Deriving an Information Flow Checker and Certifying Compiler for Java *

Gilles Barthe[1] David Naumann[2] Tamara Rezk[1]

[1]INRIA Sophia-Antipolis, Project EVEREST,
{Gilles.Barthe,Tamara.Rezk}@sophia.inria.fr

[2]Stevens Institute of Technology
naumann@cs.stevens.edu

Abstract

Language-based security provides a means to enforce end-to-end confidentiality and integrity policies in mobile code scenarios, and is increasingly being contemplated by the smart-card and mobile phone industry as a solution to enforce information flow and resource control policies.

Two threads of work have emerged in research on language-based security: work that focuses on enforcing security policies for source code, which is tailored towards developers that want to increase confidence in their applications, and work that focuses on efficiently verifying similar policies for bytecode, which is tailored to code consumers that want to protect themselves against hostile applications. These lines of work serve different purposes—and thus have been developed independently—but connecting them is a key step towards the deployment of language-based security in practical applications.

This paper introduces a systematic technique to connect source code and bytecode security type systems. The technique is applied to an information flow type system for a fragment of Java with exceptions, thus confronting challenges in both control and data flow tracking.

1. Introduction

Security models for mobile code typically rely on typing mechanisms that statically enforce basic safety policies (e.g. code containment or absence of pointer arithmetic) and mechanisms such as stack inspection in Java and .NET that dynamically enforce access control policies. While these mechanisms are intended to enforce confidentiality and integrity policies, it is difficult to assess the end-to-end policies that they guarantee. In particular, they do not guarantee information-flow policies that are commonly desirable in scenarios involving the execution of untrusted code. To enforce these stronger security guarantees, information-flow type systems provide a means to track data flows and control dependencies, and thereby to reject any program that may leak confidential information.

JFlow is an extension of Java with a flexible and expressive information flow type system [11]. The flexibility and expressiveness of information flow policies supported by JFlow has been exploited for modeling and analyzing the policies that underlie battleship and mental poker games [2]. Despite the richness of its language, the information-flow policies of JFlow can be enforced automatically by a constraint-based algorithm that rejects programs that may violate their policy. However, the flexibility of the type system, especially with respect to declassification, makes it difficult to characterize formally the security properties that are verified by typable programs. Thus, subsequent work [4, 3] has focused on developing for a fragment of Java that includes objects, inheritance, and methods, an information-flow type system that enforces *non-interference*, a baseline policy that requires that secret input data cannot be inferred by an attacker that can only observe public output. (Non-interference also captures integrity, by duality.)

The benefits provided by JFlow (and [3]) do not directly address mobile code security since they apply to source code, whereas Java applets are deployed in compiled form as JVM bytecode programs. Thus, it is desirable to develop information flow type systems at bytecode level. An extended bytecode verifier is provided by [7]; their type system guarantees secure information flow for a fragment of Java bytecode that includes objects, inheritance, methods, and (a simplified mechanism for) exceptions.

The type systems of JFlow and of [7] have been developed and applied in isolation: JFlow offers a practical tool for developing secure applications, and in particular for ensuring to developers that applications meet high-level policies about API usage. In contrast, the type system of [7] augments the Java security architecture to provide assur-

* Work partially supported by IST Projects Mobius; by the ACI Sécurité SPOPS; by the RNTL project CASTLES; and by US NSF CCR-0208984 and CCF-0429894.

ance to users that applets respect high-level policies about API usage. The value of these two lines of work can be greatly increased by connecting them via a type preservation result, showing that programs typable in a suitable fragment of JFlow will be compiled into bytecode programs that pass information-flow bytecode verification (as suggested, e.g., by Abadi [1]). The interest of such a result is to show on the one hand that applications written with JFlow can be deployed in a mobile code architecture that delivers the promises of JFlow in terms of confidentiality, and on the other hand that the enhanced security architecture from [7] can benefit from practical tools for developing applications that meets the policy it enforces.

This paper makes the following contributions:

- We propose a systematic method to derive an information flow type system for a high-level language (here, an extension to exceptions of the fragment of Java in [3]) from an information flow type system for a low-level language (here, the fragment of the JVM in [7]) using a non-optimizing compiler [16] from Java source to bytecode. The novelty of the resulting type system, as compared to other type systems for high-level languages, is to rely on *security environments*: these assign security levels to control points instead of to code fragments. The flow of information via control is tracked by propagating the levels of guard conditions along control paths. This approach was previously used only for low-level languages. Another novelty induced by the presence of security environments at source level is the simplification of the information flow typing rules for the JVM, as the security environment can be retrieved from the source code and does not need to be computed at bytecode level. The result is a simpler and faster bytecode verification algorithm for information flow.
- We introduce a conventional type system for the same fragment of Java source code and show that each program typable in this *high level type system* is also typable in the *intermediate* type system based on security environments. Further, by combining this result with type-preserving compilation (the first item above), we show that a non-optimizing Java compiler [16] preserves information flow typing, in the sense that a program typable by the high-level system for source programs will be compiled into a program that will be accepted by a bytecode verifier that enforces secure information flow.

The only previous type preserving compiler for information flow is that of [6] for a simple imperative language. As [6] points out, an additional benefit of type preservation is that it proves noninterference for the source language type system, provided that the compiler preserves semantics (as it should) and noninterference has been shown for the target language. Thus another contribution of this paper is to add exceptions to the language in [3], but, instead of extending their soundness proof we can obtain noninterference for the high level type system using our type preservation result and the noninterference result of [7].

Besides [7], there has been some work on secure information flow for typed assembly languages. Medel at al. [10] treat a simple imperative assembly language, tracking implicit flows using code annotations so the typing rules can, in effect, recover structured control flow. A richer assembly language is treated by Yu and Islam [19] in work subsequent to ours. The language has indirect addressing/aliasing and code pointers but still not heap allocation or object-oriented features. Both works prove that their rules ensure noninterference; [19] also gives a security-type preserving translation from an imperative language with first order procedures.

Overview. The key technical idea is to connect unstructured bytecode with structured source code by way of an intermediate type system which applies to source code but tracks control dependence regions in terms of control flow labels just as it is done for bytecode in [7]. Although one of the contributions is a method to *derive* a type system for source code from one for target code, we only have space to present the result of this derivation. So for clarity we present the type systems top down. Section 2 reviews security policy and noninterference in the presence of exceptions and heap allocation; it also gives an overview of control dependence regions. Section 3 gives the source language and its high level security type system. Section 4 gives the key definition of the paper, a type system for source programs that deals with control flow using explicit control point labels. Section 5 connects these two type systems. Section 6 gives the target language and its security typing and defines the non-optimizing compiler. Section 7 connects the intermediate typing system with the target one and proves the main result. The concluding section discusses how the results can be adapted to optimizing compilers and richer policies such as those in JFlow.

2. Secure flow policy and control dependence regions

We assume given a lattice $\mathcal{S}$ of security levels. Confidentiality policy is specified by labeling with security levels the observable input and output channels of the program (in particular, static variables like `System.in` and `System.out` and also public method parameters and results in public APIs) [9]. In order to statically check conformance with policy, labels are needed on intermediate interfaces such as methods, fields, and local variables. It is straightforward (and implemented by the JFlow tool) to infer labels for local variables, and feasible to infer labels for fields and methods and to allow label polymorphism [17]. In this paper we take as given the result of inference, that is, we assume labels are given for all methods, fields, and locals.

Without loss of generality we assume that all field, and local variable names are distinct so that the label assignment can be given as a single function, Γ, from variable names to levels and a function, fieldlevel, from field identifiers to levels.

Security policy and noninterference. A policy determines a notion of indistinguishability for states. Indistinguishability is interpreted here in terms suitable for static checking of programs: it does not consider covert channels such as timing and power consumption. Roughly speaking, two states are indistinguishable for an observer at level k if they agree on variables and fields labeled k or below. A program is called *noninterferent* [14] if it conforms to the policy in the standard sense: given two initial states that are indistinguishable for the low observer, the corresponding pair of final states from two runs of the program are indistinguishable. In this paper we do not consider divergence to be observable. The low observer can distinguish between normal and abnormal termination (uncaught exception). To account for dynamic allocation and aliasing, indistinguishability is formalized using a bijective renaming for low-visible locations [3, 7].

Barthe et al. [7] prove noninterference for a bytecode type system very similar to the one used here, so we refrain from giving the formal details in the present paper. Note, however, that their formalization treats all abnormal states as indistinguishable (as if upon abnormal termination the attacker cannot read any of the state). In the present paper we consider that exceptions can carry information. Surely the low observer can test the type and even fields of an exception object; other parts of the state could therefore be observable. In fact, the type system of [7] ensures indistinguishability of the entire state even in the abnormal case (as can be seen by inspecting their proofs).

Our treatment of exceptions is rather conservative for the sake of notational simplicity: we do not distinguish the security levels of exceptions by type, but instead lump all exceptions together. For practical use, one would replace our single exception level by a set of levels, indexed by relevant exception type; this can then by tracked with relative precision since the try-catch construct declares the type of exceptions handled. In the full version of the paper we spell out this generalization, which is very similar to the treatment in JFlow [11], and formalize the details of indistinguishability where the low observer can read state even upon abnormal termination. The noninterference proof [7] is then adapted to the JVM rules in this paper.

In this version of the paper, exceptions are thrown for null dereferencing and by the explicit throw construct, but not by object construction. Whereas Java has an OutOfMemoryException that can be thrown by (new), we assume an unbounded heap since that is what is done in [7].

Control dependence regions. Tracking information flow via control flow in a structured language without exceptions is easy since the analysis can exploit control structure [18, 3]. Exceptions make tracking cumbersome due to the loss of structured control flow. Our high level type system tracks exception levels in a way similar to [8].

For unstructured low level code, such as Java bytecode, implicit flows can be tracked in terms of an analysis of *control dependence regions* which gives information about different blocks in the program due to conditional or exceptional instructions. This analysis can be statically approximated [7, 13].

To deal with this mismatch between tracking of implicit flows at source and bytecode level, we introduce an intermediate analysis that applies to source code but uses control dependence regions. One of the contributions of this paper is an inductive definition of control dependence regions for a language with exceptions. An important consequence of such an analysis for source programs is that, given a compiler from source programs to target programs, it is possible to obtain control dependence regions for the compiled program (Definition 7). In a scenario where the compiler is not trusted, it is possible to check that the regions given by the analysis based on compilation has the requisite properties for soundness of the information flow analysis (see Property 1, also in [7]).

Methods. For lack of space, this version of the paper also omits method calls. Their typing constraints are analogous to assignment statements, taking into account the security signature of the method. Signatures are used so that checking is modular, in particular, each method body is checked separately. We plan to make the extension in future work and do not expect it to be too difficult. It was done in [3] and [7] without exceptions, and the full version of [7] includes exceptions. The security signature of a method reflects the security context used in typing judgements (in particular, of the method body), in particular an upper bound on the result and lower bound on heap writes. A method signature would also give a bound on the level of information that can escape via uncaught exceptions. These constraints would be reflected in the rule for the return statement as well constraints involving handlers, in rules for exception-throwing commands like field update.

3. High level security type system

In this section we define a high level type system for the source language. By high level we mean that it consists of syntax-directed rules together with subsumption rules. It enforces the policy specified by a mapping, Γ, from variable names in the program to security levels and a mapping, fieldlevel, from field names to security levels.

Source language. Exceptions in Java can be explicitly programmed and are also thrown by expressions such as field access (null dereference) and type cast (cast failure). We use a desugared source language in which exceptions can only occur in specific syntactic forms. This helps simplify the formalization in section 4 where we attach control flow labels to commands that can branch due to exceptions. We also require the main program to end with a return, to simplify the definition of control flows. Any command can be desugared to this form using additional local variables.

The grammar is as follows; x represents any variable name, v a literal value, C a class name, f a field name, and op represents an arithmetic or boolean operation.

$$
\begin{array}{rcl}
e & ::= & x \mid v \mid e\ op\ e \mid \mathsf{new}\ C \\
c & ::= & x := e \mid x := e.f \mid e.f := e \mid \mathsf{throw}\ e \mid \\
 & & \mathsf{return}\ e \mid c; c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \\
 & & \mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{try}\ c\ \mathsf{catch}(X\ x)\ c \\
\mathit{program} & ::= & \mathsf{Prog}\ (\vec{x})\{c; \mathsf{return}\ e\}
\end{array}
$$

Here $\vec{x}$ is the set of variables used in the body of the program. We use the term *exception-throwing* for commands of the form $e.f := e'$ or $x := e.f$ or $\mathsf{throw}\ e$. Handlers for exceptions appear in the catch part of a try-catch command. For simplicity there is a single class, named X, for exceptions.

Type system. In presence of exceptions, commands can have multiple exits and the region of a branch need not be contained within the command. We define *high level judgements* $\vdash c : k_1, k_2$ with the meaning that c is secure and writes variables/fields of level at least k_1. Moreover, the information revealed by the termination mode (i.e., whether there is an exception and if so the information carried by the exception object) is at most k_2. Furthermore, the judgement allows k_2 to be $\emptyset$ which indicates that no exception can escape from the command.

Figure 1 gives the typing rules for source language expressions. These are used both in the high level typing system and in the intermediate system. Figure 2 gives the high level typing rules for source programs.

Notice that commands such as $x := e$ or $\mathsf{return}\ e$ are typed with exception effect $\emptyset$, meaning that these commands cannot throw exceptions.

We write $\leq$ for the lattice ordering on the set $\mathcal{S}$ of security levels and $\sqcup$ for least upper bounds. The latter is extended to $\emptyset$ by defining $\emptyset \sqcup k = k$. We write H (respectively, L) for the top secret level (resp. most public level). In the case of exception-throwing commands, the exception effect is given by the type of the expression which might cause the exception to be thrown, e.g., in the case $x := e.f$ the exception effect k is the type of expression e (see rule [ASSIGN2]). The exception effect is used to impose constraints on successor commands, e.g., in $c; c'$ the exception effect of c must be less or equal than the write effect of c' (see $k_1 \leq k'_1$). In general, the exception effect of a command restricts the write effect of its successor commands, except in the case of try-catch. In the [CATCH] rule, the exception effect for the command in the try part imposes a constraint on the type of variable x (that stores the exception object) and imposes a constraint on the write effect of the code of the handler (catch part). If the catch part cannot throw an exception, i.e., its exception effect is $\emptyset$, there are no constraints on the write effects of successor commands of try-catch.

The following example illustrates how exception-throwing commands inside while commands can lead to information leaks.

Example 3.1 (exceptions in while) *Let x and x' be high variables and y be low. The program*

$$\mathsf{while}\ x \leq 3\ \mathsf{do}\ \{x' := y.f; x := x + 1;\}\ \mathsf{return}\ e$$

is interferent because $x' := y.f$ can throw an exception and there is no handler for it. Suppose that the variable y is initially null. Then the program terminates in an abnormal state if the (high) expression $x \leq 3$ is true and it terminates in a normal state if $x \leq 3$ is false. This program will be rejected by the [WHILE] rule, which does not allow low exceptions in high environments (constraint $k' = k$). Now assume that variable x is low and x' and y are high. Again the program is interferent: A high exception can be thrown so the program will terminate abnormally depending on the value of high variable y. This program will also be rejected by the [WHILE] rule.

The following lemma is useful; it says that any subcommand of a command c has at least the same write effect as that of c.

Lemma 3.1 *Suppose $\vdash c : k_1, k_2$ and $\vdash c' : k'_1, k'_2$. Let c' be a subcommand of c. Then $k_1 \leq k'_1$ and if c' has no handler inside c then $k'_2 \leq k_2$.*

Example 3.2 (Example with try-catch) *Consider the command*

$$\mathsf{try}\ \mathsf{throw}\ x;\ x := 1;\ \mathsf{catch}\ (X\ y)\ y := \mathit{null}$$

where variable x is high and variable y is low. This is interferent, indeed its effect is the same as the direct assignment $y := x$. It is rejected by the type system because of the [CATCH] rule, where the exception effect of the try part (here, H) has to be less or equal than the type of the variable in catch (here, L).

4. Intermediate type system for source code

In this section we introduce another type system for the source language. The *intermediate type system* serves as a bridge between the high level type system and the type system for the target language.

Source labels and control flows. To name *program points* where control flow can branch or writes can occur, we add natural number labels to the source syntax —not to be confused with security labels given by Γ and $\mathsf{fieldlevel}$.

$$
\begin{array}{rcl}
c & ::= & [x := e]^n \mid [x := e.f]^n \mid [e.f := e]^n \mid [\mathsf{throw}\ e]^n \mid \\
 & & [\mathsf{return}\ e]^n \mid c; c \mid [\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c]^n \mid \\
 & & [\mathsf{while}\ e\ \mathsf{do}\ c]^n \mid [\mathsf{try}\ c\ \mathsf{catch}(X\ x)\ c]^n
\end{array}
$$

$$\frac{}{\vdash x : \Gamma(x)}\ \textsc{Var} \qquad \frac{}{\vdash v : L}\ \textsc{Val} \qquad \frac{}{\vdash \mathit{new}\ C : L}\ \textsc{New} \qquad \frac{\vdash e : k_2 \quad \vdash e' : k_1}{\vdash e\ \mathit{op}\ e' : k_1 \sqcup k_2}\ \textsc{Op} \qquad \frac{\vdash e : k \quad k \le k'}{\vdash e : k'}\ \textsc{Subsume}$$

Figure 1. High level typing rules for source language expressions, using policy Γ.

$$\frac{\vdash e : k \quad k \le \Gamma(x)}{\vdash x := e : \Gamma(x), \emptyset}\ \textsc{Assign} \qquad \frac{\vdash c : k, k_1 \quad \vdash c' : k', k_1' \quad k_1 \le k'}{\vdash c; c' : k \sqcap k', k_1 \sqcup k_1'}\ \textsc{Seq} \qquad \frac{\vdash e : k \quad \vdash c : k, k' \quad k' \neq \emptyset \Rightarrow k = k'}{\vdash \mathsf{while}\ e\ \mathsf{do}\ c : k, k'}\ \textsc{While}$$

$$\frac{\vdash e : k \quad \vdash c : k, k' \quad \vdash c' : k, k' \quad k' \neq \emptyset \Rightarrow k \le k'}{\vdash \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c' : k, k'}\ \textsc{Cond} \qquad \frac{\vdash e : k \quad k \le \mathsf{fieldlevel}(f) \le \Gamma(x)}{\vdash x := e.f : \Gamma(x), k}\ \textsc{Assign2}$$

$$\frac{\vdash e : k \quad \vdash e' : k' \quad k \sqcup k' \le \mathsf{fieldlevel}(f)}{\vdash e.f := e' : \mathsf{fieldlevel}(f), k}\ \textsc{Update} \qquad \frac{\vdash c : k, k_1 \quad \vdash c' : k, k_1' \quad k_1 \le k \sqcap \Gamma(x) \quad k_1' \neq \emptyset \Rightarrow k_1 \le k_1'}{\vdash \mathsf{try}\ c\ \mathsf{catch}\ (X\ x)\ c' : k, k_1'}\ \textsc{Catch}$$

$$\frac{\vdash e : k}{\vdash \mathsf{throw}\ e : H, k}\ \textsc{Throw} \qquad \frac{\vdash e : L}{\vdash \mathsf{return}\ e : L, \emptyset}\ \textsc{Return} \qquad \frac{\vdash c : k_1, k_2 \quad k_1' \le k_1 \quad k_2 \le k_2'}{\vdash c : k_1', k_2'}\ \textsc{Subsume2}$$

Figure 2. High level typing rules for source language commands, using policy Γ and fieldlevel.

The notation for labels of compound commands is convenient but visually misleading in that the label pertains to the branching point in the control flow graph, for if and while, and the start of the handler for try-catch.

By contrast with Nielson et al. [12], we do not need to label expressions. The significance of labeling commands will become clear later in the paper, when we give a definition for control dependence regions for programs as mapping from branching commands (such as if, while or exception-throwing commands) to sets of labels, corresponding to commands included in their regions.

The function labels takes a command and returns all the labels of its subcommands, e.g. $\mathsf{labels}([x := e]^n) = \{n\}$, and $\mathsf{labels}(c; c') = \mathsf{labels}(c) \cup \mathsf{labels}(c')$.

Labels on program points are used in *intermediate judgements* of the form $\vdash c : E$. Here E is a *security environment*, i.e., a function $E : \mathsf{labels}(SP) \to \mathcal{S}$ that assigns levels to all program points of the program. A source program SP is typable, written $\vdash SP : E$, if its command part is typable with respect to E according to the rules given in Figure 4. Roughly, the idea is that for a field or variable assignment with control label n, the field or variable must have level at least $E(n)$. Note that the rules recurse on the structure of constituent commands c of SP, but a single E is used throughout.

If R is a set of program points then $\mathrm{lift}_k(E, R)$ is the security environment E' such that $E'(n) = E(n)$ for $n \notin R$ and $E'(n) = k \sqcup E(n)$ for $n \in R$. The constraints in the rules involve control dependence regions (sregion), which we now proceed in several steps to define.

We use the notation $C[-]$ to denote context of a command. We use square brackets both for labeling and for contexts; it should be clear that $[c]^n$ without a capital letter in front means that command c has label n, whereas $C[c]$ with a capital C in front means $C[-]$ is the context of command c.

We define here a successor relation between commands in the language. Following Nielson et al. [12] we define for each labeled command the *init*, *final*, and *except* labels; the set *except*(n) are the labels from which there can be an —uncaught— exception.

Given a source program SP, $n \mapsto n'$ denotes that command labeled n' is a successor of command labeled n in the control flow graph. We define $n \mapsto n'$ iff $(n, n') \in \mathit{flow}(c)$. The latter is defined in two steps, which are given in Figure 3. Let $\mapsto^+$ (resp. $\mapsto^\star$) denote the transitive (resp. transitive and reflexive) closure of $\mapsto$.

Definition 1 (branching commands, $\mathcal{LL}^\sharp$) *The* branching commands *are those of the form* if e then c_1 else c_2, while e do c_1, $x := e.f$, *and* $e.f := e$. *The set* $\mathcal{LL}^\sharp$ *is all the labels of branching commands in the program.*

For example, if the source program SP is

$$[\ \mathsf{if}\ x\ \mathsf{then}\ [x := x]^4\ \mathsf{else}\ [x := x]^7\ ]^2; [\mathsf{return}\ x]^9$$

c	$init(c)$	$final(c)$	$except(c)$
$[x := e]^n$	n	$\{n\}$	$\emptyset$
$[x := e.f]^n$	n	$\{n\}$	$\{n\}$
$[e.f := e]^n$	n	$\{n\}$	$\{n\}$
$[\mathsf{throw}\ e]^n$	n	$\emptyset$	$\{n\}$
$c_1; c_2$	$init(c_1)$	$final(c_2)$	$except(c_1) \cup except(c_2)$
$[\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]^n$	n	$final(c_1) \cup final(c_2)$	$except(c_1) \cup except(c_2)$
$[\mathsf{while}\ e\ \mathsf{do}\ c_1]^n$	n	$\{n\}$	$except(c_1)$
$[\mathsf{try}\ c_1\ \mathsf{catch}(X\ x)\ c_2]^n$	$init(c_1)$	$final(c_1) \cup final(c_2)$	$except(c_2)$
$[\mathsf{return}\ e]^n$	n	$\{n\}$	$\emptyset$

c	$flow(c)$
$[x := e]^n$	$\emptyset$
$[x := e.f]^n$	$\emptyset$
$[e.f := e]^n$	$\emptyset$
$[\mathsf{throw}\ e]^n$	$\emptyset$
$c_1; c_2$	$flow(c_1) \cup flow(c_2) \cup \{(n, init(c_2)) \mid n \in final(c_1)\}$
$[\mathsf{try}\ c_1\ \mathsf{catch}(X\ x)\ c_2]^n$	$flow(c_1) \cup flow(c_2) \cup \{(n', n) \mid n' \in except(c_1)\} \cup \{(n, init(c_2))\}$
$[\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]^n$	$flow(c_1) \cup flow(c_2) \cup \{(n, init(c_1)), (n, init(c_2))\}$
$[\mathsf{while}\ e\ \mathsf{do}\ c_1]^n$	$flow(c_1) \cup \{(n, init(c_1))\} \cup \{(p, n) \mid p \in final(c_1)\}$
$[\mathsf{return}\ e]^n$	$\emptyset$

Figure 3. Forward flows.

then $\mathcal{LL}^\sharp$ is $\{2\}$.

Definition 2 (inner-most handler) *Consider an exception-throwing command $[c]^n$ in program SP. Then an* inner-most handler decomposition *of $[c]^n$ consists of contexts $C_1[-]$, $C_2[-]$, command c', and label t such that*

$$SP \equiv C_1[[\mathsf{try}\ C_2[[c]^n]\ \mathsf{catch}\ (X\ x)\ c']^t]$$

and $C_2[-]$ does not have a try-catch *that encloses $[c]^n$ in its try part. We say that t is the* inner-most handler *of n and c' is the* handler *for c.*

For any $[c]^n$, either there is no handler for exceptions thrown by c, or there is a unique inner-most handler decomposition.

Definition 3 (handler function) *We define a function* sHandler *as follows: for label n it returns the label of the inner-most handler of n, if there is one; otherwise* $\mathsf{sHandler}(n)$ *is undefined, denoted* $\mathsf{sHandler}(n) \uparrow$.

Example 4.1 (inner-most handler) *In the program below,* $\mathsf{sHandler}(n) = t$.

$$[\mathsf{try}\ ([\mathsf{try}\ [c]^n\ \mathsf{catch}\ (X\ x)\ c']^t)\ \mathsf{catch}\ (X\ y)\ c'']^{n'}; [\mathsf{return}\ e]^{n''}$$

Any program that has high uncaught exceptions is rejected by our rules. Of course this would be refined with the inclusion of methods: uncaught exceptions are then allowed but bounded by the method signature.

Control dependence regions. Control dependence regions are used by the intermediate type system to impose constraints on commands depending on branching instructions, just as they are in the target system (Section 6). For example, in the high level type system, if the expression e in command $[\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c']^n$ is typable as $\vdash e : k$ then c and c' can only assign variables of level at least k. The intermediate system expresses this constraint by $E = \mathsf{lift}_k(E, \mathsf{sregion}(n))$, which means that every command in the region of n (including commands in branches depending on exceptions) has security level at least k.

Definition 4 (sregion and $\rhd$) *The region of a labeled command $[c]^n$ in source program SP with $n \in \mathcal{LL}^\sharp$ is written* $\mathsf{sregion}(n)$ *and defined to be the set of all labels n' such that $n \rhd n'$. Here $\rhd$ is defined inductively as follows.*

– *If $[c]^n$ is an exception-throwing command with* $\mathsf{sHandler}(n) \uparrow$ *and $n \mapsto^+ n'$ then $n \rhd n'$.*

$$\frac{\vdash e : k \qquad k \sqcup E(n) \leq \Gamma(x)}{\vdash [x := e]^n \ : E}\ \textsc{Assign} \qquad \frac{\vdash c : E \qquad \vdash c' : E}{\vdash c ; c' \ : E}\ \textsc{Seq} \qquad \frac{\vdash e : k \qquad \vdash c \ : \ E \qquad E = \mathrm{lift}_k(E, \mathsf{sregion}(n))}{\vdash [\mathsf{while}\ e\ \mathsf{do}\ \ c]^n \ \ : \ E}\ \textsc{While}$$

$$\frac{\vdash e : k \qquad \vdash c : E \qquad \vdash c' : E \qquad E = \mathrm{lift}_k(E, \mathsf{sregion}(n))}{\vdash [\mathsf{if}\ e\ \mathsf{then}\ \ c\ \mathsf{else}\ c']^n \ \ : \ \ E}\ \textsc{Cond}$$

$$\frac{\vdash e : k \qquad E(n) \sqcup \mathsf{fieldlevel}(f) \leq \Gamma(x) \qquad \begin{array}{c} k \leq \mathsf{fieldlevel}(f) \\ E = \mathrm{lift}_k(E, \mathsf{sregion}(n)) \end{array} \qquad \mathsf{sHandler(n)} \uparrow \Rightarrow E(n) \sqcup k = L}{\vdash [x := e.f]^n \ : E}\ \textsc{Assign2}$$

$$\frac{\vdash e \ : \ k \qquad \vdash e' \ : \ k' \qquad k \sqcup k' \sqcup E(n) \leq \mathsf{fieldlevel}(f) \qquad E = \mathrm{lift}_k(E, \mathsf{sregion}(n)) \qquad \mathsf{sHandler(n)} \uparrow \Rightarrow E(n) \sqcup k = L}{\vdash [e.f := e']^n \ : \ E}\ \textsc{Update}$$

$$\frac{\vdash c : E \qquad \vdash c' : E \qquad E(n) \leq \Gamma(x)}{\vdash [\mathsf{try}\ c\ \mathsf{catch}\ (X\ x)\ c']^n \ \ : \ E}\ \textsc{Catch} \qquad \frac{\vdash e : k \qquad E(n) \sqcup k = L}{\vdash [\mathsf{return}\ e]^n : \ E}\ \textsc{Return}$$

$$\frac{\vdash e : k \qquad \mathsf{sHandler}(n) \uparrow \Rightarrow E(n) \sqcup k = L \qquad \mathsf{sHandler(n)} = n' \Rightarrow k \leq E(n')}{\vdash [\mathsf{throw}\ e]^n : \ E}\ \textsc{Throw}$$

Figure 4. Intermediate typing rules for high-level language commands.

– *If $[c]^n$ is an exception-throwing command with an inner-most handler decomposition*

$$C_1[[\mathsf{try}\ C_2[[c]^n]\ \mathsf{catch}\ (X\ x)\ c']^t]$$

then there are two sub-cases:

(1) *if $n \mapsto^+ n'$, $n' \in \mathsf{labels}(C_2[[c]^n])$ then $n \rhd n'$;*

(2) *if $d \in \mathsf{labels}(c') \cup \{t\}$ then $n \rhd d$.*

– *If $[c]^n$ is of the form $[\mathsf{if}\ e\ \mathsf{then}\ \ c_1\ \mathsf{else}\ c_2]^n$ and $d \in \mathsf{labels}(c_1) \cup \mathsf{labels}(c_2)$ then $n \rhd d$.*

– *If $[c]^n$ is of the form $[\mathsf{while}\ e\ \mathsf{do}\ \ c_1]^n$ and $d \in \mathsf{labels}(c_1)$ then $n \rhd d$.*

– *If $n \rhd d$ and $d \rhd d'$ then $n \rhd d'$.*

Note that any label in the region of an exception-throwing command is either inside the try part of the inner-most handler, or is a successor of the code in the catch part.

5. Connecting the high level and intermediate type systems

This section shows that if labeled source program SP is typable in the high level system then it is typable in the intermediate system as well. (We apply the high level typing system to labeled commands, by simply ignoring the labels.) In order to do this, we first show how a security environment E for SP can be obtained from the typing derivation in the high level system.

Let D be a typing derivation for SP in the high level type system. For each constituent c of SP there is an instance of an introduction rule with conclusion $\vdash c : k, k'$ for some k, k' (as opposed to uses of the subsumption rule). In this case we say the type k, k' of c is *given by its intro judgement* and write $D :: \vdash c : k, k'$. When the types k, k' are irrelevant, we use $D :: \vdash c$ to mean that SP is typable with derivation tree D and c occurs in SP.

The intro judgement for a subcommand reflects the essential constraints for security of this command in its context. Subsumption serves to weaken typing information, e.g., to match the two branches of a conditional in high level rule Cond, but it loses precision. So we define the security environment E in terms of intro judgements.

Definition 5 (environment E from high level typing) *Let D be a typing derivation for a source program SP in the high level type system. Define security environment $E : \mathsf{labels}(SP) \rightarrow \mathcal{S}$ as follows:*

- *If n belongs to some region of a branching label n' of a command c' in SP such that the intro judgement for c' types it with write effect or exception effect H, then $E(n)$ is defined as the write level of the intro judgement for $[c]^n$ in D. That is, if $D :: \vdash c : k, k'$ then $E(n) = k$.*
- *If n does not belong to any high region in SP, then $E(n) = L$.*

Example 5.1 (obtaining E from D) *Let the source code c be*

$$[\mathsf{if}\ y_H = 0\ \mathsf{then}\ [y_H := x_L]^6\ \mathsf{else}\ [y_H := 1]^9\ ]^4;\ [\mathsf{new}\ C.f_L := 3]^{12}$$

(Labels are chosen for compilation compatibility, as will be defined in later sections) Let $\Gamma(x_L) = L$, $\Gamma(y_H) = H$ and $\mathsf{fieldlevel}(f_L) = L$. The type for c is L, L. The derivation tree in the high level system shown in Fig. 5.1. The security environment is $E(4) = H$, $E(6) = H$, $E(9) = H$, and $E(12) = L$.

The following lemma states a relation between types of commands in the high level type systems and regions.

Lemma 5.1 *Let $[c]^n$ be an exception-throwing command. Let $n' \in \mathsf{sregion}(n)$ and let $D ::\vdash [c']^{n'} : k'_1, k'_2$ be the intro judgement for n' in derivation D of a program P and let $D ::\vdash [c]^n : k_1, k_2$ be the intro judgement for $[c]^n$. Then $k_2 \leq k'_1$.*

Lemma 5.2 (exception effect and region) *Let c be an exception-throwing command. Let $D ::\vdash [c]^n : k, k'$ (typable according to the corresponding intro judgement for c) and E be the security environment derived from D then $E = \mathsf{lift}_{k'}(E, \mathsf{sregion}(n))$.*

Now we can prove, by induction on the structure of source commands, that every program typable in the high level system is also typable in the intermediate system.

Theorem 5.3 *If $D ::\vdash c$ then $\vdash c : E$, where E is obtained from D by Definition 5.*

Example 5.2 (Preservation of types) *Recall command c of Example 5.1, its type derivation and its derived E. According to Definition 4, $\mathsf{sregion}(4) = \{6, 9\}$. The derivation tree using the intermediate type system and E from Example 5.1 is given in Figure 5.2. It is easy to see that constraint $E = \mathsf{lift}_H(E, \mathsf{sregion}(4))$ is satisfied since $E(6) = H$ and $E(9) = H$. The constraint $E = \mathsf{lift}_L(E, \mathsf{sregion}(12))$ holds trivially since 12 has no successors, and thus its region is empty.*

6. Target language: type system and compilation

The type system for the target language is compatible with bytecode verification and is adapted from [7]. Whereas the system in [7] specifies an algorithm for finding a security environment, ours simply works with a fixed security environment for bytecode which we derive from the source code type environment.

The instruction set we consider is a subset of the Java Virtual Machine and is given below. As in the source language, v ranges over literal values, x over variables, op over primitive operations. Also, j ranges over integers.

$$\begin{aligned} instr ::= {} & \mathsf{prim}\ op \mid \mathsf{push}\ v \mid \mathsf{load}\ x \mid \mathsf{store}\ x \mid \mathsf{ifeq}\ j \mid \mathsf{goto}\ j \\ & \mid \mathsf{new}\ C \mid \mathsf{getfield}\ f \mid \mathsf{putfield}\ f \mid \mathsf{throw}\ e \mid \mathsf{return} \end{aligned}$$

A program in the target language consists of an instruction list $TP[1..N]$ together with an exception table that defines the handlers protecting points of the program's body. We let $TP[i]$ be the ith instruction in TP and write $TP[i..j]$ for the subsequence from i to j inclusive. We let $\mathcal{PP}$ be the set of program points, i.e., $\{1, \ldots, N\}$. A handler is a triple $\langle i, j, t\rangle$; it transfers control to address t if an exception is thrown by an instruction in the interval $[i, j)$. We define partial function $\mathsf{Handler} : \mathcal{PP} \rightharpoonup \mathcal{PP}$ that, given a program point in the program's body, returns its inner-most handler if it exists.

Type system. The analysis is expressed as an abstract transition system that is parameterized by a function $\mathsf{tregion}$ defined on program points of branching instructions, i.e., on the set $\mathcal{PP}^\sharp = \{i \in \mathcal{PP} \mid TP[i] = \mathsf{ifeq}\ j \vee TP[i] = \mathsf{putfield}\ f \vee TP[i] = \mathsf{getfield}\ f\}$.

The successor relation $\mapsto \subseteq \mathcal{PP} \times \mathcal{PP}$ of a program TP is defined by the clauses:

- if $P[i] = \mathsf{goto}\ j$, then $i \mapsto j$;
- if $P[i] = \mathsf{ifeq}\ j$, then $i \mapsto i + 1$ and $i \mapsto j$;
- if $P[i] = \mathsf{return}$, then i has no successors, which we write $i \mapsto$;
- $p[i] = \mathsf{throw}$ and $\mathsf{Handler}(i) = t$, then $i \mapsto t$; otherwise, $i \mapsto$;
- if $p[i] = \mathsf{putfield}\ f$, or $p[i] = \mathsf{getfield}\ f$ then $i \mapsto i+1$, and if $\mathsf{Handler}(i) = t$, then $i \mapsto t$.
- otherwise, $i \mapsto i + 1$.

In Definition 6 we use the source analysis to give a specific $\mathsf{tregion}$ function, but the target rules are defined for any $\mathsf{tregion} : \mathcal{PP}^\sharp \to \wp(\mathcal{PP})$ and $\mathsf{junc} : \mathcal{PP}^\sharp \rightharpoonup \mathcal{PP}$ that respectively compute the control dependence region and the junction point of an instruction at a given program point, provided these functions satisfy the following safe over approximation property (SOAP) [7].

Property 1 (SOAP Property) *Let $i \in \mathcal{PP}^\sharp$.*

- *If $i' \mapsto i''$, and $i' \in \mathsf{tregion}(i)$ or $i' = i$, then $i'' \in \mathsf{tregion}(i)$ or $i'' = \mathsf{junc}(i)$;*
- *If $i \mapsto^\star i'$, and $i' \in \mathcal{PP}^\sharp \cap \mathsf{tregion}(i)$ then $\mathsf{tregion}(i') \subseteq \mathsf{tregion}(i)$;*
- *If $\mathsf{junc}(i)$ is defined, then $\mathsf{junc}(i) \notin \mathsf{tregion}(i)$ and for all i' such that $i \mapsto^\star i'$ then $i' \mapsto^\star \mathsf{junc}(i)$ or $\mathsf{junc}(i) \mapsto^\star i'$.*

The abstract transition system manipulates *stack types*, st, which record the security level of values in the operand stack, and constrains the *security environment*, se, which maps program points to security levels. An abstract transition, written $i \vdash st, se \Rightarrow st', se$, constrains an instruction $TP[i]$ and its successor $TP[j]$ where $i \mapsto j$. The rules also constrain instructions without successors, i.e., return and unhandled exceptions.

Constraints on the transitions prevent bad direct flows and also indirect flows (by forbidding assignments to low variables in high regions). The abstract transition rules are shown in Figure 6.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{\dfrac{\Gamma(y_H)=H}{y_H : H} \quad \dfrac{}{0 : L}}{y_H = 0 : H}
\quad
\dfrac{\dfrac{\dfrac{\Gamma(x_L)=L}{x_L : L}}{x_L : H} \quad \Gamma(y_H)=H}{y_H := x_L : H\,,L}
\quad
\dfrac{\dfrac{}{1 : L} \quad \Gamma(y_H)=H}{y_H := 1 : H\,,L}
}{\text{if } y_H = 0 \text{ then } y_H = x_L \text{ else } y_H := 1 : H\,,L}
}{\text{if } y_H = 0 \text{ then } y_H = x_L \text{ else } y_H := 1 : L\,,L}
\quad
\dfrac{\dfrac{}{\text{new } C : L} \quad \text{fieldlevel}(f_L) = L \quad \dfrac{}{3 : L}}{\text{new } C.f_L := 3 : L,\ L}
}{\vdash \ \text{if } y_H = 0 \text{ then } y_H := x_L \text{ else } y_H := 1\,;\ \text{new } C.f_L := 3 :\ L,\ L}
$$

Figure 5. Derivation for Example 5.1, using high level rules.

$$
\dfrac{
\dfrac{
\dfrac{\dfrac{\dfrac{\Gamma(y_H)=H}{y_H : H} \quad \dfrac{}{0 : L}}{y_H = 0 : H} \quad \dfrac{\dfrac{\Gamma(x_L)=L}{x_L : L}}{x_L : H}}{y_H := x_L : E}
\quad
\dfrac{\dfrac{}{1 : L} \quad E(9) \leq H}{[y_H := 1]^9 : E}
\quad
E = \text{lift}_H(E, \text{sregion}(4))
}{[\text{if } y_H = 0 \text{ then } y_H = x_L \text{ else } y_H := 1]^4 : E}
\quad
\dfrac{\begin{array}{c} 3 : L \\ \text{new } C : L \\ E(12) \leq \text{fieldlevel}(f_L) \\ E = \text{lift}_L(E, \text{sregion}(12)) \\ \text{sHandler}(12) \uparrow\Rightarrow E(12) = L \end{array}}{[\text{new } C.f_L := 3]^{12} :\ E}
}{\Gamma \vdash \ [\text{if } y_H = 0 \text{ then } [y_H := x_L]^6 \text{ else } [y_H := 1]^9\,]^4;\ [\text{new } C.f_L := 3]^{12} :\ E}
$$

Figure 6. Derivation for Example 5.2, using intermediate rules.

Low assignments in high regions are prevented by the constraint $se = \text{lift}_k(se, \text{tregion}(i))$ in the rules for ifeq and for exception-throwing instructions.

A program TP is typable, denoted $\vdash TP$, if there is an se such that a dataflow analysis based on the abstract transition system yields a solvable system of constraints on stack types. Moreover, there is an assignment of stack types st to program points, such that for all $i, j \in \mathcal{PP}$ and all flows $i \mapsto j$ we have $i \vdash st_i, se \Rightarrow st_j, se$ and if $i \mapsto$ then $i \vdash st_i, se \Rightarrow$. Note, e.g., that getfield has a normal flow and also an exceptional flow if there is a handler; different rules cater to these cases.

The type system presented in this section is very similar to the one in [7], with the following differences. First, there are some simplifications since we omit method calls (cf. the rule for return. A second difference is the constraint $se = \text{lift}_k(se, \text{tregion}(i))$ appearing in the rules (Figure 7). In the type system of [7], se is not fixed but rather changed though the abstract transitions, to facilitate calculating it by data flow algorithms. Here we simplify the system by fixing se, since it is derived from E; fixing it clarifies the connection between the intermediate type system and the target type system. (We retain the before/after notation to make it easier to compare the two sets of rules.) Thirdly, in the type system of [7], types are polyvariant, i.e. for each program point in the program is assigned a set of stack types. We do not need polyvariance, since programs obtained by compilation have the property that operand stacks at junction points are always empty.

Compilation. Compilation is done by a function, $\mathcal{W}$, from source programs to target programs. The compiler is based on [16].

The compilation function from source programs to target programs $\mathcal{W} : \text{Prog} \rightarrow \text{Prog}_c$ is defined from a compilation function on expressions $\mathcal{E} : \text{Expr} \rightarrow \text{Instr}^\star$, and a compilation function on commands $\mathcal{S} : \text{Comm} \rightarrow \text{Instr}^\star$. Their formal definitions are given in Figure 8. The compilation of every labeled command includes a *primary instruction* —e.g., getfield is the primary instruction for a field access $[x := e.f]^n$— and these are indicated in the definition of the compiler. Exception tables are defined in Figure 9. In order to enhance readability, we use :: both for consing an element to a list and concatenating two lists, and we omit details of calculating program points (we use pc to represent current program point) in the clauses for while and if expressions. We also use # to denote the length of a list.

For a complete source program P with body c; return e, we define the compilation $\mathcal{W}(P)$ to be $\mathcal{S}(c;\text{return } e)$.

We assume *label compatibility*: the label of a source command is the same as the label of the program point of the primary instruction in its compilation, e.g., if getfield is obtained by compilation of $[x := e.f]^n$, its corresponding program point in the target program is n. This implies that $\mathcal{LL}^\sharp = \mathcal{PP}^\sharp$ for a source program and its compilation. The simplifying assumption loses no generality since source labels can be read off the compilation (as was done in the example just after Definition 1).

Given the definition of regions for the source program, we obtain regions for its compilation as follows.

$$
\begin{array}{rcl}
\mathcal{E}(x) & = & \mathsf{load}\ x \\
\mathcal{E}(n) & = & \mathsf{push}\ n \\
\mathcal{E}(e\ op\ e') & = & \mathcal{E}(e) :: \mathcal{E}(e') :: \mathsf{prim}\ op \\
\mathcal{S}(x := e.f) & = & \mathcal{E}(e) :: \underline{\mathsf{getfield}\ f} :: \mathsf{store}\ x \\
\mathcal{E}(\mathsf{new}\ C) & = & \mathsf{new}\ C \\
\mathcal{S}(x := e) & = & \mathcal{E}(e) :: \underline{\mathsf{store}}\ x \\
\mathcal{S}(c_1;\ c_2) & = & \mathcal{S}(c_1) :: \mathcal{S}(c_2) \\
\mathcal{S}(\mathsf{while}\ e\ \mathsf{do}\ c) & = & \quad let \quad le = \mathcal{E}(e); lc = \mathcal{S}(c); \\
 & & \quad in \quad \mathsf{goto}\ (pc + \#lc + 1) :: lc :: le :: \underline{\mathsf{ifeq}\ (pc - \#lc - \#le)} \\
\mathcal{S}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) & = & \quad let \quad le = \mathcal{E}(e); lc_1 = \mathcal{S}(c_1); lc_2 = \mathcal{S}(c_2); \\
 & & \quad in \quad le :: \underline{\mathsf{ifeq}\ (pc + \#lc_2 + 2)} :: lc_2 :: \mathsf{goto}\ (pc + \#lc_1 + 1) :: lc_1 \\
\mathcal{S}(e.f := e') & = & \mathcal{E}(e') :: \mathcal{E}(e) :: \underline{\mathsf{putfield}\ f} \\
\mathcal{S}(\mathsf{throw}\ e) & = & \mathcal{E}(e) :: \underline{\mathsf{throw}} \\
\mathcal{S}(\mathsf{try}\ c_1\ \mathsf{catch}\ (X\ x)\ c_2) & = & \quad let \quad lc_1 = \mathcal{S}(c_1); lc_2 = \mathcal{S}(c_2); \\
 & & \quad in \quad lc_1 :: \mathsf{goto}\ (pc + \#lc_2 + 1) :: \underline{\mathsf{store}\ x} :: lc_2 \\
\mathcal{S}(\mathsf{return}\ e) & = & \mathcal{E}(e) :: \mathsf{return}
\end{array}
$$

Figure 8. Compilation function. Primary instructions are underlined.

$$
\begin{array}{rcl}
\mathcal{X}(c_1;\ c_2) & = & \mathcal{X}(c_1) :: \mathcal{X}(c_2) \\
\mathcal{X}(\mathsf{while}\ e\ \mathsf{do}\ c) & = & \mathcal{X}(c) \\
\mathcal{X}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) & = & \mathcal{X}(c_1) :: \mathcal{X}(c_2); \\
\mathcal{X}(\mathsf{try}\ c_1\ \mathsf{catch}\ (X\ y)\ c_2) & = & \quad let \quad lc_1 = \mathcal{S}(c_1); lc_2 = \mathcal{S}(c_2); \\
 & & \quad in \quad \mathcal{X}(c_1) :: \mathcal{X}(c_2) :: \langle 1, \#lc_1 + 1, \#lc_1 + 2 \rangle \\
\mathcal{X}(\_) & = & \epsilon
\end{array}
$$

Figure 9. Definition of exception table.

Definition 6 (compiler for regions) *Suppose* $[c']^{n'}$ *is a branching command in source program* SP*. Then define* $\mathsf{tregion}(n')$ *to be the union, over all* $[c]^n$ *with* $n \in \mathsf{sregion}(n')$ *(in the source language), of* $\{i..j\}$ *where* $TP[i..j]$ *is the compilation of* c*. That is, all program points in the compilation of commands in the (source) region of some branching command with label* n' *are included in the (target) region of the compiled program for the branching instruction* n'*.*

Furthermore

- *if* n' *is a command of the form* if e then c_1 else c_2*, and* $\#lc_2$ *is the length of the compilation of command* c_2*, then* $n' + lc_2 + 1 \in \mathsf{tregion}(n)$ *(note that* $n' + lc_2 + 1$ *corresponds to the* goto *instruction).*
- *if* n' *is an exception-throwing command with* $\mathsf{sHandler}(n') = t$*, then program points* t *and* $t - 1$ *corresponding to* goto *and* store *instructions in the compilation of a try-catch with label* t*, are also included in region of* n'*.*
- *if* n *is a while command, then* $n \in \mathsf{tregion}(n)$*.*

We will use regions for the target language defined as in Definition 6 for proofs of preservation, in next section. The following lemma claims that regions defined as in Definition 6 have the SOAP property.

Lemma 6.1 *Let* $n \in \mathcal{PP}^\sharp$ *be a program point in a target program* P *and let* $\mathsf{tregion}(n)$ *be defined as in Definition 6 from compilation of a command* $[c]^n$*. Then SOAP holds for* $\mathsf{tregion}(n)$*.*

7. Connecting the intermediate and target type systems

The main result is that compilation preserves typing. That is, given a typing derivation for a source program in the high level system, a corresponding security type for bytecode can be obtained. We show that the defined secu-

$$\frac{TP[i] = \mathsf{push}\ n}{i \vdash st, se \Rightarrow se(i) :: st, se}$$

$$\frac{TP[i] = \mathsf{prim}\ op}{i \vdash k_1 :: k_2 :: st, se \Rightarrow k_1 \sqcup k_2 \sqcup se(i) :: st, se}$$

$$\frac{TP[i] = \mathsf{load}\ x}{i \vdash st, se \Rightarrow (\Gamma(x) \sqcup se(i)) :: st, se}$$

$$\frac{TP[i] = \mathsf{store}\ x \quad se(i) \sqcup k \leq \Gamma(x)}{i \vdash k :: st, se \Rightarrow st, se}$$

$$\frac{TP[i] = \mathsf{ifeq}\ j \qquad se = \mathrm{lift}_k(se, \mathsf{tregion}(i))}{i \vdash k :: st, se \Rightarrow \mathrm{lift}_k(st), se}$$

$$\frac{TP[i] = \mathsf{goto}\ j}{i \vdash st, se \Rightarrow st, se} \qquad \frac{TP[i] = \mathsf{return} \qquad se(i) \sqcup k = L}{i \vdash k :: st, se \Rightarrow}$$

$$\frac{\begin{array}{c} TP[i] = \mathsf{putfield}\ f \\ k \leq \mathsf{fieldlevel}(f) \qquad (se(i) \sqcup k') = L \qquad \mathsf{Handler}(i) \uparrow \end{array}}{i \vdash k :: k' :: st, se \Rightarrow st, se}$$

$$\frac{\begin{array}{c} TP[i] = \mathsf{putfield}\ f \qquad k \sqcup se(i) \sqcup k' \leq \mathsf{fieldlevel}(f) \\ \mathsf{Handler}(i) = i' \qquad se = \mathrm{lift}_{k'}(se, \mathsf{tregion}(i)) \end{array}}{i \vdash k :: k' :: st, se \Rightarrow se(i') :: \epsilon, se}$$

$$\frac{\begin{array}{c} TP[i] = \mathsf{putfield}\ f \qquad k \sqcup se(i) \sqcup k' \leq \mathsf{fieldlevel}(f) \\ \mathsf{Handler}(i) \downarrow \qquad se = \mathrm{lift}_{k'}(se, \mathsf{tregion}(i)) \end{array}}{i \vdash k :: k' :: st, se \Rightarrow \mathrm{lift}_{k'}(st), se}$$

$$\frac{TP[i] = \mathsf{getfield}\ f \qquad (se(i) \sqcup k) = L \qquad \mathsf{Handler}(i) \uparrow}{i \vdash k :: st, se \Rightarrow \mathsf{fieldlevel}(f) :: st, se}$$

$$\frac{\begin{array}{c} TP[i] = \mathsf{getfield}\ f \qquad k \leq \mathsf{fieldlevel}(f) \\ \mathsf{Handler}(i) = i' \qquad se = \mathrm{lift}_k(se, \mathsf{tregion}(i)) \end{array}}{i \vdash k :: st, se \Rightarrow se(i') :: \epsilon, se}$$

$$\frac{\begin{array}{c} TP[i] = \mathsf{getfield}\ f \qquad k \leq \mathsf{fieldlevel}(f) \\ \mathsf{Handler}(i) \downarrow \qquad se = \mathrm{lift}_k(se, \mathsf{tregion}(i)) \end{array}}{i \vdash k :: st, se \Rightarrow (\mathsf{fieldlevel}(f) \sqcup se(i)) :: \mathrm{lift}_k(st), se}$$

$$\frac{TP[i] = \mathsf{throw} \qquad (se(i) \sqcup k) = L \qquad \mathsf{Handler}(i) \uparrow}{i \vdash k :: st, se \Rightarrow}$$

$$\frac{TP[i] = \mathsf{throw} \qquad \mathsf{Handler}(i) = i' \qquad k \leq se(i')}{i \vdash k :: st, se \Rightarrow se(i') :: \epsilon, se}$$

Figure 7. Typing rules for instructions in JVM.

rity type satisfies all the conditions for typing of a bytecode program.

First, given a source program SP together with a security environment E for it we obtain a security environment se with which to type the compilation of SP.

Definition 7 (se determined by E) *We define se by induction on syntax of source commands. The domain of se is the set of program points in the compilation $\mathcal{W}(SP)$. Define $se(i)$ as $E(n)$ where $[c]^n$ is the smallest subcommand of SP whose compilation contains program point i.*

Lemma 7.1 *Suppose $D ::\vdash [c]^n : k, k'$ (intro judgement), let E be the security environment derived from D, and let se be determined by E. Then $se = \mathsf{lift}_{k'}(se, \mathsf{tregion}(n))$.*

Now we can show that compilation of a command preserves typing in the intermediate system. The following is proved by induction on the source program.

Lemma 7.2 *Let c be a command in source program SP, typed $\vdash c : E$, and let $[i..j]$ be the program points in compilation of c. Let se be the security environment determined by E. Then, for all st there exists $st_{i+1}, .., st_j$ such that if we define $st_i = st$, we have*

- *$l \vdash st_l, se \Rightarrow st_{l'}, se$, for all $l \mapsto l'$, $l \in \{i..j\}$;*
- *if c is not a throw command, then $st_j = st$;*
- *if $[c]^n$ is a exception-throwing command with $\mathsf{sHandler}(n) = n'$, then $st_{n'} = se(n') :: st$*

Finally, putting together Lemma 7.2 and Theorem 5.3 we obtain the main result.

Theorem 7.3 *Suppose, for given Γ and* fieldlevel*, we have $\vdash SP$, i.e. this source program is typable in the high level system. Then $\vdash \mathcal{W}(SP)$, i.e., its compilation is typable in the target system.*

Example 7.1 *The compilation $\mathcal{S}(c)$ of the program introduced in Example 5.1 and its types obtained with the Target type system is shown below. To construct se for this program, we use E from Example 5.1 and Definition 7 to obtain: $se = \{1 \mapsto H, 2 \mapsto H, 3 \mapsto H, 4 \mapsto H, 5 \mapsto H, 6 \mapsto H, 7 \mapsto H, 8 \mapsto H, 9 \mapsto H, 10 \mapsto L, 11 \mapsto L, 12 \mapsto L\}$.*

1	load y_H	ϵ, se
2	push 0	$H :: \epsilon, se$
3	prim $=$	$H :: H\epsilon, se$
4	ifeq 8	$H :: \epsilon, se$
5	load x_L	ϵ, se
6	store y_H	$H :: \epsilon, se$
7	goto 10	ϵ, se
8	push 1	ϵ, se
9	store y_H	$H :: \epsilon, se$
10	new C	ϵ, se
11	push 3	$L :: \epsilon, se$
12	putfield f_L	$L :: L, se$
		ϵ, se

It is easy to corroborate that the constraints of the type system to apply the transfer rules with the types given above hold. For the ifeq *instruction at program point* 4, $se = \mathsf{lift}_H(se, \mathsf{tregion}(4))$ *can be checked using region* $\{5, 6, 7, 8, 9\}$*, obtained according to Definition 6. Region of* 12 *is empty, since it does not have any successors.*

8. Conclusion

The development and deployment of software that respects an end-to-end confidentiality policy requires a number of ingredients. First, developers must be able to specify how interfaces relate with the policy using labels. They need tools to provide them accurate feedback whether their labeling is consistent with the policy. Second, execution platforms must reflect the assumptions of the policy (e.g. a low attacker is not able to directly read channels labeled high), and must feature security functions that enforce the policy. Verification tools for developers and security functions for execution platforms have a similar purpose, namely to verify that the labeling is correct and ensures the policy. Establishing a formal relation between them, and devising means to exploit the results of source code verification for verification of executables is a significant step towards the adoption of end-to-end security policies in mobile code.

The focus of this paper is on checking that a program is noninterferent with respect to a given labeling—that is, controlling fine-grained flows within a program. The paper follows a long line of work in this area and makes significant progress by showing a formal relation between typability at source code, and bytecode verification for an extended bytecode verifier that enforces noninterference.

In fact we obtained this relation by deriving the typing systems for source code from the bytecode system. First, we establish a correspondence between regions in source code and regions in bytecode. Then we obtain constraints on a security environment for source fragment c, in terms of regions, by combining the constraints from the bytecode rules as applied to the compilation of c. Next, we formulate high level judgements and a connection between them and the security environment. Finally, this connection is used to obtain a high level rule for each source code construct, using the constraints in the construct's intermediate rule and the definition of regions for source code.

We have solved the problem of connecting control dependence between source and target language—even reducing control dependence to an inductive property amenable to machine verification—and we are confident that our approach is robust and can be adapted to more sophisticated settings, including richer languages, more sophisticated policies, and optimizing compilers.

Richer languages. Our language already handles some main complexities of Java, including exceptions and heap allocated mutable objects. The full version of the article also considers methods and multiple exceptions. We expect to treat multiple exceptions in a way similar to JFlow (corresponding to our single exception effect, JFlow uses a list of levels indexed by exception types, called "paths"). We believe that type preservation can be extended to sequential Java, provided existing type systems at source code and bytecode are extended appropriately—in particular, it is possible to treat JVM subroutines, but not so interesting as they will disappear from Java 1.6. The real challenge is to understand whether type preservation scales up to concurrent Java, but for the time being there is no information flow type system that has been proved sound for a concurrent fragment of Java source code or bytecode.

More sophisticated policies. Practical end-to-end policies weaker than full non-interference are often needed, e.g., to cater for downgrading [15] and to connect flow policy with access controls (based on stack inspection or execution history) [3, 5]. Such policies are under active investigation but several current proposals provide source level type systems to express and statically enforce specific policies that enable declassification. We believe that most of these type systems can be adapted to bytecode in such a way that type-preservation will be ensured. For example, it is trivial to extend type preservation to an extension of Java with a cryptographic API where encryption turns secret data into public one.

Practical enforcement mechanisms for end-to-end policies are also likely to combine information flow type systems with other type systems, e.g. for exception analysis, or with logical verification methods. Extending type-preservation results to combinations of type systems is thus an interesting topic for future work, as is preservation of typability/provability in a framework that combines type systems and logical reasoning.

Optimizing compilers. Common Java compilers such as Sun's javac or IBM's jikes only perform very limited optimizations such as constant folding, dead code elimination, and rewriting conditionals whose conditions always evaluate to the same constant. These source-to-source transformations can easily be shown to preserve information-flow typing; thus type preservation lifts to standard Java compilers.

More aggressive optimizations may break type preservation, even though they are semantics preserving, and therefor security preserving. Of course, if the transformations are made after bytecode verification (as in JIT), type preservation is not needed (instead transformations become part of the TCB). For example, applying common subexpression elimination to the program $x_H := n_1 * n_2; y_L := n_1 * n_2$, where n_1 and n_2 are constant values, will result into the program $x_H := n_1 * n_2; y_L := x_H$. Assuming that variable x_H is a high variable and y_L is a low variable, the original program is typable, but the optimized program is not, since the typing rule for assignment will detect an explicit flow $y_L := x_H$. (A naive solution to recover typability is to create a low auxiliary variable z_L in which to store the result of the computation $n_1 * n_2$, and assign z_L to x_H and y_L, i.e. $z_L := n_1 * n_2; x_H := z_L; y_L := z_L$.) Adapting standard program optimizations so that they do not break type preservation is left as future work. We conjecture that most optimizations will only need minor modifications, provided advanced features of the type system such as label polymorphism are available.

Acknowledgments: We thank Martín Abadi, Anindya Banerjee, Andrei Sabelfeld and anonymous reviewers for providing valuable comments.

References

[1] Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[2] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In S. De Capitani di Vimercati, P.F. Syverson, and D. Gollmann, editors, *Proceedings of ESORICS'05*, volume 3679 of *Lecture Notes in Computer Science*, pages 197–221. Springer-Verlag, 2005.

[3] A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15:131–177, March 2005. Special Issue on Language-Based Security.

[4] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of CSFW'02*, pages 253–270. IEEE Computer Society Press, 2002.

[5] A. Banerjee and D.A. Naumann. History-based access control and secure information flow. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 27–48. Springer-Verlag, 2004.

[6] G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In B. Steffen and G. Levi, editors, *Proceedings of VMCAI'04*, volume 2934 of *Lecture Notes in Computer Science*, pages 2–15. Springer-Verlag, 2004.

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

[8] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, 2002. Preliminary version available as INRIA Research report 4254.

[9] D. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.

[10] R. Medel, A. Compagnoni, and E. Bonelli. A typed assembly language for non-interference. In *Ninth Italian Conference on Theoretical Computer Science*, volume 3701 of *Lecture Notes in Computer Science*, pages 360–374, 2005.

[11] A.C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of POPL'99*, pages 228–241. ACM Press, 1999. Ongoing development at `http://www.cs.cornell.edu/jif/`.

[12] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[13] V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In Mooly Sagiv, editor, *Proceedings of ESOP*, pages 77–93, 2005.

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

[15] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings of CSFW'05*, pages 255–269. IEEE Press, 2005.

[16] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer-Verlag, 2001.

[17] Q. Sun, A. Banerjee, and D.A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In R. Giacobazzi, editor, *Proceedings of SAS'04*, volume 3148 of *Lecture Notes in Computer Science*, pages 84–99. Springer-Verlag, 2004.

[18] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4:167–187, December 1996.

[19] D. Yu and N. Islam. A typed assembly language for confidentiality. In *Proceedings of ESOP*, 2006.