illustrated by the following program, where f is low and g is high.

$$\mathsf{class}\ C\ \{\ f = 1/0,\ g = 1\ \}$$

$$\mathsf{try}\ C.f := 1\ \mathsf{catch\ skip};$$
$$\mathsf{try}\ (\mathsf{if}\ h = 0\ \mathsf{then}\ C.g := 0\ \mathsf{else\ skip})\ \mathsf{catch}\ l := 1$$

The above program is insecure and indeed rejected by our type system.

For the sequence statement, we prohibit s_1 from performing side effects lower than the level of exceptions that s_0 may throw. Rules for if- and while-statements are similar. The exception levels of the boolean guards are propagated to branches. In try-statement, the catch clause must not perform side effects lower than the exception level of the try block. Note that the exception level of the whole statement is ℓ_1, the exception level of the catch clause.

Proposition 1. *If $pc \vdash s : \emptyset \hookrightarrow \delta :: \ell$ then s satisfies noninterference.*

We now come back to the examples in Section 2. Since none of the class definitions from Section 2 involves high fields, they are typable with respect to an obvious type environment Γ mapping all the classes and class-field pairs to low. We keep the convention that $\Gamma(l) = low$ and $\Gamma(h) = high$. All the programs but P_1 are not typable. For P_1, we have $low \vdash P_1 : \emptyset \hookrightarrow \{D\} :: high$. Note that the programs P_0, P_1 and P_2 satisfy noninterference, but P_3, P_4 and P_5 do not (assuming the operational semantics is appropriately extended to take class hierarchy into account.)

6 Related Work

A survey [SM03] on language-based information-flow security contains an overview of the area. Most related to ours is work on tracking information flow in object-oriented languages and on information-flow controls in the presence of exceptions.

Objects To the best of our knowledge, the only information-flow mechanism that addresses class initialization is the one implemented by Jif [Mye99,MZZ+10], a compiler for Java extended with security types. As discussed earlier, Jif is rather conservative about class initialization code. This code is restricted to simple constant manipulation that may not raise any exceptions. As mentioned earlier, sometimes it is desirable to lift these restrictions.

Much other work has been done on information-flow security for object-oriented languages. Although none of the approaches directly addresses problems with class initialization, we nevertheless discuss some recent highlights.

Barthe and Serpette [BS99] present a type system for enforcing information-flow security in a simple object-oriented language based on the Abadi–Cardelli functional object calculi [AC96]. Bieber et al. [BCG+02] apply model-checking for securing information flow in smartcard applets.

Avvenuti et al. [ABF03] suggest an information-flow analysis in a language similar to Java bytecode. Bernardeschi et al. [BFLM05] check information-flow security in Java bytecode by a combination of program transformation and bytecode verification. These two approaches assume fixed security levels for classes. This might not be a flexible choice since it forces all instances and attributes to conform to the class level. Another concern is the scalability of this choice in presence of inheritance.

Banerjee and Naumann [BN05] show how to guarantee noninterference by a type-based analysis for a Java-like imperative language with objects. Amtoft et al. [ABB06] present a flow-sensitive logic for reasoning about information flow in the presence of pointers. Naumann [Nau06] investigates invariant-based verification of information-flow properties in a language with heaps. Barthe and Rezk [BR05] consider type-based enforcement of secure information flow in Java bytecode-like languages. Barthe et al. [BRN06] extend this work to derive an information-flow certifying compiler for a Java-like language.

Hammer and Snelting [HS09] develop a flow-sensitive, context-sensitive, and object-sensitive framework for controlling information flow by program dependence graphs. This approach takes advantage of similarities of information-flow and slicing analyses.

Exceptions As noted earlier, our treatment of exception handling draws on standard approaches from the literature (which we extend with the must-analysis). The intuition is if an occurrence of an exception in a statement may carry sensitive information, then there must be no publicly-observable side effects in either the code that handles the exception or in the code between the statement and the exception-handling block. Jif [Mye99,MZZ+10] implements such a discipline. Based on a similar discipline, Pottier and Simonet [PS03] propose a sound treatment of exceptions for ML.

Barthe and Rezk [BR05] treat a single type of exceptions in a JVM-like language. Barthe et al. [BPR07] extend this approach to multiple types of catchable exceptions. Connecting this with security-type preserving compilation, Barthe et al. [BRN06] show how to securely compile a source language with a single type of catchable exceptions to the low-level language of Barthe and Rezk [BR05].

Hedin and Sands [HS06] prove a noninterference property for a type system that tracks information flow via class-cast and null-pointer exceptions in a language with non-opaque pointers. Askarov and Sabelfeld [AS09] show how to achieve permissive

yet secure exception handling by providing the choice for each type of exception: either the traditional discipline discussed above or by consistently disallowing to catch exceptions. The actual choice for each kind of exception is given to the programmer.

7 Conclusion

Seeking to shed light on a largely unexplored area, we have presented considerations for and a formalization of secure class initialization. Our considerations highlight that class initialization poses challenges for security since controlling (the order of) side effects performed by class initialization is challenging. Hence, great care needs to be taken by information-flow enforcement mechanisms to guarantee security. One path, taken by Jif [Mye99,MZZ+10], is to severely restrict class initialization code so that it may only manipulate constants in an exception-free manner. Arguing that it is sometimes too restrictive, we have explored another path: allow powerful initialization code, but disallow class initialization inside conditionals and loops that branch on secret data. This approach has the advantage that the side effects in class initialization do not have to be predicted since they may not carry sensitive information in the first place: the attacker may not deduce anything interesting from observing these side effects anyway.

Our formalization demonstrates the idea by a type-and-effect system for a simple language that enforces noninterference. To the best of our knowledge, it is the first formal approach to the problem of secure class initialization. (Soundness of Jif's class initialization is yet to be established.)

Future work includes an extension to handle class hierarchies. We believe our approach of ruling out class initialization in high contexts is sound in the presence of class hierarchies. To extend our technical results to class hierarchies, we only need to adjust the operational semantics so that when a class is initialized, all its (uninitialized) super classes are initialized. Based on the results of the paper, we are currently working on more sophisticated type systems that allow initialization of *high* classes in high contexts.

Acknowledgments K. Nakata acknowledges the support of action IC0701 of COST, the Estonian Centre of Excellence in Computer Science, EXCS, financed mainly by ERDF, and the Estonian Science Foundation grant no. 6940. A. Sabelfeld is supported by the Swedish research agencies SSF and VR.

References

[ABB06] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 91–102, 2006.

[ABF03] M. Avvenuti, C. Bernardeschi, and N. De Francesco. Java bytecode verification for secure information flow. *SIGPLAN Notices*, 38(12):20–27, 2003.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, 1996.

[AHSS08] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. European Symp. on Research in Computer Security*, volume 5283 of *LNCS*, pages 333–348. Springer-Verlag, October 2008.

[AS09] A. Askarov and A. Sabelfeld. Catch me if you can: Permissive yet secure error handling. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.

[BCG+02] P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, and G. Zanon. Checking secure interactions of smart card applets: extended version. *J. Computer Security*, 10(4):369–398, 2002.

[BFLM05] C. Bernardeschi, N. De Francesco, G. Lettieri, and L. Martini. Checking secure information flow in java bytecode by code transformation and standard bytecode verification. *Software: Practice and Experience*, 34:1225–1255, 2005.

[BN05] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, March 2005.

[BPR07] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. In *Proc. European Symp. on Programming*, LNCS. Springer-Verlag, 2007.

[BR05] G. Barthe and T. Rezk. Non-interference for a jvm-like language. In *Proc. Types in Language Design and Implementation*, pages 103–112, 2005.

[BRN06] G. Barthe, T. Rezk, and D. Naumann. Deriving an information flow checker and certifying compiler for java. In *Proc. IEEE Symp. on Security and Privacy*, pages 230–242, 2006.

[BS99] G. Barthe and B. Serpette. Partial evaluation and non-interference for object calculi. In *Proc. FLOPS*, volume 1722 of *LNCS*, pages 53–67. Springer-Verlag, November 1999.

[Cro09] D. Crockford. Making javascript safe for advertising. adsafe.org, 2009.

[DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[Den76] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.

[Exc] Excalibur. Documentation and Software available at `http://excalibur.apache.org/index.html`.

[Fac09] Facebook. FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`, 2009.

[GJSB96] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification*. Addison-Wesley, 1996.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

[HS06] D. Hedin and D. Sands. Noninterference in the presence of non-opaque pointers. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, 2006.

[HS09] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive informationflow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, December 2009. Supersedes ISSSE and ISoLA 2006.

[Koz99] D. Kozen. Language-based security. In *Proc. Mathematical Foundations of Computer Science*, volume 1672 of *LNCS*, pages 284–298. Springer-Verlag, September 1999.

[LB98] S. Liang and G. Bracha. Dynamics class loading in the Java virtual machine. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 36–44, 1998.

[Ler03] X. Leroy. Java bytecode verification: algorithms and formalizations. *J. Automated Reasoning*, 30(3–4):235–269, 2003.

[LY99] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, second edition edition, 1999.

[MSL+08] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript, 2008.

[Mye99] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.

[MZZ+10] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, 2001–2010.

[Nau06] D. Naumann. From coupling relations to mated invariants for checking information flow. In *Proc. European Symp. on Research in Computer Security*, pages 279–296. Springer-Verlag, 2006.

[PS03] F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, January 2003.

[Sim03] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml`, July 2003.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[SMH00] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 86–101. Springer-Verlag, 2000.

[Sun] Java 2 platform, standard edition 5.0, API specification. Available at `http://java.sun.com/j2se/1.5.0/docs/api/`.

[Sys10] Praxis High Integrity Systems. Sparkada examinar. Software release. `http://www.praxis-his.com/sparkada`, 2010.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[WAF00] D. S. Wallach, A. W. Appel, and E. W. Felten. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, October 2000.