Catch Me If You Can: Permissive Yet Secure Error Handling

Aslan Askarov
Department of Computer Science
Cornell University
aslan@cs.cornell.edu

Andrei Sabelfeld
Department of Computer Science and Engineering
Chalmers University of Technology
andrei@chalmers.se

Abstract

Program errors are a source of information leaks. Tracking these leaks is hard because error propagation breaks out of program structure. Programming languages often feature *exception* constructs to provide some structure to error handling: for example, the `try`...`catch` blocks in Java and Caml. Mainstream information-flow security compilers such as Jif and FlowCaml enforce rigid rules for exceptions in order to prevent leaks via public side effects of computation whose reachability depends on exceptions.

This paper presents a general and permissive alternative to the rigid solution: the programmer is offered a choice for each type of error/exception whether to handle it or not. The security mechanism ensures that, in the former case, it is never handled and, in the latter case, it is always handled with the mainstream restrictions. This mechanism extends naturally to a language with procedures and output, where we show the soundness of the mechanism with respect to termination-insensitive noninterference.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.4.6 [*Security and Protection*]: Information flow controls

General Terms Languages, Security

Keywords Security type system, exception handling

1. Introduction

Program errors are a source of information leaks. Tracking these leaks is hard because error propagation breaks out of program structure. Programming languages often feature *exception* constructs to provide some structure to error handling: for example, the `try`...`catch` blocks in Java and Caml. Mainstream information-flow compilers such as Jif [24] and FlowCaml [30] for Java and Caml, respectively, enforce rigid rules for exceptions: for example, if an exception is thrown in a sensitive context, then no public side effects are allowed either in the code that follows the exception in the `try` block or in the `catch` handler.

Assume h and h' are variables storing secret (high) information and variable low public (low) information. Programs like

```
h' := 1/h; low := 0;
```

are not allowed by Jif and FlowCaml, since the reachability of the public assignment depends on secret h. Further programs like

```
try { h' := 1/h; low := 0; } catch { }
```

are rejected by Jif and FlowCaml because the reachability of the public assignment still depends on secret h. On the other hand, programs like

```
try { h' := 1/h;} catch { }; low := 0;
```

are allowed since now the public assignment is reachable independently of secrets.

The exception mechanism in the mainstream information-flow compilers ensures that all exceptions are handled, and the constraints on the public side effects in the `try` and `catch` parts are enforced. Although the intentions of the exception mechanism are good, there is an unfortunate price that the programmer has to pay: cluttering the code with exception handling. Although Jif provides some help for reducing the need to handle exceptions (for example, a null-pointer analysis), there is clear evidence that exception handling has become a usability bottleneck for Jif programming:

- "The obligation to handle runtime exceptions in Jif can easily make code more clunky than necessary." [2]
- "Handling all these potential control flows [due to exceptions] becomes arduous and clutters the code." [20]
- "Handling every runtime exception is an exceedingly time-consuming part of converting Java code to Jif code." [20]

Experiments by King et al. [22] suggest that exception-related annotations are not always critical for security. For

PLAS '09 June 15, Dublin, Ireland.

example, they find that 706 out of 757 unhandled exception warnings of JLift [21], an interprocedural extension of Jif, are in fact false alarms (around 93%!).

The goal of this paper is to help reduce the burden of catching errors by programmers but without giving up security guarantees. One observation is that the program fragment from the first listing is in fact secure (in the sense of *termination-insensitive security*), as long as we consistently *not* catch the exception in the outer context. Termination-insensitive security is the target security policy of Jif and FlowCaml, since they ignore leaks due to the termination behavior of the program. The first program

```
h' := 1/h; low := 0;
```

is secure according to this policy because its only leak is through termination behavior: depending on the secret, the program terminates normally (or not). There are no other dependencies from the secret by the public observer. The second program

```
try { h' := 1/h; low := 0; } catch { }
```

on the other hand, is much more dangerous, since it is possible to exploit the termination channel within the program itself and magnify it. This program leaks the value of an n-bit nonnegative integer bit-by-bit:

```
i := 0;
while (i<n) {
  low_i := 0;
  try   { h':= 1/(h & 2^i); low_i := 1; }
  catch { }
  i := i + 1;
}
```

An exception can be thrown if the i-th bit of the variable h is zero and in that case the second assignment in the `try` block is not executed as the control jumps to the `catch` block. As a result, this program leaks all of the secret h into low. Finally, the third program

```
try { h' := 1/h;} catch { }; low := 0;
```

is innocent because the `try` and `catch` parts do not have any public side effects.

Based on these observations, we propose a security mechanism for permissive yet secure error-handling. We can achieve the best of the both worlds (i.e., security and usability), by the following "all-or-nothing" per-exception policy: for each error/exception, either *never* catch exceptions or *always* do it. The former may result in a leak through abnormal termination and the latter corresponds to no leak. From the enforcement point of view, as we have mentioned, leaks through abnormal termination are typically allowed by mainstream compilers (the simplest example of a program with such a leak is a loop with a secret guard and no public side effects). From the security-policy point of view, these leaks are ignored by *termination-insensitive noninterference* [1], a target property for compilers as Jif and FlowCaml.

$$
\begin{aligned}
e &::= n \mid x \mid e \ op \ e \\
c &::= \texttt{skip} \mid x := e \mid c; c \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \\
&\quad \mid \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{try}\ c\ \texttt{catch}\ c
\end{aligned}
$$

Figure 1. Syntax

Note that "never handle an exception" corresponds to the policies implemented in FlowCaml and Jif for serious errors, such as resources exhaustion (out-of-memory, stack-overflow, etc.). For this kind of errors, FlowCaml and Jif do not allow exceptions to be caught. However, we believe that giving the programmer control over which errors/exceptions are handled is a more promising alternative. It is appealing because it may relieve programmers from cluttering the code with exception handling without jeopardizing security.

We show that our solution extends naturally to a language with procedures and output, which we illustrate by proving the soundness with respect to termination-insensitive noninterference.

For simplicity, the rest of the paper uses the term *exceptions* for all kinds of errors that can be handled in code. This corresponds to the `Throwable` class in the Java jargon.

Section 2 illustrates the essence of our approach by a security type system for a trimmed down language with exceptions. Section 3 defines a more ambitions language with procedures, multiple exception types, and output. Section 4 presents a security type system for the extended language and Section 5 argues that this system guarantees security. Section 6 presents is a discussion of, among other things, examples. Section 7 discusses related work. Section 8 draws some conclusions.

2. Permissive error handling in a nutshell

To demonstrate the key idea of the permissiveness we gain by our approach, this section presents a security type system for a simple imperative language (displayed in Figure 1) equipped with simple error recovery. We assume that operations on integers (such as division) may throw a runtime error.

Assume a simple security lattice with levels L (*low*, or public) and H (*high*, or secret), where $L \sqsubset H$. Metavariables ℓ, pc, and eh range over security levels. Γ is a mapping of program variables to their security levels. We extend Γ to expressions by assuming an expression is mapped to the least upper bound of the security levels of the variables that occur in it. In some examples, we assume that l and h are typical low and high variables, respectively.

Expressions The typing rules for expressions have the form $\Gamma, pc \vdash e : \ell$, where ℓ is the level of an excep-

$$\text{(T-SKIP)}\ \Gamma, pc, eh \vdash \texttt{skip} : L$$

$$\text{(T-ASGN)}\ \frac{\Gamma, pc \vdash e :\ \ell \qquad \Gamma(e) \sqsubseteq \Gamma(x) \qquad pc \sqsubseteq \Gamma(x)}{\Gamma, pc, eh \vdash x := e\ : \ell}$$

$$\text{(T-SEQ-1)}\ \frac{\Gamma, pc, X \vdash c_1 : \ell_1 \qquad \Gamma, pc \sqcup \ell_1, X \vdash c_2 : \ell_2}{\Gamma, pc, X \vdash c_1; c_2 : \ell_1 \sqcup \ell_2}$$

$$\text{(T-SEQ-2)}\ \frac{\Gamma, pc, \emptyset \vdash c_1 : \ell_1 \qquad \Gamma, pc, \emptyset \vdash c_2 : \ell_2}{\Gamma, pc, \emptyset \vdash c_1; c_2 : \ell_1 \sqcup \ell_2}$$

$$\text{(T-IF)}\ \frac{\Gamma, pc \vdash e : \ell \qquad \Gamma, pc \sqcup \Gamma(e), eh \vdash c_i : \ell_i, i = 1, 2}{\Gamma, pc, eh \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \ell \sqcup \ell_1 \sqcup \ell_2}$$

$$\text{(T-WHILE)}\ \frac{\Gamma, pc \vdash e : \ell \qquad \Gamma, pc \sqcup \Gamma(e), eh \vdash c : \ell''}{\Gamma, pc, eh \vdash \texttt{while}\ e\ \texttt{do}\ c : \ell \sqcup \ell''}$$

$$\text{(T-TRY)}\ \frac{\Gamma, pc, X \vdash c_1 : \ell_1 \qquad \Gamma, pc \sqcup \ell_1, eh \vdash c_2 : \ell_2}{\Gamma, pc, eh \vdash \texttt{try}\ c_1\ \texttt{catch}\ c_2 : \ell_2}$$

Figure 2. Typing rules for commands

tion that expression e may throw in the context pc. That is, $\Gamma, pc \vdash e\ : L$ is possible only if e does not throw an exception or if both exception and pc are low.

Commands The typing rules for commands have the form $\Gamma, pc, eh \vdash c\ : \ell$, where eh is the *exception handler* which effectively indicates whether there are any exception handlers in the context where c is executed. We let eh range over two values X and $\emptyset$, where X means that c is executed within a `try`...`catch` block and $\emptyset$ means that it is not. In the latter case we know that exceptions raised within c are not going to be caught. As with expressions, ℓ is the level of exception that c may throw.

The rule for `skip` records L as the exception level since it does not throw any exceptions. The rule for assignment prevents explicit flows (such as $l := h$), *implicit* flows [16] via control structure (as in `if` h `then` $l := 1$ `else` $l := 0$) in a standard fashion [16, 32, 28]. In addition, the security level of an exception the expression may throw is propagated to the command level.

We have two typing rules for sequential composition. Rule (T-Seq-1) covers the case when the exception handler is set. In this case, following the standard practice [23, 27, 29, 18], we enforce that c_2 does not have side effects lower than the level of possible exceptions that c_1 may throw. Rule (T-Seq-2) considers the case when exceptions in c_1 are not handled. In this case, we allow low side effects in c_2: this rule does not restrict the pc for c_2. Both rules record the level of exceptions that $c_1; c_2$ may throw as a join of levels ℓ_1 and ℓ_2 of exceptions that c_1 and c_2, respectively, may throw.

$$\begin{aligned}
\mathit{Program} &::= \mathit{Proc}^*\ \{\ c\ \}\\
\mathit{Proc} &::= \texttt{proc}\ p : \ell\ (\texttt{in} : \ell\ x, \texttt{out} : \ell\ y)\\
&\qquad \texttt{throws}\ X^*_\ell\ \texttt{failswith}\ X^*_\ell\ \{\ c\ \}\\
e &::= n \mid x \mid e\ \mathit{op}\ e\\
c &::= \texttt{skip} \mid x := e \mid c; c \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c\\
&\quad \mid \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{try}\ c\ \texttt{catch}\ X\ c\\
&\quad \mid p(e, x) \mid \texttt{output}(e, ch)
\end{aligned}$$

Figure 3. Syntax of Jif$_0$

Rules for `if` and `while` are similar in that they both propagate the level of exceptions that the guard e may throw. Observe that we have $\Gamma \vdash e :\ \ell$ implies $\ell \sqsubseteq \Gamma(e)$ for well-typed expressions e in the simple language. Taking this into account, the rules for `if` (resp., `while`) only requires that the branches (resp., loop body) have no side effects below $\Gamma(e)$.

The rule for `try`...`catch` sets the exception handler for c_1 to X. The body of the error handling code c_2 is typed with the original exception handler eh. Moreover, to reflect that execution of c_2 depends on the erroneous behavior at level ℓ_1, we prevent side effects below ℓ_1 in c_2. This complements the similar restriction in rule (T-Seq-1).

Revisiting the examples from Section 1, this program is rejected by the type system:

```
try { h' := 1/h; low := 0; } catch { }
```

The problem is that the second low assignment reflects whether the previous high assignment has succeeded or not. Recall that this is a dangerous leak, which, as shown in Section 1, can be magnified. The magnification is also rightfully rejected by the type system. Consider, on the other hand the program:

```
h' := 1/h; low := 0;
```

This program is accepted by the type system of Figure 2. Because exceptions are not caught, a magnification of this attack is not possible. Finally, the innocent program

```
try { h' := 1/h;} catch { }; low := 0;
```

is also accepted by the type system since neither the `try` nor the `catch` blocks have public side effects. Since the version with no exception handling is also accepted, it might be preferable from the usability perspective: it is clearly less cluttered.

3. Imperative language with procedures and outputs: Jif$_0$

This section presents an imperative language with procedures and output, dubbed Jif$_0$. Figure 3 shows the syntax of the language, and we outline the semantics below. Com-

pared to the simple language of Section 2, Jif$_0$ is enriched with multiple exception types, procedures, and outputs.

3.1 Semantics

Exceptions For simplicity, we assume that exceptions can only be raised from within expressions and are propagated further to commands as discussed below. Assuming a set of possible exceptions is fixed, let R, X, Y, Z range over possible exceptions and let $\mathbf{R}, \mathbf{X}, \mathbf{Y}, \mathbf{Z}$ denote sets of exceptions. Exception handling is done using using `try` c_1 `catch` X c_2 command which runs c_1 and passes the control to c_2 if c_1 raises an exception X.

Semantics of expressions Expressions semantics rules have the form $\langle e, M\rangle \downarrow u$. We parameterize over the details of expressions semantics and assume that u can be either an integer n or an exceptional value $err\ X$, where X is a type of the exception which evaluation of e has thrown.

Semantics of commands Semantic configurations for commands have the form $\langle c, E\rangle$, where $E = (M, C, F)$ is the environment containing memory M, channel environment C, and procedure declarations F. As shorthands, we may write $E(x)$ for $E.M(x)$, $E(p)$ for $E.F(p)$, and $E(ch)$ for $E.C(ch)$.

Figures 4 and 5 show the small-step operational semantics for our language. The semantic transitions in these figures are labeled with *low events* which are discussed in detail in Section 5. Most of the semantic rules are standard. A configuration with `skip` evaluates to a non-syntactic command *stop*. The same happens with assignment and output, unless evaluating the assigned (output) expression in these commands raises an exception. In that case this exception is propagated into commands.

Exception propagation Exceptions in commands are represented by a non-syntactic command $throw\ X$, where X is the type of an exception. Exception propagation rule in Figure 5 formalizes how exceptions are propagated from expressions to commands by defining a context $Q[e]$. Here Q ranges over all contexts that involve expression evaluation.

Exception propagation among commands takes place in the rules for sequential composition and the rules for `try` c_1 `catch` X c_2. This is shown in Figure 4. If evaluation of c_1 yields an exception then so does $c_1; c_2$. An exception is consumed if the type of exception c_1 evaluates to $throw\ X$, and X is declared in the `catch` statement. Otherwise, if c_1 evaluates to $throw\ X'$ for some $X' \neq X$, then the entire `try` c_1 `catch` X c_2 evaluates to $throw\ X'$.

Procedures A notable feature of Jif$_0$ is that procedure declarations specify two sets of exceptions that a procedure can throw. One set corresponds to exceptions that can be thrown by the method in conventional sense (as in Java/Jif) and assumed to be handled by a method caller. The other one corresponding to exceptions which must not be handled by the caller. For simplicity, we fix procedures to two parameters x

$$\langle \texttt{skip}, E\rangle \rightarrow \langle stop, E\rangle$$

$$\frac{\langle e, E.M\rangle \downarrow n \qquad \Gamma(x) = L \implies \nu = (x, n) \qquad \Gamma(x) = H \implies \nu = \epsilon}{\langle x := e, E\rangle \rightarrow_\nu \langle stop, E[M.x \mapsto n]\rangle}$$

$$\frac{\langle c_1, E\rangle \rightarrow_\nu \langle c_1', E'\rangle \qquad c_1' \notin \{stop, throw\ X\}}{\langle c_1; c_2, E\rangle \rightarrow_\nu \langle c_1'; c_2, E'\rangle}$$

$$\frac{\langle c_1, E\rangle \rightarrow_\nu \langle stop, E'\rangle}{\langle c_1; c_2, E\rangle \rightarrow_\nu \langle c_2, E'\rangle}$$

$$\frac{\langle c_1, E\rangle \rightarrow_{err} \langle throw\ X, E\rangle}{\langle c_1; c_2, E\rangle \rightarrow_{err} \langle throw\ X, E\rangle}$$

$$\frac{\langle e, E.M\rangle \downarrow n \qquad n \neq 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, E\rangle \rightarrow \langle c_1, E\rangle}$$

$$\frac{\langle e, E.M\rangle \downarrow n \qquad n = 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, E\rangle \rightarrow \langle c_2, E\rangle}$$

$$\frac{\langle e, E.M\rangle \downarrow n \qquad n \neq 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, E\rangle \rightarrow \langle c; \texttt{while}\ e\ \texttt{do}\ c, E\rangle}$$

$$\frac{\langle e, E.M\rangle \downarrow n \qquad n = 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, E\rangle \rightarrow \langle stop, E\rangle}$$

$$\frac{\langle e, E.M\rangle \downarrow n \qquad E(p) = \texttt{proc}(a, b)\ \{\ c\ \}}{\langle p(e, x), E\rangle \rightarrow \langle c[n/a, x/b], E\rangle}$$

$$\frac{\langle e, E.M\rangle \downarrow n \qquad \Gamma(ch) = L \implies \nu = O(ch, n) \qquad \Gamma(ch) = H \implies \nu = \epsilon}{\langle \texttt{output}(e, ch), E\rangle \rightarrow_\nu \langle stop, E.C[ch \mapsto E(ch) : n]\rangle}$$

$$\frac{\langle c_1, E\rangle \rightarrow_\nu \langle c_1', E'\rangle \qquad c_1' \notin \{stop, throw\ X'\}}{\langle \texttt{try}\ c_1\ \texttt{catch}\ X\ c_2, E\rangle \rightarrow_\nu \langle \texttt{try}\ c_1'\ \texttt{catch}\ X\ c_2, E'\rangle}$$

$$\frac{\langle c_1, E\rangle \rightarrow_\nu \langle stop, E'\rangle}{\langle \texttt{try}\ c_1\ \texttt{catch}\ X\ c_2, E\rangle \rightarrow_\nu \langle stop, E'\rangle}$$

$$\frac{\langle c_1, E\rangle \rightarrow_{err} \langle throw\ X, E'\rangle}{\langle \texttt{try}\ c_1\ \texttt{catch}\ X\ c_2, E\rangle \rightarrow \langle c_2, E'\rangle}$$

$$\frac{\langle c_1, E\rangle \rightarrow_{err} \langle throw\ X', E'\rangle \qquad X \neq X'}{\langle \texttt{try}\ c_1\ \texttt{catch}\ X\ c_2, E\rangle \rightarrow_{err} \langle throw\ X', E'\rangle}$$

Figure 4. Semantics of the extended language

$$\frac{\langle e, E.M\rangle \downarrow \mathit{err}\ X}{\langle Q[e], E\rangle \rightarrow_{\mathit{err}}\ \langle \mathit{throw}\ X, E\rangle}$$

$$Q[e] ::= x := e; \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{output}(e, ch) \mid p(e, x)$$

Figure 5. Exception propagation

and y, where x is passed by value, and y is passed by reference, so that the names of local variable in procedures do not clash with global variables. The syntax for procedures is

$$\texttt{proc}\ p : \ell_0\ (\texttt{in} : \ell_1\ x, \texttt{out} : \ell_2\ y)\ \texttt{throws}\ X^*_\ell\ \texttt{failswith}\ R^*_\ell\ \{\ c\ \}$$

where p is the name of a procedure, ℓ_0 is the *begin-label* of the procedure [23, 24], ℓ_1 and ℓ_2 are labels of the procedure arguments, sets X^*_ℓ and R^*_ℓ are two sets of *labeled exceptions* that p may throw, and, finally, c is the procedure body. The begin-label specifies the lower bound on the side-effects in the body c. The two argument labels ℓ_1, ℓ_2 specify upper bounds on the levels of actual arguments that are passed to the procedure. The exceptions following the keywords `throws` and `failswith` are annotated with labels as well.

Neither of the labels have effect at run-time and are used for type checking as discussed in Section 4.

Figure 4 also contains a rule for procedure invocation $p(e, x)$. At the invocation time, if the expression e evaluates to a value, then the configuration transitions to the procedure body c, where call-by-value argument is substituted with the evaluated value n, and call-by-reference argument is substituted with the second argument—variable x. This rule relies on the non-clashing name assumption mentioned above. Although simplified, such semantics for procedure invocation are sufficient for our purposes of illustrating the treatment of exceptions.

Outputs The rule for outputs generates low events (observable by the attacker, as we discuss in Section 5), but only if the level of the output channel is low. No event is generated for output on a high channel.

In our semantics we have a channel environment $E.C$, with each channel represented by a stream. Outputting a value appends the value to the corresponding channel stream.

4. Type system

This section presents a security type system for Jif$_0$. The type system extends the idea described in Section 2 to deal with multiple exception types, procedures, and outputs. We assume the following form of a typing environment $\Gamma = \Sigma, \Omega, \Delta$, where Σ is the procedure environment, Ω is the variable environment, and Δ is the channel environment.

4.1 Exception sets and labeled exceptions

We use notation $\mathbf{X}$ to refer to a set of exceptions $\{X, Y, Z, \dots\}$. To let the type system track security level associated with an exception let *labeled exception* X_ℓ specify that exception X has security level ℓ. For the sets of labeled exceptions (not necessary of the same level) we use notation $\mathbf{X}^\star$. An additional requirement on labeled exception sets is that a set $\mathbf{X}^\star$ may not contain any two labeled exceptions X_ℓ, $X_{\ell'}$ with $\ell \neq \ell'$.

Given a labeled exception X_ℓ, we refer to the corresponding unlabeled exception X by simply dropping the label ℓ from its name. Similarly, we refer to the unlabeled exception set that corresponds to $\mathbf{X}^\star$ via $\mathbf{X}$.

Exception tainting We overload the operator $\sqcup$ to apply it to the set of labeled exceptions as follows:

$$\mathbf{X}^\star \sqcup \ell' = \{X_{\ell \sqcup \ell'} \mid X_\ell \in \mathbf{X}^\star\}$$

Union, intersection, and subtraction We also define the following auxiliary operations on the exception sets that are used in the typing rules.

1. *Union of two labeled sets*

$$\mathbf{X}^\star \oplus \mathbf{Y}^\star = \{X_\ell | X_\ell \in \mathbf{X}^\star\ \wedge\ X_\ell \notin \mathbf{Y}\} \cup \{Y_\ell | Y_\ell \notin \mathbf{X}^\star\ \wedge\ Y_\ell \in \mathbf{Y}^\star\} \cup \{Z_{\ell \sqcup \ell'} | Z_\ell \in \mathbf{X}^\star \wedge Z_{\ell'} \in \mathbf{Y}^\star\}$$

   In this definition if sets $\mathbf{X}$ and $\mathbf{Y}$ contain no common exceptions then $\oplus$ matches an ordinary set union of labeled exception sets.

2. *Intersection of an exception set with a labeled exception set* $\mathbf{X}^\star \sqcap \mathbf{Y} = \{X_\ell \mid X_\ell \in \mathbf{X}^\star \wedge X \in \mathbf{Y}\}$

3. *Subtraction of an exception set from a labeled exception set* $\mathbf{X}^\star \setminus \mathbf{Y} = \{X_\ell \mid X_\ell \in \mathbf{X}^\star \wedge X \notin \mathbf{Y}\}$

4.2 Expression types

Typing rules for expressions have the form $\Gamma \vdash e : \mathbf{R}^\star$, where $\mathbf{R}^\star$ is the set of labeled exceptions that expression e may raise.

Since we parameterize over expression semantics, we formulate properties that every instance of expression type system must satisfy. These properties are listed later in Section 5.

4.3 Command types

Typing rules for commands have the form $\Gamma, pc, \mathbf{X} \vdash c : \mathbf{R}^\star$, where $\mathbf{X}$ is the set of exceptions that are handled by the environment which executes c, and $\mathbf{R}^\star$ is the set of labeled exceptions that c may raise.

Note that the set $\mathbf{R}^\star$ is labeled and reflects security levels of possible exceptions, while $\mathbf{X}$ is unlabeled. This is because our semantics do not have exception levels at run-time. This is consistent with Jif's implementation, where levels are erased from Jif's exception types, and at runtime there is no difference between catching what was `Exception : H` or `Exception : L` at the source level, because both types are erased into Java's `Exception` type. Hence, the type system must be aware of this. While we follow Jif's implementation on this, we do not foresee much difference if the semantics are designed to keep track of exception levels at run-time instead.

Figure 6 presents the typing rules. The type system is a mix of standard techniques typical for security type systems and the new elements specific for our treatment of exceptions. We briefly go through the interesting rules, focusing on new features.

In the rule for `skip`, the set of exceptions that can be raised is $\emptyset$. The rule for assignment propagates the set of exceptions $\mathbf{R}^\star$ that expression e may raise. Moreover, the resulting set is tainted with the pc label to ensure that the levels of possible exceptions record implicit flows. The same happens in the rule for output.

The rule for sequential composition $c_1; c_2$ is one place where the set of exceptions $\mathbf{X}$ that are handled by the context is used. Here it affects the pc label in which the second command c_2 is typed. Given labeled exception set $\mathbf{R}_\mathbf{1}^\star$ that c_1 may raise, if any of these exceptions may be handled by the outer context then c_2 must have no side-effects lower than the level of those exceptions. This is enforced by typing c_2 with program counter $pc \sqcup lev(\mathbf{R}_\mathbf{1}^\star \sqcap \mathbf{X})$. The function $lev(\mathbf{Y}^\star)$ returns the least upper bound of levels of exceptions among $\mathbf{Y}^\star$ and returns L in case $\mathbf{Y}^\star$ is empty. Finally, the rule for sequential composition returns a union of $\mathbf{R}_\mathbf{1}^\star$ and $\mathbf{R}_\mathbf{2}^\star$ as a set of exceptions that $c_1; c_2$ may raise.

The rules for conditionals and loops record that evaluating the guard e may raise exceptions and the level of those exceptions is tainted with the pc label, when constructing the final set of exceptions.

The rule for exception handling `try` c_1 `catch` Y c_2 is another interesting spot. To type c_1, we add exception Y into the context of exceptions that can be handled. The rule requires that Y must be possible to raise from within c_1 by having a clause which demands that Y in a labeled form must belong to $\mathbf{R}^\star{}_1$. The level with which Y is raised is important to restrict side-effects in c_2, and that is what the last premise of the rule does. Finally, to record all unhandled exceptions that `try` c_1 `catch` Y c_2 may raise itself, we return all exceptions from $\mathbf{R}_\mathbf{1}^\star$ but Y together with $\mathbf{R}_\mathbf{2}^\star$.

Procedure types For procedures we introduce procedure type $proc\ \ell_0\ \ell_1\ \ell_2\ \mathbf{A}^\star\ \mathbf{B}^\star$. Here ℓ_0 is the *begin-label* of the method [24] and ℓ_1, ℓ_2 are labels of the method arguments. The labeled exception sets $\mathbf{A}^\star$ and $\mathbf{B}^\star$ respectively record which exceptions can be thrown and which must not be handled by the method caller. We assume that the two sets are disjoint, that is $\mathbf{A} \cap \mathbf{B} = \emptyset$.

The rule for typing a procedure declaration uses the extended environment Σ, Ω, Δ. The rule constructs local environment Γ_p to type the body of the procedure, where the procedure arguments are annotated with respective levels. Here Ω_p is the variable environment that contains global variables together with the local procedure variables.

The body of the procedure c is type-checked starting with the program counter set to the begin-label ℓ_0. Recall that for exceptions listed after `throws` keyword we assume that they can be handled by the caller. Therefore, the set $\mathbf{X}$ is added to the environment for type-checking c. Similarly, exceptions following `failswith` keyword must not be handled by the context, and, therefore, the rule requires that the intersection of unlabeled sets $\mathbf{X}$ and $\mathbf{R}$ is empty. Finally, the rule ensures that the set of labeled exceptions that c may raise $\mathbf{Z}^\star$ matches what is specified in the procedure declaration by requiring $\mathbf{Z}^\star = \mathbf{X}^\star \oplus \mathbf{R}^\star$.

Calling a procedure $p(e, x)$ is another place where we inspect the set $\mathbf{X}$ of exceptions that the calling context may handle. Here the rule enforces that the set $\mathbf{X}$ must not overlap with exceptions that are listed after `failswith` in the declaration of p. Moreover, we require invariant typing on the in-out argument x.

4.4 Program typing

The last rule in Figure 6 is the one for typing an entire program. For typing an entire program we only need variable and channel environments. This rule builds the procedure environment from the procedure declarations and type checks each of the procedures and the program body in the constructed environment.

5. Security guarantees

This section introduces the attacker model for our system, requirements on expression semantics and type system, and establishes guarantees that the type system of Section 4 enforces.

5.1 Attacker model: low events

Our attacker is modeled via *low events*—pieces of information that the attacker can obtain during program execution. A low event α can have one of the following forms:

events	$\nu ::= \epsilon \mid \alpha$
low events	$\alpha ::= (x, n) \mid O(ch, n) \mid err$

Event (x, n) is generated upon an assignment of value n to a low variable x ($\Gamma(x) = L$). Similarly, event $O(ch, n)$ is an output on a low channel ch of the value n. The low event err is produced when the final configuration of the semantic transition is $\langle throw\ X, E\rangle$ for some exception X and environment E.

$$\Gamma, pc, \mathbf{X} \vdash \texttt{skip} : \emptyset \qquad \frac{\Gamma \vdash e : \mathbf{R}^\star \qquad \Gamma(e) \sqsubseteq \Gamma(x) \qquad pc \sqsubseteq \Gamma(x)}{\Gamma, pc, \mathbf{X} \vdash x := e : \mathbf{R}^\star \sqcup pc} \qquad \frac{\Gamma \vdash e : \mathbf{R}^\star \qquad \Gamma(e) \sqsubseteq \Gamma(ch) \qquad pc \sqsubseteq \Gamma(ch)}{\Gamma, pc, \mathbf{X} \vdash \texttt{output}(e, ch) : \mathbf{R}^\star \sqcup pc}$$

$$\frac{\Gamma, pc, \mathbf{X} \vdash c_1 : \mathbf{R_1^\star} \qquad \Gamma, pc \sqcup lev(\mathbf{R_1^\star} \sqcap \mathbf{X}), \mathbf{X} \vdash c_2 : \mathbf{R_2^\star}}{\Gamma, pc, \mathbf{X} \vdash c_1; c_2 : \mathbf{R_1^\star} \oplus \mathbf{R_2^\star}} \qquad \frac{\Gamma \vdash e : \mathbf{R}^\star \qquad \Gamma, pc \sqcup \Gamma(e), \mathbf{X} \vdash c_i : \mathbf{R_i^\star} \quad i = 1, 2}{\Gamma, pc, \mathbf{X} \vdash \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 : (\mathbf{R}^\star \sqcup pc) \oplus \mathbf{R_1^\star} \oplus \mathbf{R_2^\star}}$$

$$\frac{\Gamma \vdash e : \mathbf{R}^\star \qquad \Gamma, pc \sqcup \Gamma(e), \mathbf{X} \vdash c : \mathbf{R_2^\star}}{\Gamma, pc, \mathbf{X} \vdash \texttt{while } e \texttt{ do } c : (\mathbf{R}^\star \sqcup pc) \oplus \mathbf{R_2^\star}} \qquad \frac{\Gamma, pc, \mathbf{X} \cup \{Y\} \vdash c_1 : \mathbf{R_1^\star} \qquad \exists \ell \,.\, Y_\ell \in \mathbf{R_1} \qquad \Gamma, pc \sqcup \ell, \mathbf{X} \vdash c_2 : \mathbf{R_2^\star}}{\Gamma, pc, \mathbf{X} \vdash \texttt{try } c_1 \texttt{ catch } Y\ c_2 : (\mathbf{R_1^\star} \setminus \{Y\}) \oplus \mathbf{R_2^\star}}$$

$$\frac{\Gamma_p \triangleq \Sigma, \Omega_p[x \mapsto \ell_1, y \mapsto \ell_2], \Delta \qquad \Gamma_p, \ell_0, \mathbf{X} \vdash c : \mathbf{Z}^\star \qquad \mathbf{X} \cap \mathbf{R} = \emptyset \qquad \mathbf{Z}^\star = \mathbf{X}^\star \oplus \mathbf{R}^\star}{\Sigma, \Omega, \Delta \vdash \texttt{proc } p : \ell_0\ (\texttt{in} : \ell_1\ x, \texttt{out} : \ell_2\ y)\ \texttt{throws } \mathbf{X}^\star\ \texttt{failswith } \mathbf{R}^\star\ \{\ c\ \} : proc\ \ell_0\ \ell_1\ \ell_2\ \mathbf{X}^\star\ \mathbf{R}^\star}$$

$$\frac{\Gamma.\Sigma(p) = proc\ \ell_0\ \ell_1\ \ell_2\ \mathbf{A}^\star\ \mathbf{B}^\star \qquad \Gamma \vdash e : \ell', \mathbf{R}^\star \qquad pc \sqsubseteq \ell_0 \qquad \ell' \sqsubseteq \ell_1 \qquad \Gamma(x) \sqsubseteq \ell_2 \qquad \ell_2 \sqsubseteq \Gamma(x) \qquad \mathbf{X} \cap \mathbf{B} = \emptyset}{\Gamma, pc, \mathbf{X} \vdash p(e, x) : \mathbf{A}^\star \oplus \mathbf{B}^\star \oplus (\mathbf{R}^\star \sqcup pc)}$$

$$\frac{\begin{array}{c} Proc_j = \texttt{proc } p_j : \ell_j\ (\texttt{in} : \ell'_j\ x, \texttt{out} : \ell''_j\ y)\ \texttt{throws } \mathbf{A_j^\star}\ \texttt{failswith } \mathbf{B_j^\star}\{\ c_j\ \} \quad j = 1 \ldots k \\ \Sigma \triangleq \cup_j [p_j \mapsto proc\ \ell_j\ \ell'_j\ \ell''_j\ \mathbf{A_j^\star}\ \mathbf{B_j^\star}] \qquad \Sigma, \Omega, \Delta \vdash Proc_j : proc\ \ell_j\ \ell'_j\ \ell''_j\ \mathbf{A_j^\star}\ \mathbf{B_j^\star} \qquad \Gamma \triangleq \Sigma, \Omega, \Delta \qquad \Gamma, L, \emptyset \vdash c : \mathbf{R}^\star \end{array}}{\Omega, \Delta \vdash Proc_1 \ldots Proc_k\ \{\ c\ \}}$$

Figure 6. Type system for Jif$_0$: commands, procedures, and programs

Note that the type of the exception X does not propagate to the low event because the specifics of the exception are not observable to the attacker.

Refining attacker's view Observing low assignments leads to a strong attacker model. We adopt this conservative model for simplicity, which is justified because it subsumes more liberal attackers. For example, a more realistic attacker is the one that observes only outputs and errors.

We assume that the attacker observes a single run of a program. Observing multiple runs of the program with different inputs correspond to a more powerful attacker. We assume an external security mechanism may be placed to control the bandwidth of this channel (for example, this mechanism may introduce delays between programs runs).

5.2 Semantics with low events

Semantic transitions in Figure 4 are already annotated with low events. Low events are produced in the rules for assignment and output. The rule for exception propagation on Figure 5 generates an error event corresponding to an exception. Rules for sequential composition propagate low events. So do the rules for exception handling in case the thrown exception is not dedicated for the handler. Finally, an error event may be consumed by the exception handling if the type of the exception matches the one of the handler.

Notation Given a sequence of transitions from $\langle c, E\rangle$ to $\langle c', E'\rangle$ that produce a single low event α we write $\langle c, E\rangle \longrightarrow^*_\alpha \langle c', E'\rangle$. For transitions from $\langle c, E\rangle$ to $\langle c', E'\rangle$ that produce a sequence of low events $(\alpha_0 \ldots \alpha_n)$ we write $\langle c, E\rangle \longrightarrow^*_{\alpha_0 \ldots \alpha_n} \langle c', E'\rangle$. A sequence of low events $\alpha_0 \ldots \alpha_n$ can also be abbreviated as $\vec{\alpha}$ and we write the above transitions as $\langle c, E\rangle \longrightarrow^*_{\vec{\alpha}} \langle c', E'\rangle$. If no low events are produced in transitions from $\langle c, E\rangle$ to $\langle c', E'\rangle$, this can be denoted by $\langle c, E\rangle \longrightarrow^* \langle c', E'\rangle$ or $\langle c, E\rangle \longrightarrow^*_\epsilon \langle c', E'\rangle$. We also write $\langle c, E\rangle \longrightarrow^*_\nu \langle c', E'\rangle$ to refer to zero or more transitions that generate at most one low event.

5.3 Security condition

The target security property for our system we have *termination-insensitive noninterference (TINI)*. Before stating the definition of TINI, we introduce *low-equivalence* of memories and channel environments.

Definition 1 (Low-equivalence)**.**

- *Two memory environments M_1 and M_2 are low-equivalent, written $M_1 =_L M_2$, when they agree on all low variables.*
- *Two channel environments C_1 and C_2 are low-equivalent, written $C_1 =_L C_2$, when they agree on all low streams.*

We also write $E_1 =_L E_2$ if E_1 and E_2 agree on the same procedure declaration environments, and their respective memories and channel environments are low-equivalent.

Our definition of termination-insensitive noninterference definition is a variant of the one from [1] adapted for the semantics with error low events:

Definition 2 (TINI)**.** *A program $P = Proc_1 \ldots Proc_k\ \{\ c\ \}$ satisfies* termination-insensitive noninterference (TINI) *if*

- *whenever $M_1 =_L M_2$, $C_1 =_L C_2$, and*

- F *is a procedure declaration environment derived from* P*, and*
- $E_j \triangleq (M_j, C_j, F_j), j = 1, 2$*, and*
- $\langle c, E_1\rangle \longrightarrow^*_{\vec{\alpha}} \langle c'_1, E'_1\rangle$

then there are c'_2, E'_2 *such that one of the following holds:*

1. $\langle c, E_2\rangle \longrightarrow^*_{\vec{\alpha}} \langle c'_2, E'_2\rangle$*, or*
2. $\langle c, E_2\rangle \longrightarrow^*_{\vec{\beta} err} \langle c'_2, E'_2\rangle$ *and* $\vec{\beta}$ *is a prefix of* $\vec{\alpha}$*, or*
3. $\langle c, E_2\rangle \longrightarrow^*_{\vec{\beta}} \langle c'_2, E'_2\rangle$*,* $\vec{\beta}$ *is a prefix of* $\vec{\alpha}$*, and no further low events can be produced from* $\langle c'_2, E'_2\rangle$*.*

Let us recall a few examples from [1] that illustrate how termination-insensitivity is captured by the security definition. Program

```
low := h;
```

is rejected by the above definition since given any two low-equivalent environments the resulting low events may not be the same.

On the other hand the program

```
h := low; low' := h
```

is accepted by the definition because the low assignment event is going to be the same $(\texttt{low}', n)$, where $E_1.M(\texttt{low}) = E_2.M(\texttt{low}) = n$ for all pairs of low-equivalent environments E_1, E_2.

Consider now the program

```
while h do skip; output (0, low_ch)
```

which leaks through the termination behavior: whether the low output is reachable depends on the secret guard. The program has no other leaks though, and so it is accepted by the termination-insensitive definition. Indeed, take a pair of low-equivalent environments E_1, E_2 with $E_1.M(h) = 0$ and $E_2.M(h) \neq 0$. In this case run of this program from E_1 produces a low event $\alpha = O(\texttt{low\_ch}, 0)$, while the run from E_2 enters enter the non-terminating loop and produces no events. This corresponds to clause (3) in Definition 2, where $\vec{\beta} = \epsilon$ which is a trivial prefix of α. If in this example the `while` loop is replaced with a command that may throw an error depending on h such as $\texttt{h}' := 1/\texttt{h}$ the modified program is still accepted by Definition 2, now due to clause (2).

The next example is

```
while (h > 0) do skip; output (h, low_ch);
```

The program outputs a non-positive secret on a low channel. This program is rejected. It is enough to consider two low-equivalent environments where h is non-negative, but not necessary the same.

The following program is accepted by Definition 2.

```
while i < n {
  output (0, low_ch);

  if (i == h) then while 1 do skip;
              else skip;
  i := i + 1;
}
```

Observe that this program leaks the value of h because the attacker can deduce h by counting the number of low outputs. Although Definition 2 accepts this kind of brute-force attacks (and so do information-flow mainstream tools such as Jif [24], FlowCaml [30], and the SPARK Examiner [5, 9]), the impact of these brute-force attacks is limited: the attacker cannot learn the secret in polynomial time in the size of the secret; and, for uniformly distributed secrets, the advantage the attacker gains when guessing the secret after observing a polynomial amount of output is negligible in the size of the secret [1].

As with an earlier example, if we replace the nonterminating loop with an exception-raising statement, such as $h := 1/0$, the modified program is still accepted.

On the other hand, the magnification attack is rightfully rejected by the definition because of different low events in runs that start with memories different in h.

5.4 Assurance

This section first formulates our main assurance result, spells out requirements on expression type system and a key lemma that is used in the soundness proof The main theorem is

Theorem 1 (Soundness of the type system)**.** *Given a program* P *such that* $\Omega, \Delta \vdash P$ *then* P *satisfies TINI.*

Proof of Theorem 1 relies on a number of properties that expression type system and semantics must satisfy and on command lemmas, including statement of Lemma 1 that we present below. The proof details are available in the accompanying technical report [3].

Properties of expressions The properties of the expression type system and semantics that we demand are:

$$\text{If } \Gamma \vdash e : \mathbf{R}^\star \text{ then } \forall X_\ell \in \mathbf{R}^\star \,.\, \ell \sqsubseteq \Gamma(e) \quad (1)$$

$$\text{If } \Gamma \vdash e : \mathbf{R}^\star \text{ and } \langle e, E\rangle \downarrow \langle err\ X, E\rangle \text{ then } X_\ell \in \mathbf{R}^\star \text{ for some } \ell \quad (2)$$

$$\text{If } E_1 =_L E_2 \text{ and } \Gamma \vdash e : \mathbf{R}^\star \text{ and } \langle e, E_1\rangle \downarrow err\ X \text{ and } \langle e, E_2\rangle \downarrow u, u \neq err \text{ then } X_H \in \mathbf{R}^\star \quad (3)$$

Property (1) solely depends on the type system and specifies that a level of a possible exception may not be higher than the level of the expression.

Property (2) specifies that if expression evaluation results in an exception, this exception must have been properly recorded by the type system.

Property (3) requires that given two low-equivalent environments such that evaluation of an expression results in an exception in only one of them, then this exception must depend on high variable, that is, its level must be H.

Lemmas for commands The key property of the command type system is reflected in the following lemma that enumerates all possible cases of how low events behave in well-typed programs.

Lemma 1. *If* $\Gamma, pc, \mathbf{X} \vdash c : \mathbf{R}^\star$ *and we have*

1. $E_1 =_L E_2$, *and*
2. $\langle c, E_1\rangle \longrightarrow^* \langle c_1', E_1'\rangle \rightarrow_\alpha \langle c_1'', E_1''\rangle$, *and*
3. $\langle c, E_2\rangle \longrightarrow^* \langle c_2', E_2'\rangle \rightarrow_\beta \langle c_2'', E_2''\rangle$, *then*

- *If* $\alpha \neq err$ *then*
  - $\beta \neq err \implies \alpha = \beta$, $c_1' = c_2'$, $c_1'' = c_2''$, $E_1' =_L E_2'$, *and* $E_1'' =_L E_2''$.
  - $\beta = err \implies c_2'' = throw\ Z$, *where* $Z \notin \mathbf{X}$
- *If* $\alpha = err$ *with* $c_1'' = throw\ Z$ *then*
  - $Z \in \mathbf{X} \implies \beta = err$ *and* $c_2'' = throw\ Z'$ *for some* Z'.

The intuition behind this Lemma is following. The first case $\alpha \neq err$, $\beta \neq err$ implies that both configurations must agree on commands and low parts of the environments at the time when low events are produced. Hence, the low events are the same, the resulting commands are the same, and the resulting environments are low-equivalent.

In case $\alpha \neq err$, $\beta = err$, then the second run must terminate with an unhandled exception Z'. This is a consequence of the typing rule for sequential composition, which otherwise could allow magnification leaks.

The last case complements the previous two by specifying what happens if $\alpha = err$.

The proof of Lemma 1 is by induction on the structure of c. Special cases include sequential composition and exception handling. For sequential composition $c_1; c_2$ one uses an inner induction on c_1, considering separately the cases when c_1 runs without generating α and when it does. For exception handling there are several sub-cases, the most interesting of which is when the control may pass to the exception handler depending on a secret. In this case we appeal to the property of the `try...catch` typing rule that prevents low side-effects in the exception handler.

5.5 Implication for batch-job execution

The language of Section 3 supports outputs whose effect is immediate to the attacker. In a language without intermediate outputs, when the result of program is available to the attacker only at the end of its execution, the type system of this section enforces traditional *batch-style termination-insensitive noninterference*. Formally, we have the following corollary of Theorem 1.

Corollary 1. *If* $P = Proc_1 \ldots Proc_k \{ c \}$ *is a program without outputs then for any* E_1, E_2, *such that* $E_1 =_L E_2$, *if* $\langle c, E_1\rangle \longrightarrow^*_{\vec{\alpha}} \langle stop, E_1'\rangle$ *and* $\langle c, E_2\rangle \longrightarrow^*_{\vec{\beta}} \langle stop, E_2'\rangle$ *then* $E_1' =_L E_2'$.

6. Examples and discussion

A new language feature gives new possibilities to programmers. We discuss some pros and cons of the proposed exception treatment.

6.1 Monotonicity of exceptions

One property of the type system above is that a code that has been designed for catching exceptions can be used in a context where exceptions are ignored:

```
proc P:H (in:H px, out:H py)
  throws DivByZero:H { py = 1/px; }
proc Q:H (in:H qx, out:H qy) {
   try { P(qx, qy); }
   catch (DivByZero) {qy = 0; }
}
proc R:L (in:H rx, out:H ry)
  failswith DivByZero:H
    { P(rx, ry); output(0, low_ch); }
```

In this example a procedure `P` is declared to throw an exception `DivByZero`. This procedure can be called in two different kinds of contexts—one is exemplified by procedure `Q` where the exception declared in `P` is caught. The other one is demonstrated by procedure `R` where this exception type is declared via `failswith` keyword. We refer to this feature of the type system as *monotonicity* of exception sets.

6.2 Impact on code style and refactoring

With the possibility of ignoring exceptions there is a danger that programmers write code where most of the exceptions are ignored either deliberately or because constraints on the used methods forbid callers from handling them.

In this light, we believe our approach provides most benefit at the early stage of program development, when the core functionality is implemented. As the codebase evolves and programmers shift their attention from mere functionality to adherence with interfaces and usability of their code, the treatment of exceptions can change as well.

Existing approaches that deal with exceptions at early stages of development in security-typed languages are:

1. Explicit declassification of the pc label after a statement that raises the pc label.
2. Surrounding the statement that raises the pc label with `try...catch` construct where exceptions are simply ignored.

While helping the programmer to "put the compiler out of the way", each of these options has its drawbacks. Declassification of the pc label makes it less transparent for the security guarantees that the program provides. Based on our [2] experience and the one by Hicks et al. [20], `try...catch` blocks reduce transparency of the code and can negatively affect programmer's productivity at the early development stages, when the basic code functionality is being tested and debugged, and runtime exceptions actually do happen.

```
static void main(principal user, String[] args) {
  jif.runtime.Runtime[user] rt = null;
  try {
    rt = jif.runtime.Runtime[user].getRuntime();
  } catch (SecurityException e) {}
  InputStreamReader[{user:}] inS = null;
  try { inS = new InputStreamReader[{user:}](
          rt.stdin(new label{user:}));
  } catch (SecurityException ex) {
  } catch (NullPointerException e) {}
  ... }
```

Figure 7a. Example Jif code from [19]

Compared to the above two approaches we believe that the possibility of safely ignoring exceptions and `failswith` declaration allows programmer to both

- keep the program code relatively transparent, and
- avoid spurious declassifications, keeping the confidence in the obtained security guarantees.

Consider the scenario when at first iteration a programmer implements a method `R` with the following declarations

```
proc R:L (in H:rx, out H:ry)
 failswith X, Y, Z { ... }
```

The method `R` can be used in the context where none of `X`, `Y`, or `Z` are handled such as

```
proc Q:L (in H:qx, out H:qy)
 failswith X, Y, Z { ...  P(.., ..); ... }
```

If later the body of `R` is rewritten to reduce the types of exceptions that it may throw (e.g., by catching them internally), then `R` can now be also used in a new context permitting the context to catch more exceptions (this may be necessity due to other methods).

6.3 Examples in Jif syntax

Figure 7a is an example code fragment from [19] where the two possible exceptions of types `SecurityException` and `NullPointerException` are caught by the code, but no special handling is implemented. Indeed, there is little one can do if any of these exceptions take place. We moreover note that such approach to exception handling is very common to other Jif case studies [2, 12] as well. Figure 7b is a pseudocode of the same method rewritten in a syntax that supports `failswith` keyword. Due to `failswith` declaration extensive wrapping with `try`...`catch` is not necessary here yet we may have low side-effects later in the body of this method.

6.4 Further extensions

This paper presents the core idea for a permissive yet secure exception treatment. We briefly outline some directions for extensions.

```
static void main(principal user, String[] args)
   failswith SecurityException,
             NullPointerException{
   jif.runtime.Runtime[user] rt =
     jif.runtime.Runtime[user].getRuntime();
   InputStreamReader[{user:}] inS =
      new InputStreamReader[{user:}](
         rt.stdin(new label{user:}));
   ... }
```

Figure 7b. Alternative declaration using `failswith`

Explicit exception propagation As discussed earlier ignoring exceptions completely may be not the best programming practice. We may suggest a new syntactic construction `try` c_1 `catchandpropagate` c_2.

```
try { ... } catchandpropagate (SomeException e) {
 ... throw new AnotherException() }
```

The idea with `try` c_1 `catchandpropagate` c_2 is that it allows handling exceptions from c_1 in some way but c_2 is required to throw a different exception of an adequate level to the caller. The type system would enforce that c_2 indeed throws an exception by employing some variant of *must* static analysis. The end security guarantee will not be affected by this—as long as the type system knows that some exception is thrown to the caller, magnification attacks of the kind discussed in Section 1 are not possible.

This extension can be handy in combination with the *ArgCheck* programming pattern discussed in [2]. In particular, the pattern proposes checking method parameters for null-values and throwing `InvalidArgumentException` instead of `NullPointerException` that results in more meaningful error reporting for the caller.

This can also be useful if the programmer needs a finer distinction of particular throw-sites as in the following example where method `m()` has `failswith` annotation.

```
void m() failswith NullPointerException {...}
```

This method can be called in a context that also throws `NullPointerException` (abbreviated as `NPE`) which the programmer wants to handle.

```
try {
  foo.bar(); // foo could be null
  try { m(); }  catchandpropagate(NPE e) {
                throw new Error(); }
} catch (NPE e) { // NPEs thrown by m() will not
// be caught here, but the NPE thrown if foo is
// null will be caught here
}
```

Syntactic support for exception hierarchies Java (and Jif) exceptions form a class hierarchy, which allows catching multiple exception types in a single `catch`. Similarly the `throws` in Java/Jif relies on the exception class hierarchy and subtyping. To make `failswith` approach more usable

in a language with exception hierarchies `catch` syntax can be extended to exclude certain exception types. We exemplify the idea with simple instance of how this would look

```
try { ... }
catch (Exception but NPE e) {...}
```

For Jif, such extension can be implemented with moderate effort using Polyglot [25] framework. One compilation strategy in this case can be to catch exceptions declared after the `but` keyword first, re-throwing them there, and handling the rest of the exceptions (declared before the `but` keyword) in a separate successful `catch` block after that.

Alternative translation An alternative to propagating exceptions is a translation of failswith annotation that instruments the method body with a `try`...`catch` block catching exceptions named in failswith list and throwing a runtime error in place. The same functionality can be recovered with `catchandpropagate` extension discussed in this section. The price in both cases is light runtime overhead due to wrapping `try`...`catch`/`catchandpropagate` blocks. Note that failswith annotations in isolation do not use instrumentation and also allow for more precise reflection of methods' behavior in their types.

Polymorphic libraries The approach of this paper can be further extended to libraries that allow selective handling of exceptions. If the library user is willing to handle exceptions, the library fields and methods receive more restrictive labels. Similarly, if the library user prefers not to handle the exceptions, the library fields are labeled more permissively.

Inspired by optional methods [17, 4] we outline an idea for *optional exceptions*—a mechanism, that, together with label parametrization, such as in Jif, would allow library designers provide single code that is usable in both restrictive and permissive contexts. Optional exceptions can be declared as follows:

```
void m() throws Exc when (/*constraints*/) {...}
```

The `when` clause is followed by a list of label constraints such as $\ell \leq \ell'$. This specifies that the exception `Exc` is declared only when the corresponding constraints hold at the call site. Otherwise, if the constraints do not hold at the call site, this declaration is equivalent to a `failswith` declaration.

Jif classes can be parameterized over labels using keyword `label L`, where `L` is a parameter name; this parameter can appear in security annotations in the body of the class.

The following snippet illustrates how optional exceptions could be used with Jif label parametrization. Here `L`, `H`, and `R` are ordinary class parameters, where `R` has a helper role: it is an upper bound on how much the client's *pc* label gets tainted due to exceptions.

In the method `m()` the exception `ArithmeticException` has level `H`. The `when` clause specifies that for that exception to be catchable `R` needs to be instantiated to a label that is as restrictive as `H`.

```
class C [label L, label H, label R]  {
  int {L;R} x; // the annotation uses L and R
  int {H} h;   // high data

  void m{L;R}()throws ArithmeticException{H}
  when {H <= R}  {
    int t = 1/h; // may result in exception
    this.x ++;
  }
}
```

Such exception declaration is *optional* in the sense that if the constraint in the `when` clause is not satisfied at the call site, the exception is assumed to result in runtime error. In general, the more restrictive `R` is, the more exceptions the context needs to handle. In the most restrictive case, `R` can be set to top-most security label and an optional exception effectively becomes a mandatory one.

The two examples below illustrate how the above class can be used. In both examples `L` is instantiated to $\{\}$ — a label corresponding to most public and least trusted security level, and `H` is instantiated to some high label `High`, while `R` is different. In the first example, the caller prefers not to handle exceptions and sets `R` to $\{\}$. Note that the variable `obj.x` receives permissive security level, as exemplified by the last assignment in the example.

```
C[{}, {High}, {}] obj = new C();
obj.m();
int {} low = obj.x; // OK
```

In the second example, the caller is interested in handling exceptions, setting `R` to `High`. This is reflected in the type of variable `obj.x` that receives a more restrictive level now. An assignment, similar to the previous one, is illegal in this case.

```
C[{}, {High}, {High}] obj = new C();
try {  obj.m(); } catch (Exception e) {...}
// the following assignment is illegal
int {} low = obj.x;
```

7. Related work

Denning and Denning [16] informally discuss exception handling in the context of static information-flow analysis. They consider top-level exception declarations (in the style of the PL/I language), which prescribe actions to be executed, should an exception condition occur, before the control is returned back to the point of the exception. They acknowledge that it would be a practical solution to inhibit all traps except those that for which actions have been defined explicitly by the program, which is in the spirit of our solution. However, they do not consider local exception handling as in the `try` . . . `catch` blocks, nor they discuss the soundness of the approach.

Volpano and Smith [31] appear to be the first to prove soundness for a static security analysis for programs with exceptions. However, it is limited to *fatal* (uncatchable) exceptions. Further, the analysis imposes severe restrictions: secret data is not allowed to be involved in operations that might raise an exception. This is the price for ensuring termination-sensitive noninterference by the analysis.

Myers [23] proposes a more permissive mechanism, based on *path labels* for catchable exceptions. This mechanism is the core of exception handling in Jif [24]. The soundness of the approach is not discussed, however. Myers remarks that resource exhaustion conditions (such as out-of-memory and stack-overflow) should be treated as fatal, which should not be caught. Later versions of Jif disallow handling exceptions of class `Error`.

Pottier and Simonet [26, 27, 29] present a mechanism, similar to Jif's, and prove its soundness with respect to batch-job termination-insensitive noninterference in a functional setting. This mechanism is the core of exception handling in FlowCaml [30].

Barthe and Rezk [7] track exceptions in a JVM-like language. They consider a single type of catchable exceptions and prove batch-job termination-insensitive noninterference. Barthe et al. [6] extend this approach to multiple catchable exceptions. In a related effort, Barthe et al. [8] provide security-type preserving compilation of a source language with a single type of catchable exceptions to the low-level language of Barthe and Rezk [7].

Hedin and Sands [18] consider class-cast and null-pointer exceptions. They establish a batch-job termination-insensitive noninterference property for their security analysis.

To sum up, some [31] approaches consider only uncatchable exceptions, some [26, 27, 29, 7, 8, 18, 6] only catchable, and some [16, 23] even both, but none gives the programmers control over which exceptions should be handled.

On the secure programming side, our JifPoker [2] case study identifies patterns for secure programming with exceptions. JifClipse [20], an integrated environment for developing Jif code provides automatic insertion of `try . . . catch` blocks, although this is intended as a quick-fix feature.

Deng and Smith [15] propose array semantics where abnormal computation is replaced by default values. This coincides with our motivation to reduce the burden of error handling, but is different from our goal to preserve the original semantics of the program.

Demange and Sands [14] offer termination-insensitivity with respect to large secrets and termination-sensitivity with respect to small secrets in their notion of "secret-sensitive noninterference".

Refining termination-insensitive security and determining when it is adequate is an important problem. Further progress in quantitative information flow [10, 11, 13] could be of help here, but this is outside of the scope of this paper.

8. Conclusions

We have presented a permissive yet secure mechanism for exception handling in programs. By giving the programmer control over which exception should be handled, we provide an attractive possibility for reducing the burden of exception handling without loss of security. This opens up interesting possibilities for increasing the usability of such systems as Jif. We have showed that our approach scales for a language with procedures, multiple exception types, and output. The treatment of procedures is particularly appealing: procedure annotations for what exceptions can be handled and not by the callers provide an elegant compositional solution, which naturally extends Jif (which currently supports only the former).

As is common for Denning-style analyzes, our target security property is termination-insensitive noninterference. In earlier work [1] with Hunt and Sands, we have showed that the impact of leaks through abnormal termination is limited: the attacker cannot learn the secret in polynomial time in the size of the secret; and, for uniformly distributed secrets, the advantage the attacker gains when guessing the secret after observing a polynomial amount of output is negligible in the size of the secret.

Prospects for achieving termination-sensitive security without presence of exceptions do not appear to be realistic (unless Draconian restrictions are imposed [31]). Whenever there is a finite resource at disposal of the program (such as memory), exhaustion is possible, which leads to runtime errors. These errors can be used to encode information about secrets.

Future work is focused on incorporating our mechanism into Jif. We sketch some ideas on how to proceed in this direction in Section 6.4.

Acknowledgments Thanks are due to Stephen Chong, Daniel Hedin, Michael Hicks, Andrew Myers, and the anonymous reviewers for useful feedback. This work was funded by the Swedish research agencies SSF and VR, and, in part, by AFRL.

References

[1] 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*, pages 333–348, October 2008.

[2] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proc. European Symp. on Research in Computer Security*, volume 3679 of *LNCS*, pages 197–221. Springer-Verlag, September 2005.

[3] A. Askarov and A. Sabelfeld. Catch me if you can: Permissive yet secure error handling. Technical Report. Available at `http://www.cs.cornell.edu/~aslan/exceptions-tr.pdf`, 2009.

[4] J. A. Bank, A. C. Myers, and B. Liskov. Parameterized types for Java. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 132–145. ACM, 1997.

[5] J. Barnes and JG Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[6] 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.

[7] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In *Proc. ACM TLDI'05*, pages 103–112. ACM Press, 2005.

[8] 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.

[9] R. Chapman and A. Hilton. Enforcing security and safety models with an information flow analysis tool. *ACM SIGAda Ada Letters*, 24(4):39–46, 2004.

[10] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. In *QAPL'01, Proc. Quantitative Aspects of Programming Languages*, volume 59 of *ENTCS*. Elsevier, 2002.

[11] D. Clark, S. Hunt, and P. Malacaria. Quantified interference for a While language. In *QAPL'04, Proc. Quantitative Aspects of Programming Languages*, volume 112, pages 149–166, January 2005.

[12] M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: Toward a secure voting system. In *IEEE Symposium on Security and Privacy*, May 2008.

[13] M. R. Clarkson, A. C. Myers, and F. B. Schneider. Belief in information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 31–45, June 2005.

[14] D. Demange and D. Sands. All Secrets Great and Small. In *Proc. European Symp. on Programming*, LNCS, pages 239–253. Springer-Verlag, 2009.

[15] Z. Deng and G. Smith. Lenient array operations for practical secure information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 115–124, June 2004.

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

[17] B. Liskov et al. CLU reference manual. In *In Goos and Harmanis, editors*, volume 114 of *LNCS*. Springer-Verlag, Berlin, 1981.

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

[19] B. Hicks, K. Ahmadizadeh, and P. McDaniel. Understanding practical application development in security-typed languages. In *22st Annual Computer Security Applications Conference (ACSAC)*, Miami, Fl, December 2006.

[20] B. Hicks, D. King, and P. McDaniel. Jifclipse: Development tools for security-typed languages. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2007.

[21] D. King. JLift. Software release. Located at `http://www.cse.psu.edu/~dhking/jlift`, 2008.

[22] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Proc. International Conference on Information Systems Security (ICISS)*, volume 5352 of *LNCS*, pages 56–70. Springer-Verlag, December 2008.

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

[24] 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`, July 2001–2009.

[25] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *Proc. 12th International Conference on Compiler Construction, Warsaw, Poland, LNCS 2622*, pages 138âĂŞ–152, apr 2003.

[26] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, January 2002.

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

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

[29] V. Simonet. Fine-grained information flow analysis for a λ-calculus with sum types. In *Proc. IEEE Computer Security Foundations Workshop*, pages 223–237, June 2002.

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

[31] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. *Proc. IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

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