Securing Class Initialization in Java-like Languages

Willard Rafnsson, Keiko Nakata, and Andrei Sabelfeld

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, little attention has been given to the impact of class initialization on information flow. This paper turns the spotlight on security implications of class initialization. We reveal the subtleties of information propagation when classes are initialized, and demonstrate how these flows can be exploited to leak information through error recovery. Our main contribution is a type-and-effect system which tracks these information flows. The type system is parameterized by an arbitrary lattice of security levels. Flows through the class hierarchy and dependencies in field initializers are tracked by typing class initializers wherever they could be executed. The contexts in which each class can be initialized is tracked to prevent insecure flows of out-of-scope contextual information through class initialization statuses and error recovery. We show that the type system enforces termination-insensitive noninterference.

Index Terms—Information Flow Control, Program Analysis.

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: i) Java stack inspection [WAF00], which enforces a stack-based access-control discipline, ii) Java bytecode verification [Ler03], which traverses bytecode to verify type safety, and iii) web languages such as Caja [MSL+08], ADsafe [Cro09], and FBJS [Fac09], which use program transformation and language subsets 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, as these can 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 upon first active use [LY99][1]. 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 linking – 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 (not) been 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 the initialization (or success thereof) of a class containing public fields in no way depends on the evaluation of an expression (or success thereof) containing secret data, the type-and-effect system guarantees security in a form of *noninterference* [GM82]. Informally, noninterference guarantees that a program's public out-

- *Willard Rafnsson and Andrei Sabelfeld are with the Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden.*
- *Keiko Nakata is with the Institute of Cybernetics, Tallinn University of Technology, Tallinn, Estonia.*

1. 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.

puts are independent of secret inputs. A key intricacy here is that of class dependencies: An initialization of one class can cause the initialization of other classes. 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 initialization code are rather severe: only simple constant manipulations, which cannot raise exceptions, are allowed. 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 desirable in scenarios such as server-side code.

2 Background

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

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

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

$$\begin{array}{l}C.y := 0;\\ \texttt{if}\ h \neq 0\ \texttt{then new}\ D\ \texttt{else skip}\end{array}$$

Except when h has the value 0 initially, the above program results in an error, since an initialization of class D is attempted. The object creation, should it occur, is the first active use of D. This triggers initialization of D. When a class is initialized, all its field - (field-)initializer assignments are executed in the state in which the first active use of the class occurred. Here, $D.x := 1/C.y$ is performed in a state where $C.y = 0$, so a division by 0 occurs, producing an error. Note that in the terminology we have introduced, the initialization occurs in a 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:

$$\begin{array}{l}\texttt{new}\ D;\\ C.y := 0;\\ \texttt{if}\ h \neq 0\ \texttt{then new}\ D\ \texttt{else skip}\end{array}$$

In this latter program, D is initialized before it is used in a high context, so running the second `new` D statement does not incur any initialization activities.

In Java, when initialization of a class has completed abnormally, an exception is thrown and the class is marked as erroneous. Initialization of a class in an erroneous state is not possible [LY99, Ch. 2][2]. This makes initialization failure persistent throughout a run in the sense that when initialization of a class failed on it first (active) use, then it will fail on any future use irrespective of the state in which the second initialization is attempted[3]. Catching initialization errors introduces a delicate scenario of information leaks. For instance, consider the following program:

$$\begin{array}{l}C.y := 0;\\ \texttt{if}\ h \neq 0\ \texttt{then}\ (\texttt{try new}\ D\ \texttt{catch skip})\ \texttt{else skip};\\ C.y := 1;\\ \texttt{new}\ D\end{array}$$

Again, the above program results in an error except when h has the value 0 initially. In effect, information from h flows out of the scope of the `if`, through the initialization status of D, into the termination behaviour of the last statement. 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, Program p_{main}, given below, illustrates the subtlety of the problem:

$$\begin{array}{l}C.y := 0;\\ \texttt{if}\ h \neq 0\ \texttt{then}\ (\texttt{try new}\ D\ \texttt{catch skip})\ \texttt{else skip};\\ C.y := 1; l := 0;\\ \texttt{try new}\ D\ \texttt{catch}\ l := 1\end{array}$$

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

Leaks like these can easily occur in practise; consider the Java program in Figure 1. Here, class D, originating from [Bil], implements the Singleton design pattern through the "initialization on demand holder" idiom, which is thread-safe (as opposed to using double-checked locking [Sch97]). To achieve thread-safety, this implementation leaves it to the class loader to construct the instance of D by calling the constructor in a field initializer. Since the class initialization phase is guaranteed to be serial [GJS96, Section 12.4], no race conditions occur, so only one instance of D is ever created. However, the class dependencies in this program are exactly the same as those in p_{main}, so in exactly the same way

2. To be precise, the *Class* object representing the class is labeled as erroneous.

3. 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.

Source File

```
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class SingletonExample{
  static class D{                    // Singleton
    private static final D INST = new D();
    public int x;
    public D(){
      x = 1 / C.y;                   // fails if C.y=0
    }
    public static D getInstance(){
      return INST;
    }
  }
  static class C{
    public static int y = 1;
  }
  public static void main(String[] a)
   throws IOException{
    int l = 0;
    System.out.print("h = ");       // input (high)
    BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
    int h = Integer.parseInt(in.readLine());
    C.y = 0;
    if (h == 0){
      try{
        D.getInstance();           // fails if h=0
      } catch (LinkageError e){}
    }
    C.y = 1;
    try{
      D.getInstance();             // fails if h=0
    } catch(LinkageError e){
      l = 1;                       // run   if h=0
    }
    System.out.println("l = " + l);// output (low)
  }                                  // 1 iff h = 0
}
```

Program Execution

```
$ javac SingletonExample.java

$ java SingletonExample
h = 0
l = 1

$ java SingletonExample
h = 42
l = 0
```

Fig. 1: Singleton Example in Java

that one bit of the initial value of h is reflected in l after execution of p_{main}, one bit of the high input is reflected in the low output of the program in Figure 1.

Dependencies in class definitions can impact security: before a class C is initialized, for each field and initializer $C.x := e$ in C, for each $D.y$ occurring in e, D is initialized. We say C depends on all these classes. If, when initializing C, initialization of any class D which C depends on fails, then initialization of C fails. For instance, consider the class definitions below, involving only low fields.

$$\begin{array}{l}\mathsf{class}\ C\ \{\ y = 1\ \}\\ \mathsf{class}\ D_0\ \{\ x = 1/C.y\ \}\\ \mathsf{class}\ D_1\ \{\ x = D_0.x\ \}\end{array}$$

p_{dep}, given below, always terminates normally and reflects secret input values in public results.

$$\begin{array}{l}C.y := 0;\\ \texttt{if}\ h \neq 0\ \texttt{then}\ (\texttt{try new}\ D_0\ \texttt{catch skip})\ \texttt{else skip};\\ C.y := 1; l := 0;\\ \texttt{try new}\ D_1\ \texttt{catch}\ l := 1\end{array}$$

Class hierarchies impact security as well, since if a class C is a subclass of a class D, then C depends on D. So replacing the definition of D_1 with the following definition of D_1 in the class table above yields the same insecure flow in Program p_{dep}.

$$\mathsf{class}\ D_1\ \mathsf{extends}\ D_0\ \{\}$$

The bottom line is that class initialization may perform side effects, causing information to leak. Languages which lazily intialize static classes, and where initialization failure is persistent, have this information channel. This includes Java (as seen above) and C# (example in the associated technical report [RNS11]), but excludes C++ (no non-constant field initializers), VB.NET (no static class fields), Smalltalk (no field initializers), Ruby and Python (classes are objects, and failure is not persistent). 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. This makes the particular implementation of the Singleton pattern given in Figure 1 impossible in Jif. While it is rarely good practice to catch errors within ordinary methods, such as methods in libraries, there are several scenarios where this is good practice, e.g. in server applications to avoid crashing the entire system due to third party applications or to log messages. An example from practice where errors are caught and rethrown as exceptions can be found in [Exc].

This paper proposes and formalizes a different approach: we allow side effects during initialization, as long as these do not cause insecure flows. This paper extends and improves an earlier conference version [NS10]. The enforcement mechanism in the conference version disallows class initializations in secret contexts altogether, does not consider class dependencies, and uses a fixed lattice of security levels. These limitations are resolved in the present version: we present a more permissive type-and-effect system which is furthermore parameterized by an arbitrary lattice of security levels. We track flows through dependencies in the class hierarchy and in field initializers, and track every context in which each class can be initialized in, to prevent the kind

of leak demonstrated in program p_{main}. Finally, we prove that our type system soundly enforces a termination-insensitive notion of noninterference (TINI).

Section 5 develops a type-and-effect system for tracking information flows in a simple language with classes, defined in Section 3 (class hierarchies added in Section 6). We show in Section 7 that the type-and-effect system enforces TINI, given in Section 4. Sections 8 discusses related work, and Section 9 concludes.

3 LANGUAGE

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

$$
\begin{array}{lll}
\text{Expression} & e & ::= n \mid x \mid e \oplus e \mid C.x \\
\text{Statement} & s & ::= \texttt{skip} \mid s; s \mid x := e \mid C.x := e \\
 & & \quad \mid \texttt{if } e \texttt{ then } s \texttt{ else } s \\
 & & \quad \mid \texttt{try } s \texttt{ catch } s \mid \texttt{while } e \texttt{ do } s \\
\text{Class definition} & c & ::= C\,\{i\} \mid C <: C\,\{i\} \\
\text{Field definitions} & i & ::= (x = e)^* \\
\text{Class table} & \tau & ::= c^*
\end{array}
$$

Metavariables x, n, $\oplus$ and C range over variables, integers, operators and class names respectively. Notation $C.x$ denotes field x of class C. Compound expressions are built using binary (integer arithmetic) operators, ranged by metavariable $\oplus$. We assume a collection of such operators, partitioned into partial and total operators, ranged by $\oplus_{\mathsf{P}}$ and $\oplus_{\mathsf{T}}$ respectively. Total, resp. partial, operators are represented as functions mapping into $\mathbb{Z}$, resp. $\mathbb{Z} \cup \{\bullet\}$, if both operands are integers, and into $\{\bullet\}$ if either operand is $\bullet$. The case $n_1 \oplus_{\mathsf{P}} n_2 = \bullet$ represents the scenario where $\oplus_{\mathsf{P}}$ is undefined on (n_1, n_2). So, here, $\bullet$ denotes an evaluation *error* as a result of an undefined partial operator application. The notation $(x = e)^*$ denotes a possibly empty sequence of $x = e$. The notation c^* is similar. Our class definition notation is a compacted version of the one seen in the previous section; $C\,\{x_0 = e_0 \ldots x_k = e_k\}$ declares a class named C consisting of the fields x_i and field initializers e_i. Instead of modeling arbitrary statement execution during class initialization, we include only the minimum constructs required for flows through class initialization statuses to occur. We assume $i \neq j \implies x_i \not\equiv x_j$ in the following. When C extends C' (C a subclass of C', C inherits from C'), we instead write $C <: C'\,\{x_0 = e_0 \ldots x_k = e_k\}$ ($<:$ is the subtype relation). We sometimes write $x_0 = e_0 \ldots x_k = e_k$ as $x_0 = e_0; \ldots; x_k = e_k$ when this helps readability. If $i = \epsilon$ in $C\,\{i\}$ and $C <: C'\,\{i\}$, then C has no fields. We write these definitions as $C\,\{\,\}$ and $C <: C'\,\{\,\}$. A class table τ is a (finite) list of class definitions. We interpret τ as a function mapping a class name C to the class definition in τ named C. Formally, the function is defined by induction on τ as follows.

$$
(c\tau)(C) = \begin{cases} c & \text{if } \exists C', i \,.\, c \in \{C\,\{i\}, C <: C'\,\{i\}\}, \\ \tau(C) & \text{otherwise.} \end{cases}
$$

A program is a pair (τ, s) of a class table and a statement. We let P range over programs. To lighten notation in our formal study, we assume a fixed τ hereafter.

The following syntactic categories, not part of the language syntax, arise during program execution.

$$
\begin{array}{lll}
\text{Initialization status} & S & ::= \textsc{u} \mid \textsc{b} \mid \textsc{i} \mid \bullet \\
\text{Initialization result} & I & ::= \textsc{b} \mid \textsc{i} \mid \bullet \\
\text{Value} & V & ::= n \mid \bullet \\
\text{Termination result} & T & ::= \texttt{skip} \mid \bullet
\end{array}
$$

Programs operate on a *state* (a.k.a. *store*, or *memory*). A state, ranged over by σ, maps variables and fields to integers, and class names to *(class) initialization statuses*. A class initialization status S denotes the loaded status of a class named C in a state σ: C is uninitialized in σ when $\sigma(C) = \textsc{u}$; C is being initialized when $\sigma(C) = \textsc{b}$, and is initialized successfully when $\sigma(C) = \textsc{i}$. C has failed to initialize when $\sigma(C) = \bullet$. We use the following notation for updating the value of x in σ, and analogous notation for a $C.x \mapsto n$ and a $C \mapsto S$ update.

$$
\sigma[x \mapsto n](x') = \begin{cases} n & \text{, if, } x \equiv x' \\ \sigma(x') & \text{, otherwise} \end{cases}
$$

We use $\_$ as a wildcard, that is, any mathematical object, which we do not intend to refer to[4]. The (big-step) operational semantics of our language is given by relations of the form $\langle \sigma, \_ \rangle \Rightarrow \langle \sigma', \_ \rangle$. Here, σ is the state before evaluation, the former $\_$ is that which is evaluated under σ, the latter $\_$ is the result of the evaluation, and σ' is the resulting state. Evaluation of expressions is given in Figure 2 (a). The relation $\langle \sigma, e \rangle \Rightarrow \langle \sigma', V \rangle$ states that the expression e in the state σ evaluates to the *value* V with final state σ'. If $V = \bullet$, an error occurred during the evaluation of e under σ. Otherwise, e evaluated successfully, in which case $V = n$ for some n. The inference rules in Figure 2 (a) are straightforward except those for reading from a field of a class, $C.x$. Both read and write access to a field of a class C triggers initialization of C. That is, the definition of C, $\tau(C)$, is evaluated under σ using the rules in Figure 2 (b), using relation $\langle \sigma, c \rangle \Rightarrow \langle \sigma', I \rangle$ with $c = \tau(C) = C\,\{i\}$, which states that class definition c in state σ evaluates to *initialization result* I with final state σ'. If $\sigma(C) \neq \textsc{u}$, then initialization of C has already been triggered, and possibly failed, so $I = \sigma(C)$ and $\sigma' = \sigma$. If $\sigma(C) = \textsc{u}$, then C needs to be initialized. This is done by executing the field definitions of C as assignments. We refer to this code as the *(class) initializer of* C. Given field definitions i of C, we define the initializer of C, $\mathsf{s}(C, i)$, as follows.

$$
\begin{aligned}
\mathsf{s}(C, \epsilon) &= \texttt{skip} \\
\mathsf{s}(C, (x = e)i) &= C.x := e; \mathsf{s}(C, i)
\end{aligned}
$$

So, to initialize C, we execute $\mathsf{s}(C, i)$ under σ with initialization status of C set to $\textsc{b}$, using relation $\langle \sigma, s \rangle \Rightarrow \langle \sigma', T \rangle$ with $s = \mathsf{s}(C, i)$, which states that statement s in state σ

4. Multiple occurrences of $\_$ in the same context can represent different mathematical objects.

$$\textsc{Num}_{\Rightarrow}\ \frac{-}{\langle \sigma, n\rangle \Rightarrow \langle \sigma, n\rangle} \qquad \textsc{Var}_{\Rightarrow}\ \frac{-}{\langle \sigma, x\rangle \Rightarrow \langle \sigma, \sigma(x)\rangle}$$

$$\textsc{Field-E}_{\Rightarrow}\ \frac{\langle \sigma, \tau(C)\rangle \Rightarrow \langle \sigma', \bullet\rangle}{\langle \sigma, C.x\rangle \Rightarrow \langle \sigma', \bullet\rangle}$$

$$\textsc{Field-OK}_{\Rightarrow}\ \frac{\langle \sigma, \tau(C)\rangle \Rightarrow \langle \sigma', I\rangle \quad I \in \{\text{I}, \text{B}\}}{\langle \sigma, C.x\rangle \Rightarrow \langle \sigma', \sigma'(C.x)\rangle}$$

$$\textsc{Op-EL}_{\Rightarrow}\ \frac{\langle \sigma, e_1\rangle \Rightarrow \langle \sigma', \bullet\rangle}{\langle \sigma, e_1 \oplus e_2\rangle \Rightarrow \langle \sigma', \bullet\rangle}$$

$$\textsc{Op-ER}_{\Rightarrow}\ \frac{\langle \sigma, e_1\rangle \Rightarrow \langle \sigma', n\rangle \quad \langle \sigma', e_2\rangle \Rightarrow \langle \sigma'', \bullet\rangle}{\langle \sigma, e_1 \oplus e_2\rangle \Rightarrow \langle \sigma'', \bullet\rangle}$$

$$\textsc{Op-EP}_{\Rightarrow}\ \frac{\langle \sigma, e_1\rangle \Rightarrow \langle \sigma', n_1\rangle \quad \langle \sigma', e_2\rangle \Rightarrow \langle \sigma'', n_2\rangle \quad n_1 \oplus_{\text{P}} n_2 = \bullet}{\langle \sigma, e_1 \oplus_{\text{P}} e_2\rangle \Rightarrow \langle \sigma'', \bullet\rangle}$$

$$\textsc{Op-OK}_{\Rightarrow}\ \frac{\langle \sigma, e_1\rangle \Rightarrow \langle \sigma', n_1\rangle \quad \langle \sigma', e_2\rangle \Rightarrow \langle \sigma'', n_2\rangle \quad n_1 \oplus n_2 = n}{\langle \sigma, e_1 \oplus e_2\rangle \Rightarrow \langle \sigma'', n\rangle}$$

(a) Expressions

$$\textsc{Init-A}_{\Rightarrow}\ \frac{\sigma(C) \neq \text{U}}{\langle \sigma, C\,\{i\}\rangle \Rightarrow \langle \sigma, \sigma(C)\rangle}$$

$$\textsc{Init-U}_{\Rightarrow}\ \frac{\sigma(C) = \text{U} \quad \langle \sigma[C \mapsto \text{B}], \mathsf{s}(C, i)\rangle \Rightarrow \langle \sigma', T\rangle}{\langle \sigma, C\,\{i\}\rangle \Rightarrow \langle \sigma'[C \mapsto I(T)], I(T)\rangle}$$

(b) Class Definitions

Fig. 2: Operational Semantics for Expression Evaluation

evaluates to *termination result* T with final state σ'. If $T = \bullet$, then an error occurred during the execution. Otherwise, $T = \texttt{skip}$, signifying that the execution was error-free. Once execution of i has terminated with termination result T, we set the initialization status of C in the resulting memory to $I(T)$, where

$$I(T) = \begin{cases} \text{I} & \text{, if } T = \texttt{skip}, \\ \bullet & \text{, otherwise } (T = \bullet). \end{cases}$$

If $I = \bullet$, then this means C failed to initialize (now or previously), so its field read will fail. Otherwise, the resulting value is the value $C.x$ has in the resulting memory. Note that for readability, we have postponed all treatment of class hierarchies to Section 6.

Execution of statements is given in Figure 3. We let $\bar{0}$ denote a non-0, non-$\bullet$ value. Again, the inference rules for the big-step reduction relation are straightforward, with the exception of the expression evaluation error rule (E-E$_{\Rightarrow}$) and the class field assignment rules (FIELD-A-E$_{\Rightarrow}$) and (FIELD-A-OK$_{\Rightarrow}$). For the former, $Q[\text{e}]$ specifies a grammar with a "hole" in it. That is, e is a formal parameter, and, for instance, $Q[4 + 5 * x]$ defines a grammar where e has been replaced by $4 + 5 * x$ (`while` $4 + 5 * x$ `do skip` is generated by said grammar). For the latter, note that an assignment to $C.x$ triggers initialization of C. To initialize C in rules (FIELD-A-E$_{\Rightarrow}$) and (FIELD-A-OK$_{\Rightarrow}$), we simply read $C.x$, since doing this causes C to be initialized as a side-effect.

4 Security

We now develop a security notion for our language, and give examples of programs that leak through class initialization statuses.

$$\textsc{Var-A}_{\Rightarrow}\ \frac{\langle \sigma, e\rangle \Rightarrow \langle \sigma', n\rangle}{\langle \sigma, x := e\rangle \Rightarrow \langle \sigma'[x \mapsto n], \texttt{skip}\rangle}$$

$$\text{E-E}_{\Rightarrow}\ \frac{\langle \sigma, e\rangle \Rightarrow \langle \sigma', \bullet\rangle}{\langle \sigma, Q[e]\rangle \Rightarrow \langle \sigma', \bullet\rangle}\ \text{where}\ \begin{array}{l} Q[\text{e}] ::= x := \text{e} \mid C.x := \text{e} \\ \qquad \mid \texttt{if e then } s_1 \texttt{ else } s_2 \\ \qquad \mid \texttt{while e do } s \end{array}$$

$$\textsc{Field-A-E}_{\Rightarrow}\ \frac{\langle \sigma, e\rangle \Rightarrow \langle \sigma', n\rangle \quad \langle \sigma', C.x\rangle \Rightarrow \langle \sigma'', \bullet\rangle}{\langle \sigma, C.x := e\rangle \Rightarrow \langle \sigma'', \bullet\rangle}$$

$$\textsc{Field-A-OK}_{\Rightarrow}\ \frac{\langle \sigma, e\rangle \Rightarrow \langle \sigma', n\rangle \quad \langle \sigma', C.x\rangle \Rightarrow \langle \sigma'', n'\rangle}{\langle \sigma, C.x := e\rangle \Rightarrow \langle \sigma''[C.x \mapsto n], \texttt{skip}\rangle}$$

$$\textsc{Seq-E}_{\Rightarrow}\ \frac{\langle \sigma, s_1\rangle \Rightarrow \langle \sigma', \bullet\rangle}{\langle \sigma, s_1; s_2\rangle \Rightarrow \langle \sigma', \bullet\rangle}$$

$$\textsc{Seq-OK}_{\Rightarrow}\ \frac{\langle \sigma, s_1\rangle \Rightarrow \langle \sigma', \texttt{skip}\rangle \quad \langle \sigma', s_2\rangle \Rightarrow \langle \sigma'', T\rangle}{\langle \sigma, s_1; s_2\rangle \Rightarrow \langle \sigma'', T\rangle}$$

$$\textsc{If-T}_{\Rightarrow}\ \frac{\langle \sigma, e\rangle \Rightarrow \langle \sigma', \bar{0}\rangle \quad \langle \sigma', s_1\rangle \Rightarrow \langle \sigma'', T\rangle}{\langle \sigma, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2\rangle \Rightarrow \langle \sigma'', T\rangle}$$

$$\textsc{If-F}_{\Rightarrow}\ \frac{\langle \sigma, e\rangle \Rightarrow \langle \sigma', 0\rangle \quad \langle \sigma', s_2\rangle \Rightarrow \langle \sigma'', T\rangle}{\langle \sigma, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2\rangle \Rightarrow \langle \sigma'', T\rangle}$$

$$\textsc{Try-E}_{\Rightarrow}\ \frac{\langle \sigma, s_1\rangle \Rightarrow \langle \sigma', \bullet\rangle \quad \langle \sigma', s_2\rangle \Rightarrow \langle \sigma'', T\rangle}{\langle \sigma, \texttt{try } s_1 \texttt{ catch } s_2\rangle \Rightarrow \langle \sigma'', T\rangle}$$

$$\textsc{Try-OK}_{\Rightarrow}\ \frac{\langle \sigma, s_1\rangle \Rightarrow \langle \sigma', \texttt{skip}\rangle}{\langle \sigma, \texttt{try } s_1 \texttt{ catch } s_2\rangle \Rightarrow \langle \sigma', \texttt{skip}\rangle}$$

$$\textsc{While-F}_{\Rightarrow}\ \frac{\langle \sigma, e\rangle \Rightarrow \langle \sigma', 0\rangle}{\langle \sigma, \texttt{while } e \texttt{ do } s\rangle \Rightarrow \langle \sigma', \texttt{skip}\rangle}$$

$$\textsc{While-T}_{\Rightarrow}\ \frac{\langle \sigma, e\rangle \Rightarrow \langle \sigma', \bar{0}\rangle \quad \langle \sigma', s; \texttt{while } e \texttt{ do } s\rangle \Rightarrow \langle \sigma'', T\rangle}{\langle \sigma, \texttt{while } e \texttt{ do } s\rangle \Rightarrow \langle \sigma'', T\rangle}$$

Fig. 3: Operational Semantics for Statement Execution

4.1 Lattices

We recap some basic definitions from order theory. A partially ordered set (poset) is a set A together with a binary relation $\sqsubseteq$ over A which is reflexive, antisymmetric and transitive. For $x, y, z \in A$, z is a join (least upper bound, lub) of x and y if $x \sqsubseteq z$, $y \sqsubseteq z$ and $\forall w \in A.\, x \sqsubseteq w \wedge y \sqsubseteq w \implies z \sqsubseteq w$. If a lub of x and y exists, it is (provably) unique, so we denote it $x \sqcup y$, thus defining a join operator $\sqcup$. When $x \sqcup y$ is defined $\forall x, y$, then poset A is a join-semi-lattice. For $x, y, z \in A$, z is a meet (greatest lower bound, glb) of x and y if $z \sqsubseteq x$, $z \sqsubseteq y$ and $\forall w \in A.\, w \sqsubseteq x \wedge w \sqsubseteq y \implies w \sqsubseteq z$. If a glb of x and y exists, it is (provably) unique, so we denote it $x \sqcap y$, thus defining a meet operator $\sqcap$. When $x \sqcap y$ is defined $\forall x, y$, then poset A is a meet-semi-lattice. A poset is a lattice iff it is a join- and a meet-semi-lattice.

4.2 Noninterference

We assume an arbitrary lattice $\mathcal{L}$ of security levels [Den76]. In our examples, we use a two-level lattice; *low* (public) and *high* (secret), with *low* $\sqsubseteq$ *high*. We let metavariables ℓ and pc range over security levels. We assume that each variable and class field has been assigned a fixed security level, and denote this mapping from variables and class fields to security levels by $|\text{v}|$. We extend this mapping to expressions by letting $|\text{v}|(e)$

denote the least upper bound of the security levels of all variables and class fields occurring in $\mathsf{lvl}(e)$. For notational simplicity, we assume a fixed lvl henceforth.

These security levels define *who* can observe *what*. An observer in our setting is assigned a security level expressing which variables and class fields we assume the observer has access to. If an observer has level ℓ, then the observer can read variables and class fields with level $\sqsubseteq \ell$, and write to these before the program is run. This gives rise to an (observational) equivalence on memories.

Definition 4.1 (l-equivalence)**.** *σ_1 and σ_2 are ℓ-equivalent, written $\sigma_1 =_\ell \sigma_2$, iff,*

$$\forall x.\mathsf{lvl}(x) \sqsubseteq \ell \implies \sigma_1(x) = \sigma_2(x) \quad (1)$$

$$\forall C.(\exists C.x\,.\,\mathsf{lvl}(C.x) \sqsubseteq \ell) \implies \sigma_1(C) = \sigma_2(C) \quad (2)$$

$$\forall C.x.\left(\begin{array}{c}\mathsf{lvl}(C.x) \sqsubseteq \ell \\ \wedge\ \sigma_i(C) \in \{\mathrm{I}, \mathrm{B}\}\end{array}\right) \implies \sigma_1(C.x) = \sigma_2(C.x) \quad (3)$$

It turns out $=_\ell$ is an equivalence relation, which will turn out to be useful later.

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]. An initial memory σ_{init} in our setting satisfies $\sigma_{\text{init}}(C) = \mathrm{U}$ for all C, and $\sigma_{\text{init}}(C.x) = 0$ for all $C.x$. In this sense, the "input" to a program is a value assignment to global variables.

Definition 4.2 (TINI)**.** *s satisfies* termination-insensitive noninterference *(TINI) if, for all ℓ and initial σ_1, σ_2 s.t. $\sigma_1 =_\ell \sigma_2$, if $\langle \sigma_1, s\rangle \Rightarrow \langle \sigma_1', \texttt{skip}\rangle$ and $\langle \sigma_2, s\rangle \Rightarrow \langle \sigma_2', \texttt{skip}\rangle$, then $\sigma_1' =_\ell \sigma_2'$.*

4.3 Running Examples

We now present challenges an enforcement mechanism of TINI must deal with, in form of example programs, some of which are insecure, others which are secure. While the programs are simple, their pattern can easily arise in larger programs, so a decent enforcement mechanism must reject the insecure programs, yet accept the secure programs, presented here. Our first program, p_{main}, is the main example from Section 2. Even with all fields in the class table labeled *high*, this program leaks an out-of-scope secret context by (ab)use of the `try-catch` language construct. One way to reject this program in an enforcement would be to "taint" C by the security level of all contexts which dereferencing of C occurs in. This approach would also reject p_{art}, obtained by replacing the definition of D with the following.

$$D\,\{x = 1 + C.y\}$$

This program is secure. However, for some memories, `try new` C `catch` $l := 1$ becomes insecure. Namely, the memories that state that C failed to initialize. However, these memories never arise when running p_{art} on an initial memory (C cannot fail to initialize), meaning we cannot universally and unconditionally quantify memories when proving soundness of the enforcement mechanism, as `art`ificial flows can then arise.

The next program, p_{LinH}, leaks since the point during control flow at which a class C with an observable field is initialized, depends on a secret, and observable parts of the memory change between these points.

$$\begin{array}{l} D.l := 0; \\ \texttt{if } h = 0 \texttt{ then new } C \texttt{ else skip}; \\ D.l := 1; \\ \texttt{if } C.l \texttt{ then } l := 1 \texttt{ else } l := 0; \end{array}$$

Class table of p_{LinH}:

$$\begin{array}{l} C\,\{l = 0 + D.l\} \\ D\,\{l = 0\} \end{array}$$

This program can be secured by injecting `new` C; just before the first `if` statement in p_{LinH}. We refer to this modified p_{LinH} as p'_{LinH}. The lesson here is that it is okay for a reference to a class C, containing a *low* field, to appear in a *high* context, as long as dereferencing C there cannot cause C to be initialized.

5 Enforcement

We now present a type system for enforcing noninterference. The type system formalizes a data-flow analysis for tracking information dependencies in a program. To guarantee secure class initialization, in addition to standard information flow tracking in imperative systems [DD77], [VSI96], [SM03], we need to track two things during evaluations: *i)* which classes are initialized, and *ii)* what information is leaked when such an action does (not) produce an error. To track *i)*, our enforcement performs a *must* analysis to soundly approximate at any program point which class must be (in the process of being) initialized. To track *ii)*, our enforcement maintains a security level for each class expressing information obtained by observing whether its initialization produced an error or not ($\bot$ for classes which cannot fail to initialize). The type system is syntax-directed, in the sense that the type rules specify how to compute the constituents of a type judgment. A type inference algorithm can thus easily be obtained from our type system.

As is standard [DD77], [VSI96], the type system (over)approximates information flows during operator applications and at join points, that is, at "if", "try" and "while" constructs. The result of an operator application contains information from both operands, a partial operator application is considered able to fail, any branch in an "if" is considered possible, any number of iterations of "while" are considered possible, a try-branch of a

"try" is considered able to fail at any point, and the execution of the catch-branch is considered possible.

5.1 Type Environment

The above-mentioned tracking is maintained by the *type environment* Γ. Let $\mathcal{L}_{I\setminus\bullet}$ denote the lattice given by $\mathtt{U} \sqsubseteq \mathtt{B}$ and $\mathtt{B} \sqsubseteq \mathtt{I}$, illustrated in Figure 4. An initialization status as a type indicates the highest guarantee, in the order U, B and I, of the initialization status of a class. Type U thus means "no guarantee". A type environment Γ consists of two mappings,

Fig. 4: $\mathcal{L}_{I\setminus\bullet}$.

$$\Gamma^{\mathsf{s}} : \mathsf{dom}(\tau) \to \mathcal{L}_{I\setminus\bullet} \qquad \Gamma^{\mathsf{e}} : \mathsf{dom}(\tau) \to \mathcal{L}.$$

Γ^{s} keeps track of the initialization status of classes. The second, Γ^{e}, tracks information conveyed by observing whether a previous initialization of a class yielded an error. The notation for updating a Γ is given below (note the first case in $\Gamma[C \mapsto^{\mathsf{e}} \ell]^{\mathsf{e}}(\hat{C})$).

$$\Gamma[C \mapsto^{\mathsf{s}} I]^{\mathsf{s}}(\hat{C}) = \begin{cases} I & \text{if } \hat{C} = C \\ \Gamma^{\mathsf{s}}(\hat{C}) & \text{otherwise.} \end{cases}$$

$$\Gamma[C \mapsto^{\mathsf{e}} \ell]^{\mathsf{e}}(\hat{C}) = \begin{cases} \Gamma^{\mathsf{e}}(\hat{C}) \sqcup \ell & \text{if } \hat{C} = C \\ \Gamma^{\mathsf{e}}(\hat{C}) & \text{otherwise.} \end{cases}$$

For Γ to be sound wrt. some term t, Γ^{s} must be an underapproximation, and Γ^{e} an overapproximation. Say you have Γ_1 and Γ_2 representing type environments for two different control flow paths in the evaluation of t. To obtain a Γ for the join point of these two paths, we set $\Gamma = \Gamma_1 \odot \Gamma_2$, with $\odot$ defined

$$(\Gamma_1 \odot \Gamma_2)^{\mathsf{s}}(C) = {\Gamma_1}^{\mathsf{s}}(C) \sqcap {\Gamma_2}^{\mathsf{s}}(C)$$
$$(\Gamma_1 \odot \Gamma_2)^{\mathsf{e}}(C) = {\Gamma_1}^{\mathsf{e}}(C) \sqcup {\Gamma_2}^{\mathsf{e}}(C)$$

The rationale for Γ^{s}: We can only guarantee a class is initialized at the join point for all evaluations reaching said point if both Γ_1 and Γ_2 say the class is initialized. The rationale for Γ^{e}: We assume the attacker knows exactly which control flow path was taken by the program, and thus exactly where classes fail to initialize. As demonstrated by p_{main}, the knowledge of where during control flow a class fails to initialize can be the source of an information leak.

5.2 Type Rules

The type rules, for expressions and statements respectively, are given in Figures 5 and 6. The rules define a type judgement relation $pc \vdash \Gamma\,\{t\}\,\Gamma' : \ell$, where $pc, \ell \in \mathcal{L}$. This type judgement reads:

> "Under context pc, t maps type environment Γ to type environment Γ' and error level ℓ"

We use the type judgement to gain insight into program behaviour in the following way.

> "Assume t is to be evaluated in a context containing information pc, with Γ expressing which classes are already initialized and information leaked through initialization error observations. Then Γ' records (at most) the classes which were initialized during (successful) evaluation of t, and the new information that can leak through initialization error observations. Further, ℓ expresses the information leaked to observations of an evaluation error of t. This whole evaluation contains no insecure flows".

$$\textsc{Num}_{\vdash}\ \frac{-}{pc \vdash \Gamma\,\{n\}\,\Gamma : \bot} \qquad \textsc{Var}_{\vdash}\ \frac{-}{pc \vdash \Gamma\,\{x\}\,\Gamma : \bot}$$

$$\textsc{Field}_{\vdash}\ \frac{pc \vdash \Gamma\,\{\tau(C)\}\,\Gamma' : \ell}{pc \vdash \Gamma\,\{C.x\}\,\Gamma' : \ell}$$

$$\textsc{Op-T}_{\vdash}\ \frac{pc \vdash \Gamma_0\,\{e_1\}\,\Gamma_1 : \ell_1 \qquad pc \sqcup \ell_1 \vdash \Gamma_1\,\{e_2\}\,\Gamma_2 : \ell_2}{pc \vdash \Gamma_0\,\{e_1 \oplus_{\mathsf{T}} e_2\}\,\Gamma_2 : \ell_1 \sqcup \ell_2}$$

$$\textsc{Op-P}_{\vdash}\ \frac{pc \vdash \Gamma_0\,\{e_1\}\,\Gamma_1 : \ell_1 \qquad pc \sqcup \ell_1 \vdash \Gamma_1\,\{e_2\}\,\Gamma_2 : \ell_2}{pc \vdash \Gamma_0\,\{e_1 \oplus_{\mathsf{P}} e_2\}\,\Gamma_2 : \ell_1 \sqcup \ell_2 \sqcup \mathsf{lvl}(e_1) \sqcup \mathsf{lvl}(e_2) \sqcup pc}$$

(a) Expressions

$$\textsc{Init-T}_{\vdash}\ \frac{\Gamma^{\mathsf{s}}(C) = \mathtt{I}}{pc \vdash \Gamma\,\{C\ \{i\}\}\,\Gamma : \bot}$$

$$\textsc{Init-F}_{\vdash}\ \frac{\Gamma^{\mathsf{s}}(C) = \mathtt{U} \qquad pc \sqcup \Gamma^{\mathsf{e}}(C) \vdash_C \Gamma[C \mapsto^{\mathsf{s}} \mathtt{B}]\,\{i\}\,\Gamma' : \ell_C}{pc \vdash \Gamma\,\{C\ \{i\}\}\,\Gamma'[C \mapsto^{\mathsf{s}} \mathtt{I}, C \mapsto^{\mathsf{e}} \ell_C] : \ell_C \sqcup {\Gamma'}^{\mathsf{e}}(C)}$$

(b) Class Definitions

$$\textsc{Init}_{\vdash}\ \frac{\bigsqcup_{j=1}^{i-1} \ell_j \sqcup pc \vdash \Gamma_{i-1}\,\{e_i\}\,\Gamma_i : \ell_i \quad \mathsf{lvl}(e_i), pc, \bigsqcup_{j=1}^{n} \ell_j \sqsubseteq \mathsf{lvl}(C.x_i)}{pc \vdash_C \Gamma_0\,\{x_1 = e_1; \cdots; x_n = e_n\}\,\Gamma_n : \bigsqcup_{j=1}^{n} \ell_j}$$

(c) Class Initializers

Fig. 5: Typing of Expressions

That last remark entails that we can use our enforcement to reason about the security of programs. The type rules contain several constraints engineered to make this so. We will explain the type rules now, and highlight a proof that our type judgment carries this meaning in Section 7.

We start with the rules for expression evaluation in Figure 5 (a). Evaluation of n and x cannot fail and does not initialize classes, thus motivating ($\textsc{Num}_{\vdash}$) and ($\textsc{Var}_{\vdash}$). However, this is not the case in ($\textsc{Field}_{\vdash}$), as evaluating $C.x$ can initialize C. We therefore type the definition of C, $\tau(C)$, using the rules in Figure 5 (b), which we will describe momentarily. Now that we know how to type base expressions we move on to the type rules for compound expressions, ($\textsc{Op-T}_{\vdash}$) and ($\textsc{Op-P}_{\vdash}$). As mentioned previously, there are two kinds of operators; total and partial. In both cases, when typing $e_1 \oplus e_2$, we first type e_1, followed by e_2. Notice that the context of the e_2 typing is augmented by ℓ_1; this is because in our semantics, e_2 is not evaluated at all if the evaluation of e_1 fails. Since either of these evaluations can fail, ℓ_j are both reflected in ℓ. For total operators, we are done. However, for partial operators, $e_1 \oplus e_2$ can fail because $e_1 \oplus e_2$ is undefined. An observer might know that $e_1 \oplus e_2$ fails as a consequence of this (for instance, when this is the only way $e_1 \oplus e_2$ can fail, as in $1/h$). Thus the information in

any variable or field in e_1 and e_2 leaks to ℓ.

We now move on to Figure 5 (b). Recall from the semantics that when evaluating $C.x$, either initialization of C is triggered or not. If C is triggered, no evaluation occurs, and if C is not triggered, the initializer is evaluated. This behaviour is reflected in (INIT-T$_\vdash$) and (INIT-F$_\vdash$). Observe that the initializer can be typed multiple times, once for each different context in which initialization, and thus failure, is possible. The need for this was highlighted at the end of Section 5.1. When typing $C\,\{i\}$, we first consult $\Gamma^{\mathsf{s}}(C)$. If $\Gamma^{\mathsf{s}}(C) = \text{I}$, no flows occur since no evaluation can possibly occur here. However, if $\Gamma^{\mathsf{s}}(C) = \text{U}$, initializer i is typed using the initializer typing rule in Figures 5 (c). The context of this typing is raised by $\Gamma^{\mathsf{e}}(C)$, as this evaluation is only possible if C has not been initialized (and then possibly failed) previously. Γ is updated to $\Gamma[C \mapsto^{\mathsf{s}} \text{B}]$ since C is being initialized. As this initialization attempt can fail as a consequence of a previous initialization attempt failing, $\Gamma'^{\mathsf{e}}(C)$ is reflected in ℓ. At last, the initialization status of C is set to I and the error level tracking of C is augmented by ℓ_C, in Γ'. Notice that there is no rule for the case $\Gamma^{\mathsf{s}}(C) = \text{B}$. This means that our type system will reject any program which contains a *mutual dependency* in field initializers in the class table, such as classes C and D in the following class table.

$$C\,\{l = 1 + D.l; h = 1 + E.h\}$$
$$D\,\{l = 0 + C.h\}$$
$$E\,\{h = 4\}$$

This also means that between $\Gamma[C \mapsto^{\mathsf{s}} \text{B}]$ and Γ', the initialization status of C is not updated. So $\Gamma'^{\mathsf{s}}(C) = \text{B}$, always. While rejecting programs with mutual dependencies rules out some well-behaving programs, it has been pointed out e.g. in [War06] section 3.5.1 that a) mutual dependencies should be considered a bug since they introduce hard-to-predict behavior, and b) there are tools which can detect these. Because of this, the added challenge in proving soundness, and since mutual dependencies do not increase the bandwidth of leaks through class initialization statuses, we chose not to consider this feature in our security analysis. To add this feature to the typesystem, it should be sufficient to replace the premise of (INIT-T$_\vdash$) with $\Gamma^{\mathsf{s}}(C) \in \{\text{B}, \text{I}\}$.

We now move on to Figure 5 (c). The one rule for initializer typing, (INIT$_\vdash$), types each initializer expression in the same order as they would be evaluated in the semantics. Since the evaluation of an initializer expression depends on the success of all prior initializer expressions in this order, the context of the e_i typing is raised by $\bigsqcup_{j=1}^{i-1} \ell_j$. Likewise, a future read of any field in C will succeed only if none of the initializer expressions evaluated to error during initialization, so $\bigsqcup_{j=1}^{n} \ell_j \sqsubseteq \mathsf{lv}(C.x_i)$ must hold.

In the statement typing rules in Figure 6, we prevent explicit flows ($l := h$) and *implicit* flows [DD77] via control structure (`if` $h = 0$ `then` $l := 0$ `else` $l := 1$) in a standard fashion [DD77], [VSI96], [SM03]. What is non-

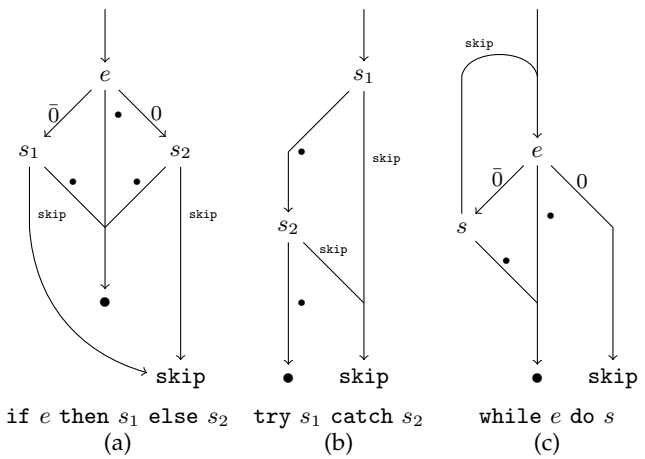


Fig. 7: Join Points in Control Flows.

standard here is how we construct type environments for joins in control flow branches. The control flows for the language constructs with join points (in successful evaluations) is illustrated in Figure 7. Compare (IF$_\vdash$) to Figure 7 (a). Here there are two successful control flow paths. The only classes we can guarantee are initialized at the join point are those that are initialized after taking either control flow path, hence the $(\Gamma_1 \odot \Gamma_2)^{\mathsf{s}}$ in the join point. Now compare (TRY$_\vdash$) to Figure 7 (b). There are two control flow paths here, one for successful evaluation of s_1, the other for successful evaluation of s_2 after an erroneous evaluation of s_1. The interesting case is the latter, as s_2 is typed under $\Gamma \odot \Gamma_1$. Since s_1 can yield an error without initializing a class, s_2 must be typed under Γ^{s}. Since s_1 can fail anywhere, and the attacker could know exactly where s_1 fails, s_2 must be typed under Γ_1^{e}. It turns out that Γ^{s} and Γ_1^{e} together are exactly $\Gamma \odot \Gamma_1$. Besides this, our treatment of exceptions is standard (see [Mye99], [PS03], [HS06], [AS09]). Now compare (WHILE$_\vdash$) to Figure 7 (c). Here there are (possibly) infinitely many control flow paths. In the type rule, n is the number of static iterations of the type system on e and s to produce successive type environments before fixpoint is reached. Observe that loops are typed the same way as the infinitely long sequence $e; s; e; s; \cdots$. Since there are finitely many Γ (as τ has finite classes, and policy has finite labels), by Lemma 7.2, eventually the typing of this sequence reaches a fixed point. Since e might yield 0 on first evaluation, the only classes we can guarantee are initialized are those that are initialized during evaluation of e, so the $^{\mathsf{s}}$-part of the join environment should equal $\Gamma_0'^{\mathsf{s}}$. Since an attacker could know exactly which iteration of the loop failed, and whether e or s failed, the $^{\mathsf{e}}$-part of the join environment must equal $\bigsqcup_{j=0}^{n} \Gamma_j'^{\mathsf{e}} \sqcup \Gamma_{j+1}^{\mathsf{e}}$. It turns out that this, together with $\Gamma_0'^{\mathsf{s}}$, is exactly $\bigodot_{j=0}^{n} \Gamma_j' \odot \Gamma_{j+1}$.

6 HIERARCHIES

So far, we have seen a complete semantics and set of type rules for our language, excluding the $C <: C'\,\{i\}$

$$\textsc{Skip}_{\vdash}\ \frac{-}{pc \vdash \Gamma\,\{\texttt{skip}\}\,\Gamma : \bot} \qquad \textsc{Field-A}_{\vdash}\ \frac{pc \vdash \Gamma\,\{e\}\,\Gamma' : \ell \quad pc \sqcup \ell \vdash \Gamma'\,\{C.x\}\,\Gamma'' : \ell' \quad \mathsf{lvl}(e), pc \sqcup \ell, \ell' \sqsubseteq \mathsf{lvl}(C.x)}{pc \vdash \Gamma\,\{C.x := e\}\,\Gamma'' : \ell \sqcup \ell'}$$

$$\textsc{Var-A}_{\vdash}\ \frac{pc \vdash \Gamma\,\{e\}\,\Gamma' : \ell \quad \mathsf{lvl}(e), pc, \ell \sqsubseteq \mathsf{lvl}(x)}{pc \vdash \Gamma\,\{x := e\}\,\Gamma' : \ell} \qquad \textsc{Seq}_{\vdash}\ \frac{pc \vdash \Gamma_0\,\{s_1\}\,\Gamma_1 : \ell_1 \quad pc \sqcup \ell_1 \vdash \Gamma_1\,\{s_2\}\,\Gamma_2 : \ell_2}{pc \vdash \Gamma_0\,\{s_1; s_2\}\,\Gamma_2 : \ell_1 \sqcup \ell_2}$$

$$\textsc{If}_{\vdash}\ \frac{pc \vdash \Gamma\,\{e\}\,\Gamma' : \ell \quad pc \sqcup \mathsf{lvl}(e) \sqcup \ell \vdash \Gamma'\,\{s_i\}\,\Gamma_i : \ell_i}{pc \vdash \Gamma\,\{\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2\}\,\Gamma_1 \odot \Gamma_2 : \ell \sqcup \ell_1 \sqcup \ell_2} \qquad \textsc{Try}_{\vdash}\ \frac{pc \vdash \Gamma\,\{s_1\}\,\Gamma_1 : \ell_1 \quad pc \sqcup \ell_1 \vdash \Gamma \odot \Gamma_1\,\{s_2\}\,\Gamma_2 : \ell_2}{pc \vdash \Gamma\,\{\texttt{try } s_1 \texttt{ catch } s_2\}\,\Gamma_1 \odot \Gamma_2 : \ell_2}$$

$$\textsc{While}_{\vdash}\ \frac{\begin{array}{c} pc \sqcup \ell_i \vdash \Gamma_i\,\{e\}\,\Gamma'_i : \ell^e_i \qquad pc \sqcup \ell_i \sqcup \ell^e_i \sqcup \mathsf{lvl}(e) \vdash \Gamma'_i\,\{s\}\,\Gamma_{i+1} : \ell^s_i \\ \ell_0 = \bot \quad \ell_{i+1} = \ell_i \sqcup \ell^e_i \sqcup \ell^s_i \quad i = 0..n \quad (\Gamma_n, \ell_n) = (\Gamma_{n+1}, \ell_{n+1}) \end{array}}{pc \vdash \Gamma_0\,\{\texttt{while } e \texttt{ do } s\}\,\bigodot_{j=0}^{n} \Gamma'_j \odot \Gamma_{j+1} : \ell_n}$$

Fig. 6: Typing of Statements

language construct. We saw in p_{dep} from Section 2 that hierarchical dependencies can be utilized to leak information. We now present the semantics of hierarchical dependencies, and how to analyze programs which utilize these. We handle hierarchical dependencies in the same way as dependencies in field initializers. While the procedure of initializing C' as a consequence of initializing C is the same as the procedure for initializing C' as a consequence of reading a field of C', we cannot emulate superclass initialization through field reads like we did in (FIELD-A-E$_{\Rightarrow}$) and (FIELD-A-OK$_{\Rightarrow}$), since a class does not necessarily have any class fields. While some rules can be merged to yield a more concise set of rules, having a rule for each possible combination of class and superclass initialization statuses simplifies case analysis of programs, and thus our proofs. As such, while these rules may be technical, they should be unsurprising, given what we have seen so far.

6.1 Semantics

The semantic rules of class hierarchies are given in Figure 8. Just like for normal class initialization, if initialization of C has begun already, we do nothing, and simply return the initialization status of C in σ. Hence (INIT-S-A$_{\Rightarrow}$). If C is uninitialized, however, then the first thing to do when initializing C is to initialize its superclass, C', if needed. If initialization of C' has failed, then we cannot proceed initializing C, so initialization of C will fail. Hence (INIT-S-UF$_{\Rightarrow}$). If initialization of C' has already begun (and has not failed), we skip the initialization of C', so this case in rule (INIT-S-UI$_{\Rightarrow}$) is just like the semantics of normal class initialization. If C' is uninitialized, we initialize it (setting the initialization status of C to being initialized, since we are currently initializing C). If this initialization of C' fails, then the initialization of C immediately fails (we don't even run i), hence (INIT-S-UUF$_{\Rightarrow}$). If initialization of C' succeeds, however, we run i. This evaluation yields either `skip` (for success) or $\bullet$ (for failure). Regardless of which, we return, with initialization of C set to the initialization status corresponding to the return value from the i evaluation, I (for success) or $\bullet$ (for failure). Hence (INIT-S-UUI$_{\Rightarrow}$).

6.2 Type Rules

The rules for typing class definitions with hierarchical dependencies are given in Figure 9. Like for normal class initialization, when the type environment asserts that C is initialized, since the semantics would do nothing in this case, no flows occur here, so when $pc \vdash \Gamma\,\{c\}\,\Gamma' : \ell$, $\Gamma' = \Gamma$ and $\ell = \bot$, hence (INIT-S-T$_{\vdash}$). If C is uninitialized in Γ, we check the status of C'. If Γ asserts that C' is already initialized, then no flows arise from or through C', so the (INIT-S-FT$_{\vdash}$) case is exactly like normal class initialization. The last rule, (INIT-S-FF$_{\vdash}$), considers the case where both C and C' are uninitialized in Γ. Since this might also be the case in the program semantics, we assume the worst, that neither class is initialized. First we type the C' class definition (under the assertion that C is being initialized). Since i would only be evaluated if C' initialized successfully, we raise the context under which we type i (evaluted after the C' definition) by $\ell_{C'}$. Likewise, because C can fail to initialize as a consequence of either the C' initialization failing or the i evaluation failing, $\Gamma''^{\mathsf{e}}(C)$ is augmented with $\ell_{C'} \sqcup \ell_C$.

7 Soundness

We now see how the different aspects of the type system fit together to form a whole which rejects leaking programs like p_{main} and p_{LinH} while at the same time staying permissive, accepting programs like p_{art} and p'_{LinH}, given in Section 4.3. As evident in program p^H_{main}, which is p_{main} with all class fields labeled $high$, the main flows of interest are not the typical explicit and implicit flows for simple imperative programs. For instance, the parts

$C.y := 0;$
`if` $h \neq 0$ `then (try new` D `catch skip) else skip;`

and

$C.y := 1;\ l := 0;$
`try new` D `catch` $l := 1$

are both secure. The insecurity arises when you put these in sequence. Typical type systems for information flow are (sequentially) compositional wrt. TINI, which makes proving soundness wrt. TINI a simple matter. As our type system does not have this property in general, our

$$\textsc{Init-s-A}_{\Rightarrow}\ \frac{\sigma(C) \neq \textsc{u}}{\langle \sigma, C <: C'\ \{i\}\rangle \Rightarrow \langle \sigma, \sigma(C)\rangle} \qquad \textsc{Init-s-UI}_{\Rightarrow}\ \frac{\sigma(C) = \textsc{u} \quad \sigma(C') \in \{\textsc{i}, \textsc{b}\} \quad \langle \sigma[C \mapsto \textsc{b}], \mathsf{s}(C,i)\rangle \Rightarrow \langle \sigma', T\rangle}{\langle \sigma, C <: C'\ \{i\}\rangle \Rightarrow \langle \sigma'[C \mapsto I(T)], I(T)\rangle}$$

$$\textsc{Init-s-UF}_{\Rightarrow}\ \frac{\sigma(C) = \textsc{u} \quad \sigma(C') = \bullet}{\langle \sigma, C <: C'\ \{i\}\rangle \Rightarrow \langle \sigma[C \mapsto \bullet], \bullet\rangle} \qquad \textsc{Init-s-UUF}_{\Rightarrow}\ \frac{\sigma(C) = \textsc{u} \quad \sigma(C') = \textsc{u} \quad \langle \sigma[C \mapsto \textsc{b}], \tau(C')\rangle \Rightarrow \langle \sigma', \bullet\rangle}{\langle \sigma, C <: C'\ \{i\}\rangle \Rightarrow \langle \sigma'[C \mapsto \bullet], \bullet\rangle}$$

$$\textsc{Init-s-UUI}_{\Rightarrow}\ \frac{\sigma(C) = \textsc{u} \quad \sigma(C') = \textsc{u} \quad \langle \sigma[C \mapsto \textsc{b}], \tau(C')\rangle \Rightarrow \langle \sigma', \textsc{i}\rangle \quad \langle \sigma'[C \mapsto \textsc{b}], \mathsf{s}(C,i)\rangle \Rightarrow \langle \sigma'', T\rangle}{\langle \sigma, C <: C'\ \{i\}\rangle \Rightarrow \langle \sigma''[C \mapsto I(T)], I(T)\rangle}$$

(a) Class Definitions

Fig. 8: Big-step Operational Semantics for Expression Evaluation, Class Hierarchies

$$\textsc{Init-s-T}_{\vdash}\ \frac{\Gamma^{\mathsf{s}}(C) = \textsc{i}}{pc \vdash \Gamma\ \{C <: C'\ \{i\}\}\ \Gamma : \bot} \qquad \textsc{Init-s-FT}_{\vdash}\ \frac{\Gamma^{\mathsf{s}}(C) = \textsc{u} \quad \Gamma^{\mathsf{s}}(C') = \textsc{i} \quad pc \sqcup \Gamma^{\mathsf{e}}(C) \vdash_C \Gamma[C \mapsto^{\mathsf{s}} \textsc{b}]\ \{i\}\ \Gamma' : \ell_C}{pc \vdash \Gamma\ \{C <: C'\ \{i\}\}\ \Gamma'[C \mapsto^{\mathsf{s}} \textsc{i}, C \mapsto^{\mathsf{e}} \ell_C] : \ell_C \sqcup \Gamma'^{\mathsf{e}}(C)}$$

$$\textsc{Init-s-FF}_{\vdash}\ \frac{\Gamma^{\mathsf{s}}(C) = \textsc{u} \quad \Gamma^{\mathsf{s}}(C') = \textsc{u} \quad pc \vdash \Gamma[C \mapsto^{\mathsf{s}} \textsc{b}]\ \{\tau(C')\}\ \Gamma' : \ell_{C'} \quad pc \sqcup \ell_{C'} \sqcup \Gamma'^{\mathsf{e}}(C) \vdash_C \Gamma'\ \{i\}\ \Gamma'' : \ell_C}{pc \vdash \Gamma\ \{C <: C'\ \{i\}\}\ \Gamma''[C \mapsto^{\mathsf{s}} \textsc{i}, C \mapsto^{\mathsf{e}} \ell_{C'} \sqcup \ell_C] : \ell_{C'} \sqcup \ell_C \sqcup \Gamma''^{\mathsf{e}}(C)}$$

(a) Class Definitions

Fig. 9: Typing of Expressions, Class Hierarchies

soundness proof is nonstandard. The key part of the soundness proof are the lemmas it makes use of, as these establish a) which semantic and typing invariants are required for compositionality, and that b) these invariants hold in initial states and typing environments. We motivate invariants with examples. Proofs of theoretical results presented in this paper can be found in the full version of this paper [RNS11]. We present the lemmas for statements; the same results hold for expressions.

7.1 Monotonicity

The semantics and the type system both satisfy simple monotonicity properties wrt. their respective environments. Let $\mathcal{L}_I$ be the (meet-semi-)lattice given by $\textsc{u} \sqsubseteq \textsc{b}$, $\textsc{b} \sqsubseteq \textsc{i}$ and $\textsc{b} \sqsubseteq \bullet$, illustrated in Figure 10. This yields a (meet-semi-)lattice $\mathcal{L}_{\Rightarrow}$ of memories, where $\sigma \sqsubseteq \sigma'$ iff

$$\forall C\,.\,\sigma(C) \sqsubseteq \sigma'(C).$$

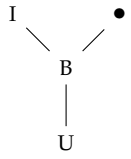


Fig. 10: $\mathcal{L}_I$.

Notice that no rule for establishing $\langle \sigma, \_\rangle \Rightarrow \langle \sigma', \_\rangle$, assigns C to an I in σ' such that $\sigma(C) \not\sqsubseteq I$. This yields the following monotonicity property.

Lemma 7.1 (Reduction relation monotone wrt. σ)**.** *For all σ, σ' and s, if $\langle \sigma, s\rangle \Rightarrow \langle \sigma', \_\rangle$, then $\sigma \sqsubseteq \sigma'$.*

We likewise obtain a lattice $\mathcal{L}_{\vdash}$ of type environments by defining $\Gamma \sqsubseteq \Gamma'$ iff

$$\forall C\,.\,\Gamma^{\mathsf{s}}(C) \sqsubseteq \Gamma'^{\mathsf{s}}(C) \wedge \Gamma^{\mathsf{e}}(C) \sqsubseteq \Gamma'^{\mathsf{e}}(C).$$

Observe that no rule for establishing $\_ \vdash \Gamma\ \{t\}\ \Gamma' : \_$ assigns C to an I and ℓ^{e} in Γ' s.t. $\Gamma^{\mathsf{s}}(C) \not\sqsubseteq I$ or $\Gamma^{\mathsf{e}}(C) \not\sqsubseteq \ell^{\mathsf{e}}$. This yields the following monotonicity property, which also helps motivate the definition of $(\Gamma_1 \odot \Gamma_2)^{\mathsf{e}}$.

Lemma 7.2 (type judgement monotone wrt. Γ)**.** *For all Γ, Γ', and s, if $\_ \vdash \Gamma\ \{s\}\ \Gamma' : \_$, then $\Gamma \sqsubseteq \Gamma'$.*

7.2 Must-analysis

Recall that part of the type system tracks, at any given point in the control flow of a program, which classes must be initialized. This analysis is crucial if we want to accept programs such as p'_{LinH}, since, without it, the enforcement would not know anything about the status of classes possibly initialized in the past, meaning the enforcement would need to regard C as uninitialized when its reference is seen under h. For this analysis to be sound, the following property must be preserved.

Definition 7.1 (agreement)**.** *σ agrees with Γ, written as $\Gamma \models_{\mathsf{dep}} \sigma$, iff, for all C,*
1) $\Gamma^{\mathsf{s}}(C) = \textsc{i} \implies \sigma(C) = \textsc{i}$,
2) $\sigma(C) = \textsc{b} \iff \Gamma^{\mathsf{s}}(C) = \textsc{b}$.

Pt. *1)* states what we expect; if the type environment states that a class is initialized, then that class is initialized in the corresponding memory. Pt. *2)* states that a class is being initialized in a run if and only if the analysis tracks that said class is being initialized[5]. While this property may be straight-forward, a straight-forward proof that the property is preserved through typing and reduction fails. Consider the case where $\tau(C)$ is to be evaluated under σ, with $\sigma(C) = \textsc{i}$, $\Gamma^{\mathsf{s}}(C) = \textsc{u}$, $\Gamma \models_{\mathsf{dep}} \sigma$ and $\_ \vdash \Gamma\ \{\tau(C)\}\ \Gamma' : \_$. Here, when $\langle \sigma, \tau(C)\rangle \Rightarrow \langle \sigma', \textsc{i}\rangle$, $\sigma' = \sigma$. However, $\Gamma' = \Gamma$ is not guaranteed. This scenario is not artificial or unrealistic, as demonstrated by the following program

$$\texttt{try}\ x := C.x\ \texttt{catch skip};$$
$$x := C.x$$

run in the presence of the following class table.

$$C\ \{x = 1 + D.x\}$$
$$D\ \{x = 5\}$$

5. About the $\iff$: Note that, since Γ overapproximates σ, there will be Γs for which $\Gamma^{\mathsf{s}}(C) = \textsc{b}$ for which there is no corresponding σ. But this is no problem, as we only need agreement to hold where such a σ exists.

In line 2 of this program, for the type environment Γ at this point, $\Gamma^{\mathsf{s}}(C) = \mathrm{U}$, since the must analysis can only guarantee that a class is initialized after a `try-catch` if said class is initialized in both the `try` and the `catch` branch (and the `catch` branch initializes nothing). However, in the memory σ at this point, $\sigma(C) = \mathrm{I}$, since the `try` branch never fails. In the final type environment Γ', $\Gamma'^{\mathsf{s}}(D) = \mathrm{I}$, and a quick review of the semantics will show that in the final memory σ', $\sigma'(D) = \mathrm{I}$, so $\Gamma' \models_{\mathsf{dep}} \sigma'$. *What* is it, in the type system, and in the semantics, that makes this so?

The answer: Dependencies. Recall that initialization of a class C can fail if any of the classes C depends on fail to initialize, or have failed previously. Thus, if C is initialized successfully, then all the classes C depends on must also be successfully initialized, for the remainder of the run (last statement follows from Lemma 7.1). The set $\mathsf{dep}(e)$ of classes e depends on is defined as follows.

$$\begin{aligned}
\mathsf{dep}(n) &= \emptyset \\
\mathsf{dep}(x) &= \emptyset \\
\mathsf{dep}(e_1 \oplus e_2) &= \mathsf{dep}(e_1) \cup \mathsf{dep}(e_2) \\
\mathsf{dep}(C.x) &= \mathsf{dep}(\tau(C)) \\
\mathsf{dep}(C\ \{i\}) &= \mathsf{dep}(i) \cup \{C\} \\
\mathsf{dep}(C <: C'\ \{i\}) &= \mathsf{dep}(C') \cup \mathsf{dep}(i) \cup \{C\} \\
\mathsf{dep}(\epsilon) &= \emptyset \\
\mathsf{dep}(x = e; i) &= \mathsf{dep}(e) \cup \mathsf{dep}(i)
\end{aligned}$$

Here, $\mathsf{dep}(e)$ is the set of classes that must be initialized as a consequence of (successfully) evaluating e. Note that in the case of $x = e$ of class C, this assignment is *always* performed as a consequence of C being initialized. Hence $\mathsf{dep}(x)$ is not part of a union on the right-hand side of the definition of $\mathsf{dep}(x = e; i)$. Since classes are only initialized when evaluating expressions, $\mathsf{dep}(\cdot)$ is not defined on s.

Definition 7.2 (dep consistency)**.** *For any function f where $\mathsf{dom}(\tau) \subseteq \mathsf{dom}(f)$, we say f is* dep*-consistent, written $\vdash_{\mathsf{dep}} f$, iff* $(\forall C \,.\, f(C) = \mathrm{I} \implies \forall C' \in \mathsf{dep}(\tau(C)) \,.\, f(C') = \mathrm{I})$.

The type system preserves dep-consistency.

Lemma 7.3 (dep-consistency preservation)**.** *For all Γ, Γ' and s, if i) $\_ \vdash \Gamma\,\{s\}\,\Gamma' : \_$, and ii) $\vdash_{\mathsf{dep}} \Gamma$, then iii) $\vdash_{\mathsf{dep}} \Gamma'$.*

Well-typed programs, run on a memory which agrees with the initial type environment, which terminate successfully, preserve dep-consistency and agreement.

Lemma 7.4 (agreement preservation)**.** *For all Γ, Γ', s, σ and σ', if i) $\_ \vdash \Gamma\,\{s\}\,\Gamma' : \_$, ii) $\vdash_{\mathsf{dep}} \Gamma$, $\vdash_{\mathsf{dep}} \sigma$, $\Gamma \models_{\mathsf{dep}} \sigma$, and iii) $\langle \sigma, s\rangle \Rightarrow \langle \sigma', T\rangle$ then iv) $\vdash_{\mathsf{dep}} \sigma'$, and v) $T \neq \bullet \implies \Gamma' \models_{\mathsf{dep}} \sigma'$.*

7.3 Errors

The main facilitator of the information channel being considered in this paper is evaluation errors, as these can cause a class initialization to fail and thus be used to store information. Cases where errors come into play in our proofs are therefore of key importance. However, as we have seen in p_{art}, this is not trivial; it is easy to construct memories which "lie", in the sense that they have registered that a class which can never fail, has failed to initialize. This creates artificial flows which our type system cannot guarantee the absence of. To address this issue, we formalize a sufficient condition for a memory to be "honest". Such memories are error consistent, that is, a class is only failed in the memory if, during runtime, that class could possibly fail. Since whether a class can fail or not depends on the same for the classes it depends on, this definition of error consistency is inductive in nature. Finally, our operational semantics preserves error consistency. This, together with the observation that initial memories are error consistent, means we only need to consider error consistent memories in our proofs.

Definition 7.3 (error consistency)**.** *σ is* error consistent*, written $\vdash_{\mathsf{err}} \sigma$, iff*

$$\begin{aligned}
&\forall C \,.\, \sigma(C) = \bullet \implies \\
&\exists \sigma';\, (\sigma' \sqsubseteq \sigma),\, (\vdash_{\mathsf{err}} \sigma'),\, (\sigma'(C) \neq \bullet) \,.\, \langle \sigma', \tau(C)\rangle \Rightarrow \langle _, \bullet\rangle.
\end{aligned}$$

Lemma 7.5 (error consistency preservation)**.** *For all σ, σ' and s, if i) $\vdash_{\mathsf{err}} \sigma$, and ii) $\langle \sigma, s\rangle \Rightarrow \langle \sigma', \_\rangle$, then iii) $\vdash_{\mathsf{err}} \sigma'$.*

Since our operational semantics preserves dep-consistency and agreement, we furthermore need only consider error consistent memories which are dependency consistent and agree with an appropriate type environment. Some term evaluations can still yield error under such memories; we call these terms volatile.

Definition 7.4 (volatile)**.** *s is volatile under Γ, written $\Gamma \models_{\mathsf{err}} s$, iff there exists a σ for which*

1) $\vdash_{\mathsf{err}} \sigma$, $\vdash_{\mathsf{dep}} \sigma$, $\Gamma \models_{\mathsf{dep}} \sigma$
2) $\langle \sigma, s\rangle \Rightarrow \langle \_, \bullet\rangle$

Finally we arrive at what we really needed: A well-typed volatile term leaks its context to its error level. Observe that in p_{art}, the context did not leak, since the type system did not encounter a partial operator application. Intuitively, volatile expressions *have* a partial operator application somewhere along the path the type system takes to analyze s. If you examine (OP-P$_{\vdash}$), you will see that this rule raises the error level by pc.

Lemma 7.6 (error leaks pc)**.** *For all Γ, s, pc and ℓ', if i) $pc \vdash \Gamma\,\{s\}\,\_ : \ell'$, ii) $\vdash_{\mathsf{dep}} \Gamma$, and iii) $\Gamma \models_{\mathsf{err}} s$, then iv) $pc \sqsubseteq \ell'$.*

Recall the leak in p_{main}. There, old confidential contextual information, out of scope at the time the "low" assignment occurs, manages to leak into said assignment. When comparing two ℓ-equivalent memories during parallel runs, we need a way to know, at the time of the low assignment, that the program, in a previous

scope, branched on confidential information, thus causing different class initialization statuses (of classes with no observable fields). ℓ-consistency gives us this.

Definition 7.5 (ℓ-consistent)**.** *σ_1, σ_2 are ℓ-consistent under Γ, written $\sigma_1 \sim_\ell^\Gamma \sigma_2$, iff, for all C,*

$$\sigma_1(C) \neq \sigma_2(C) \wedge \Gamma \models_{\mathsf{err}} \tau(C) \implies \Gamma^{\mathsf{e}}(C) \not\sqsubseteq \ell.$$

It turns out this relation is an equivalence relation, which will be useful in the next subsection. To link this relation with Lemma 7.6, intuitively, at the point where two runs start disagreeing on the initialization status of a class, the runs start entering a context containing confidential information. This context carries over to the class initialization. If that initialization can fail, then by Lemma 7.6, the error level of the initialization contains confidential information. At last, this error level is recorded in a type environment, and this recording carries over to the join point of the two runs.

7.4 Noninterference

We now have enough tools to tackle the type soundness result. If all runs on ℓ-equivalent memories followed the same control-flow path in the program, proving soundness would be an easy matter. In reality, however, two ℓ-equivalent runs can take different control-flow paths. It turns out that this only happens when an evaluation of ℓ-unobservables is involved. For instance, for `if` e `then` s_1 `else` s_2, if the evaluation of e depended only on observables, then e would evaluate to the same value in the two runs on ℓ-equivalent memories, thus resulting in the same control-flow path taken. We would like the memories at the join point to be ℓ-equivalent, but we cannot compare the intermediate memories of the two branching runs, as that reasoning will not be local. To address this issue, we ensure in our enforcement that no observable effects are allowed when the context contains confidential information. This way, if the memories at the branching point are ℓ-equivalent (and ℓ-consistent), the memories at the join point will be ℓ-equivalent (and ℓ-consistent) by transitivity.

Lemma 7.7. *For all s, σ, σ', Γ, Γ', ℓ and pc,*
if i) $pc \vdash \Gamma\,\{s\}\,\Gamma' : \_$, ii) $(\vdash_{\mathsf{dep}} \Gamma)$, $(\vdash_{\mathsf{dep}} \sigma)$, $(\Gamma \models_{\mathsf{dep}} \sigma)$, iii) $(\vdash_{\mathsf{err}} \sigma)$, iv) $\langle \sigma, s\rangle \Rightarrow \langle \sigma', \_\rangle$, and iv) $pc \not\sqsubseteq \ell$,
then vi) $\sigma \sim_\ell^{\Gamma'} \sigma'$, and vii) $\sigma =_\ell \sigma'$.

At last we get to the main lemma which uses the results we have seen so far. Given the usual assumptions, but for two ℓ-equivalent, ℓ-consistent memories, Lemma 7.8 states that the final memories are ℓ-equivalent and ℓ-consistent, and the success status of the evaluations differs only if confidential information is in ℓ'.

Lemma 7.8 (main)**.** *For all s, σ_j, σ_j', Γ, Γ', ℓ, ℓ' and T_j,*
if i) $\_ \vdash \Gamma\,\{s\}\,\Gamma' : \ell'$, ii) $(\vdash_{\mathsf{dep}} \Gamma)$, $(\vdash_{\mathsf{dep}} \sigma_j)$, $(\Gamma \models_{\mathsf{dep}} \sigma_j)$, iii) $(\vdash_{\mathsf{err}} \sigma_j)$, iv) $\langle \sigma_j, s\rangle \Rightarrow \langle \sigma_j', T_j\rangle$,
v) $\sigma_1 \sim_\ell^\Gamma \sigma_2$, and vi) $\sigma_1 =_\ell \sigma_2$,
then vii) $T_j \neq \bullet = T_{\bar{j}} \implies \ell' \not\sqsubseteq \ell$,
viii) $\sigma_1' \sim_\ell^{\Gamma'} \sigma_2'$, and ix) $\sigma_1' =_\ell \sigma_2'$.

Take special note of parts *i)*, *v)* and *vi)* in the premise, and parts *viii)* and *ix)* in the conclusion, of Lemma 7.8 — these imply type soundness! We show how to "bootstrap" Lemma 7.8 to obtain a proof of soundness for our type system now. Let ${\Gamma_{\text{init}}}^{\mathsf{s}}(C) = \mathrm{U}$ and ${\Gamma_{\text{init}}}^{\mathsf{e}}(C) = \bot$ for each class in τ.

Theorem 7.1 (type soundness)**.** *For all s,*
if there exists a Γ' and ℓ' for which $\bot \vdash \Gamma_{\text{init}}\,\{s\}\,\Gamma' : \ell'$, then s satisfies TINI.

Let initial σ_j be given such that $\langle \sigma_j, s\rangle \Rightarrow \langle \sigma_j', T_j\rangle$ and $\sigma_1 =_\ell \sigma_2$, for any ℓ. TINI then requires that $\sigma_1' =_\ell \sigma_2'$. We get this by instantiating Lemma 7.8. We have $iv)$ and $vi)$ of Lemma 7.8. Let $\Gamma = \Gamma_{\text{init}}$ and $pc = \bot$. This immediately gives us part *i)* of Lemma 7.8. Since no classes are initialized in Γ or σ_j, and none have failed in σ_j, parts *ii)*, *iii)* and *v)* of Lemma 7.8 hold. This immediately gives us part *vii)* of Lemma 7.8 (regardless of whether $T_j = \texttt{skip}$ or $T_j = \bullet$), which is exactly what we needed. So Theorem 7.1 holds.

8 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 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 Java bytecode-like language. Bernardeschi et al. [BFLM05] check information-flow security in Java bytecode by combining 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 to inheritance.

Banerjee and Naumann [BN05] show how to guarantee noninterference by type-based analysis for a Java-like object-oriented language. Amtoft et al. [ABB06] present a flow-sensitive logic for reasoning about information flow in 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 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 [BR05].

Hedin and Sands [HS06] prove a noninterference property for a type system that tracks information flow via class-cast and null-pointer exceptions. 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.

9 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 track its side effects. The enforcement ensures that the side effects do not reveal anything about the differences in control-flow paths that the program might take depending on secret input. 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 in the presence of class hierarchies. (Soundness of Jif's class initialization is yet to be established.)

Future work includes machine-checking the invariant-based proofs, currently given by structural induction in the associated technical report [RNS11]. We would also like to perform case studies to evaluate the precision of our enforcement mechanism.

Acknowledgements: This work was funded by the European Community under the WebSand and ProSecuToR projects and the Swedish research agencies SSF and VR. 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.

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.

[Bil] Bill Pugh et al. The "Double-Checked Locking is Broken" Declaration. Located at http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

[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.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, August 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, 2nd 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.

[NS10] K. Nakata and A. Sabelfeld. Securing Class Initialization. In *Proceedings of the IFIP International Conference on Trust Management (IFIPTM)*, LNCS. Springer-Verlag, June 2010.

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

[RNS11] W. Rafnsson, K. Nakata, and A. Sabelfeld. Secure class initialization. Technical report, Chalmers University of Technology, 2011. Located at http://www.cse.chalmers.se/~rafnsson/2011TDSC.

[Sch97] Schmidt, Douglas C. and Harrison, Tim. Double-checked locking. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern languages of program design 3*, pages 363–375. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[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. 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.

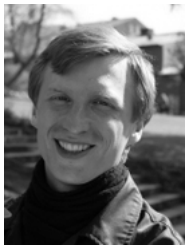
[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[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.

[War06] M. Warnier. *Language Based Security for Java and JML*. PhD thesis, Radboud University, Nijmegen, Netherlands, 2006.

Willard Rafnsson is a Ph.D student in the Department of Computer Science and Engineering at Chalmers University of Technology in Gothenburg, Sweden. His work centers around language-based information-flow security, that is, the development and use of program analysis, programming languages and computer security techniques to enforce security policies on information flows in programs. He received his master's degree in computer science from Aalborg University, Denmark, in 2008.

Keiko Nakata is a senior researcher at the Institute of Cybernetics, Tallinn University of Technology, Estonia. Her work is concerned around programming language semantics and implementation, program verification, and constructive mathematics. She received a PhD degree in computer science in 2007 from the Research Institute for Mathematical Sciences, Kyoto University, Japan.



Andrei Sabelfeld is a Professor in the Department of Computer Science and Engineering at Chalmers University of Technology in Gothenburg, Sweden. After receiving his Ph.D. in Computer Science from Chalmers in 2001 and before joining Chalmers as faculty in 2004, he was a Research Associate at Cornell University in Ithaca, NY. His research has developed the link between two areas of Computer Science: Programming Languages and Computer Security.