From Interfering to Non-interfering Programs

Gilles Barthe
Gilles.Barthe@sophia.inria.fr

Salvador Cavadini
Salvador.Cavadini@sophia.inria.fr

INRIA Sophia Antipolis, France

ABSTRACT

This paper deals with the problem of protecting the confidentiality of data manipulated by sequential programs. We focus on policies guaranteeing confidentiality of information by controlling how information flows during program execution.

There are two established means to enforce information flow policies: static analyses, that are performed at compile time and guarantee that all program executions are free of unauthorized flows; and runtime monitoring, that dynamically detects and neutralizes invalid flows for the current run.

This paper presents a new approach: transform arbitrary, and thus possibly insecure, programs into secure ones, without resorting to runtime monitors. The transformation of programs is performed in two steps: first, programs are analyzed using a constraint-based information flow type system. Then, untypable fragments of the program are eliminated through code rewriting, using a combination of program optimizations and semantics-modifying rules that improve typing. The approach is formalized and illustrated in the context of a sequential fragment of the Java Virtual Machine.

1. INTRODUCTION

Protecting sensitive information—credit card data, personal medical information, *etc*—is becoming an increasingly important issue due to ubiquity of computing systems. Traditionally, confidentiality of information is guaranteed by access control mechanisms, but there is a renewed interest in developing mechanisms that track how information flows during program execution [15].

In an information flow based approach, the confidentiality policy is specified using a lattice of security levels by attaching a security level to each variable in the program. Then, a program is secure, or non-interfering, if every pair of terminating computations, from a pair of initial states differing only in the values of secret variables, lead to final states with identical values of public variables. Thus, non-interference ensures that secret data given as input to the program will not influence the public output of programs.

There are two established means to enforce information flow policies: static analyses, that are often type-based, and runtime monitoring. Static analyses, which are most commonly used to enforce information flow policies, are performed at compile time and guarantee that all program executions are non-interfering. Such static analyses are sound, but rather conservative. For example, most type-based analyses for information flow will reject the following program because the value of *secret* variable is, possibly, printed out by the output statement:

```
1 x := public;
2 if (x=0) then
3     x:=secret
4 else skip fi;
5 output(x)
```

In contrast, runtime monitoring detects and neutralizes invalid flows for the current run of the program during execution. If it deduces, possibly by exploiting information about the execution trace, that an execution is not secure, a runtime monitor may either reject the program or alters the normal behaviour of the program to obtain a secure execution. For example, runtime monitors will only reject or modify executions of the above example when the initial value of *public* is 0, i.e. when command 3 is executed.

Runtime monitoring is more flexible than static verification, since it permits running all programs; of course, the increased flexibility is mitigated by a degradation of runtime performance, and by the fact that runtime monitors modify the semantics of programs. In view of their respective benefits, the question arises whether both approaches can be combined to achieve increased flexibility with a minimal overhead at runtime (without compromising the correctness of the enforcement mechanism). One natural possibility is to only resort to run-time monitors for code fragments that are not accepted by the type system. While such an approach seems feasible, we focus on a much simpler approach, in which we transform possibly insecure programs into secure ones, without resorting to runtime monitors. In a nutshell, the transformation of programs is performed in two steps: first, programs are analyzed using a constraint-based information flow type system (i.e. a type system that produces for all programs, including insecure ones, a system of constraints). Then, untypable fragments of the program (i.e. programs that yield unsatisfiable constraints) are eliminated through code rewriting.

Despite its simplicity, we believe that the method is of broad interest: first, it is applicable to a wide range of programming languages and information flow policies (including policies that consider information release, although such policies are ignored in this paper). Second, it relieves the developer from the burden of programming typable applications: although the transformation may change the behavior of programs, it does so in a predictable way. Third, the transformation does not incur any run-time penalty, as is the case with runtime monitoring techniques.

Contents. Section 2 provides a motivating example of the transformation method. Section 3.1 formalizes the transformation of programs into secure ones, in the context of a sequential fragment of the JVM. The section introduces a constraint-based type system that guarantees non-interference for typable programs. This systems provides the framework for the introduction of the concept of *program improvement*. Then, we provide a set of program transformations that improve program typing. Section 4 discusses how to extend the proposed method to cope with sequential JVM programs. In Section 5 we compare our work with recent related developments. We conclude in Section 6.

2. MOTIVATING EXAMPLE

We motivate the non-semantics preserving part of the transformation by considering a double-blind peer-review process. In such a process, submitting authors are not informed of who reviews their papers and the identity of the authors is not accessible for the reviewers. However, the editor/chair knows who the authors and reviewers are.

Figure 1(a) shows a program that prints out information related with submitted papers to a workshop. The available information is: the workshop name ($wshp$ variable), the submission date (*papDate*), the paper number (*papNum*), title (*papTtl*), authors names ($authors$), referee names ($referee$) and reviewers comments ($refComments$).

Variable *state* indicates if the paper is under reviewing process (state 0), accepted (1), or rejected (-1). If the paper is in accepted or rejected state, the referees comments are printed out. This is controlled through *pc* variable.

The program is written in a simple high-level language with quite standard syntax and semantics except for the command **output**(e) that is used to represent outputs to public visible channels while outputs to secret channels are ignored, consequently, they are not included in program listings.

The scenario of the example can be modeled with an information flow policy defined over a partial order of security levels *public* (P), *authors & referees* (AR), *authors* (A), *referees* (R), and *editor* (E); where $P \sqsubset AR$, $AR \sqsubset A$, $AR \sqsubset R$, $A \sqsubset E$, and $R \sqsubset E$. The policy labels *wshp*, *papTtl*, *papNum*, and *papDate* as P; *state*, *pc*, and $refComments$ as AR level; $authors$ as A; and $referee$ as R.

By transforming the program, we can automatically extract secure versions for different users of the system: authors, referees, and public —Figures 1(b), (c) and (d) respectively— The program version for authors does not print out the $referee$ variable, the sole variable labeled with a security level not equal nor lower than A. Referee's version does not print the $authors$ variable. The version for general public only prints $wshp$, *papDate*, *papNum*, and *papTtl* while all other paper-related data is kept unpublished.

These program versions are *secure slices* of the program, that is, subprograms that can be executed without the risk of producing invalid flows. We use the term secure slice in reference to program slicing [20], a program transformation which deletes commands in programs based on a slicing criterion, that is, a specification of the property that slices must preserve.

3. FORMALIZATION FOR A CORE JVM

This section formalizes more precisely the transformation of programs into secure ones, in the context of a sequential fragment of the Java Virtual Machine (JVM). We adopt as type system a constraint-based variant of the information flow type system of [2] that guarantees non-interference for typable programs.

In this framework, we provide a decidable definition of program improvement (we use the word improvement to carry the intuition that such transformations aim to make more programs typable) and prove that improvements preserve typing. Then, we provide several examples of program optimizations and semantics-modifying rules that improve typing. In addition, we observe that the obvious strategy of applying program optimizations and then semantics-modifying rules always terminate and yield typable programs (strictly speaking, we require that the initial program satisfies some basic properties; these properties are satisfied by all compiled programs that pass bytecode verification [11], and thus are not a strong restriction in practice).

3.1 Constraint based type system

Since our method aims to mend insecure program, it must rely on a type system that collects all potential sources of insecurity in programs, instead of merely rejecting those that may be insecure. For similar reasons, it is preferable that the type system does not require the program to carry any typing information, but rather computes itself the information it needs. In order to fulfill these goals, we define a variant of the type system of [2] that:

- infers for each program point its security environment, which characterizes the security level of the guards under which it executes. As in the type system of [2], we rely on a preliminary analysis that computes for each branching program point its control dependence region, i.e., the set of program points whose execution is controlled by the branching point [6];
- generates for each program P a candidate type S and a candidate security environment se, and a set of constraints $\mathcal{C}$ such that P is typable iff $\mathcal{C}$ is satisfiable.

```
output(wshp);
output(papDate);
output(papNum);
output(papTtl);
output(authors);
output(referee);
pc := false;
if (state = 0) then
 output('reviewing')
else
 if (state = 1) then
  output('accepted');
  pc := true
 else
  if (state = −1) then
   output('rejected')
   pc := true
  else skip fi
 fi
fi;
if (pc = true) then
 output(refComments);
else skip fi
```

(a) Original program

```
output(wshp);
output(papDate);
output(papNum);
output(papTtl);
output(authors);
skip;
pc := false;
if (state = 0) then
 output('reviewing')
else
 if (state = 1) then
  output('accepted');
  pc := true
 else
  if (state = −1) then
   output('rejected')
   pc := true
  else skip fi
 fi
fi;
if (pc = true) then
 output(refComments);
else skip fi
```

(b) Version for authors

```
output(wshp);
output(papDate);
output(papNum);
output(papTtl);
skip;
output(referee);
pc := false;
if (state = 0) then
 output('reviewing')
else
 if (state = 1) then
  output('accepted');
  pc := true
 else
  if (state = −1) then
   output('rejected')
   pc := true
  else skip fi
 fi
fi;
if (pc = true) then
 output(refComments);
else skip fi
```

(c) Version for referees

```
output(wshp);
output(papDate);
output(papNum);
output(papTtl);
skip;
skip;
pc := false;
if (state = 0) then
 skip
else
 if (state = 1) then
  skip;
  pc := true
 else
  if (state = −1) then
   skip
   pc := true
  else skip fi
 fi
fi;
if (pc = true) then
 skip;
else skip fi
```

(d) Version for general public

Figure 1: Example program and three of its secure versions

instr	::=	binop *op*	binary operation on stack
	|	push *n*	push value on top of stack
	|	pop	pop value from top of stack
	|	nop	no operation
	|	load *x*	load value of *x* on stack
	|	store *x*	store top of stack in variable *x*
	|	ifeq *j*	conditional jump
	|	goto *j*	unconditional jump
	|	return	return the top value of the stack
	|	dup	push a copy of the top value of the stack

where $op \in \{+, -, \times, /\}$, $n \in \mathbb{Z}$, $x \in \mathcal{X}$, and $j \in \mathcal{P}$.

Figure 2: Instruction set for JVM$_\mathcal{I}$

A control dependence region structure $(\mathsf{region}, \mathsf{jun})$ satisfies the SOAP (Safe Over APproximation) properties if the following properties hold:

SOAP1 for all program points i and all successors j, k of i ($i \mapsto j$ and $i \mapsto k$) such that $j \neq k$ (i is hence a branching point), $k \in \mathsf{region}(i)$ or $k = \mathsf{jun}(i)$;

SOAP2 for all program points i, j, k, if $j \in \mathsf{region}(i)$ and $j \mapsto k$, then either $k \in \mathsf{region}(i)$ or $k = \mathsf{jun}(i)$;

SOAP3 for all program points i, j, if $j \in \mathsf{region}(i)$ and $j \mapsto \emptyset$ then $\mathsf{jun}(i)$ is undefined.

Figure 3: Safe Over Approximation Properties

$$\frac{P[i] = \mathsf{binop}\ op \qquad n_2\ \underline{op}\ n_1 = n}{\langle i, \mu, n_1 :: n_2 :: os\rangle \leadsto \langle i+1, \mu, n :: os\rangle}$$

$$\frac{P[i] = \mathsf{push}\ n}{\langle i, \mu, os\rangle \leadsto \langle i+1, \mu, n :: os\rangle} \qquad \frac{P[i] = \mathsf{pop}}{\langle i, \mu, n :: os\rangle \leadsto \langle i+1, \mu, os\rangle}$$

$$\frac{P[i] = \mathsf{nop}}{\langle i, \mu, os\rangle \leadsto \langle i+1, \mu, os\rangle} \qquad \frac{P[i] = \mathsf{load}\ x \qquad x \in \mathsf{dom}(\mu)}{\langle i, \mu, os\rangle \leadsto \langle i+1, \mu, \mu(x) :: os\rangle}$$

$$\frac{P[i] = \mathsf{ifeq}\ j \qquad j \in \mathcal{P}}{\langle i, \mu, 0 :: os\rangle \leadsto \langle j, \mu, os\rangle} \qquad \frac{P[i] = \mathsf{ifeq}\ j \qquad n \neq 0}{\langle i, \mu, n :: os\rangle \leadsto \langle i+1, \mu, os\rangle}$$

$$\frac{P[i] = \mathsf{store}\ x \qquad x \in \mathsf{dom}(\mu)}{\langle i, \mu, n :: os\rangle \leadsto \langle i+1, \mu \oplus \{x \mapsto n\}, os\rangle} \qquad \frac{P[i] = \mathsf{goto}\ j}{\langle i, \mu, os\rangle \leadsto \langle j, \mu, os\rangle}$$

$$\frac{P[i] = \mathsf{dup}}{\langle i, \mu, n :: os\rangle \leadsto \langle i+1, \mu, n :: n :: os\rangle} \qquad \frac{P[i] = \mathsf{return}}{\langle i, \mu, n :: os\rangle \leadsto \mu}$$

Figure 4: Operational Semantics for JVM$_\mathcal{I}$

For the purpose of the exposition, it is not possible to consider the type system of [2], which contains more than 60 typing rules, and exploits preliminary analyses to reduce the control flow graph of program. Instead, we base our technical development on the imperative core of the JVM, called JVM$_\mathcal{I}$, and informally discuss extensions to objects, methods and exceptions, in Section 4.

Definition 1 (Program, region).

- *A* JVM$_\mathcal{I}$ *program P consists of a finite sequence of instructions, taken from the instruction set of Figure 2, where $\mathcal{X}$ denotes the set of program variables and $\mathcal{P}$ denotes the set of program points.*
- *The successor relation $\mapsto \subseteq \mathcal{P} \times \mathcal{P}$ of a program P is defined as:*

  - $i \mapsto \{j\}$ *if* $P[i] = \mathsf{goto}\ j$
  - $i \mapsto \{j, i+1\}$ *if* $P[i] = \mathsf{ifeq}\ j$
  - $i \mapsto \emptyset$ *if* $P[i] = \mathsf{return}$
  - $i \mapsto \{i+1\}$ *otherwise*

  and the set of branching points as $B = \{i \in \mathcal{P} | P[i] = \mathsf{ifeq}\ j\}$.
- *A control dependence region structure for program P consists of a function $\mathsf{region} : B \to \wp(\mathcal{P})$ that maps branching points to its control regions, and a (partial) function $\mathsf{jun} : B \to \mathcal{P}$ that maps branching points to its immediate postdominators [6], such that the SOAP axioms —Figure 3— are satisfied.*

The set $\mathsf{State}_\mathcal{I}$ of JVM$_\mathcal{I}$ states is defined as the set of triples $\langle i, \mu, os\rangle$, where $i \in \mathcal{P}$ is the program counter that points to the next instruction to be executed; $\mu \in \mathcal{X} \rightharpoonup \mathbb{Z}$ is a partial function from local variables to values and $os \in \mathbb{Z}^\star$ is the operand stack.

The small-step operational semantics of a program P is given in Figure 4 as a relation $\leadsto \subseteq \mathsf{State}_{\mathcal{I}} \times (\mathsf{State}_{\mathcal{I}} + \mathcal{X} \to \mathcal{V})$, where we use $\underline{op}$ to denote the standard interpretation of operation op over $\mathbb{Z}$, and $\mu \oplus \{x \mapsto v\}$ to denote the unique function μ' such that $\mu'(y) = \mu(y)$ if $y \neq x$ and $\mu'(x) = v$.

We write $\leadsto^\star$ for the transitive closure of $\leadsto$. The evaluation of a program P from an initial memory μ to final memory μ' is defined by $P, \mu \Downarrow \mu' \equiv \langle 1, \mu, \varepsilon \rangle \leadsto^\star \mu'$, where ε represents an empty operand stack.

An information flow policy is given by a lattice $(\mathcal{S}, \sqsubseteq)$ of security levels and a map $\Gamma : \mathcal{X} \to \mathcal{S}$ that assigns a security level to each variable. Non-interference ensures that there is no flow of information from inputs of level k to outputs of level k', unless $k \sqsubseteq k'$.

Definition 2 (public-equal memory). *The public equality between memories μ_1 and μ_2 w.r.t. a security level $k \in \mathcal{S}$ and an information flow policy Γ, noted $\mu_1 \approx_\Gamma^k \mu_2$, is given by:*

$$\forall x \in \mathcal{X}.\ \Gamma(x) \sqsubseteq k \Longrightarrow \mu_1(x) = \mu_2(x) \quad \Longrightarrow \quad \mu_1 \approx_\Gamma^k \mu_2$$

Although different notions of non-interference are applicable to JVM$_{\mathcal{I}}$, we use a termination insensitive one:

Definition 3 (non-interfering JVM$_{\mathcal{I}}$ program). *A JVM$_{\mathcal{I}}$ program P, with an information flow policy Γ, is* non-interfering *at level $k \in \mathcal{S}$ if for every memories μ_1, μ_2, μ_1', μ_2' we have that $P, \mu_1 \Downarrow \mu_1'$, $P, \mu_2 \Downarrow \mu_2'$, and $\mu_1 \approx_\Gamma^k \mu_2$ imply $\mu_1' \approx_\Gamma^k \mu_2'$.*

The information flow type system manipulates stack levels and security environments, as in [2].

Definition 4.

- *The set of stack levels is defined as $\mathbb{S} = \mathcal{S}^\star$. Given $st, st' \in \mathbb{S}$, we define $st' \sqsubseteq st$, iff both have the same number of elements and for each element in st its corresponding element in st' is lower or equal, symbolically:*

$$st' \sqsubseteq st \Longleftrightarrow \begin{cases} |st| = |st'| = n \\ \forall i \in [0, n-1] : st'[i] \sqsubseteq st[i] \end{cases}$$

  where $st[i]$ denotes the i^{th} element of the stack ($st[0]$ is the top of the stack)

- *The set of security environments is defined as $\mathbb{E} = \mathcal{P} \to \mathcal{S}$. Given $se, se' \in \mathbb{E}$, we define $se' \sqsubseteq se$ iff $se'(i) \sqsubseteq se(i)$ for every $i \in \mathcal{P}$.*

In addition, the type system uses constraints to accumulate potential sources of information leaks.

Definition 5 (Constraint).

- *The set $\mathcal{S}^+$ of extended levels is defined by the clause*

$$\mathcal{S}^+ := \Gamma(x) \mid st_i[0]$$

  with $x \in \mathcal{X}$, and $i \in \mathcal{P}$.

- *The interpretation $[\![k_0]\!]_S$ of an extended level k_0 w.r.t. $S \in \mathcal{P} \to \mathbb{S}$ is defined by the clause:*

$$[\![k_0]\!]_S = \begin{cases} \Gamma(x) & \text{if } k_0 = \Gamma(x) \\ \mathsf{head}(S(i)) & \text{if } k_0 = st_i[0] \end{cases}$$

- *A constraint is a pair $k \sqsubseteq k'$ of extended levels. The set of constraint systems is defined as $\mathbb{C} = \mathsf{set}(\mathcal{S}^+ \times \mathcal{S}^+)$. Given $\mathcal{C}, \mathcal{C}' \in \mathbb{C}$, we define $\mathcal{C} \sqsubseteq \mathcal{C}'$ if $\mathcal{C} \subseteq \mathcal{C}'$.*

- *We say that $S \in \mathcal{P} \to \mathbb{S}$ satisfy a constraint set $\mathcal{C}$, written $S \models \mathcal{C}$ iff $[\![k]\!]_S \sqsubseteq [\![k']\!]_S$ for every $k \sqsubseteq k' \in \mathcal{C}$.*

The typing rules, which are given in Figure 5, are of the form

$$\frac{P[i] = instruction}{i \vdash st, se, \mathcal{C} \Rightarrow st', se', \mathcal{C}'}$$

where $st, st' \in \mathbb{S}$, $se, se' \in \mathbb{E}$ and $\mathcal{C}, \mathcal{C}' \in \mathbb{C}$, and where, given $k \in \mathcal{S}$, $se \in \mathbb{E}$ and $X \subseteq \mathcal{P}$, the security environment $\mathrm{lift}_k(se, X)$ is defined by the clause

$$\mathrm{lift}_k(se, X)(i) = \begin{cases} se(i) \sqcup k & \text{if } i \in X \\ se(i) & \text{otherwise} \end{cases}$$

Note that our rules slightly depart from those of [2] in two respects: we require the operand stack to be empty after a branching point (an assumption satisfied by all compiled programs and that allows us to avoid lifting the operand stack and to elude the reference to $se(i)$ in the constraints), and we do not add any constraint for returns (we assume that return instructions do not give any value back). Both changes are for the clarity of presentation.

The constraint-based type system will generate a candidate type and a candidate security environment for all programs such that the stack is of fixed height at each program point (a property that is required by bytecode verification).

Definition 6 (Typing rules, typable programs). *Let $S \in \mathcal{P} \to \mathbb{S}$ and $se \in \mathbb{E}$ and $\mathcal{C} \in \mathbb{C}$.*

- *$(S, se, \mathcal{C})$ is a solution of P, written $(S, se, \mathcal{C}) \in \mathcal{SOL}(P)$, if for every $i, j \in \mathcal{P}$ such that $i \mapsto j$, there exists st' and se' and $\mathcal{C}'$ such that $i \vdash S(i), se, \mathcal{C} \Rightarrow st', se', \mathcal{C}'$ and $st' \sqsubseteq S(j)$ and $se' \sqsubseteq se$, and $\mathcal{C}' \sqsubseteq \mathcal{C}$.*

- *$(S, se, \mathcal{C})$ is a type for P, written $S, se, \mathcal{C} \vdash P$, if both $(S, se, \mathcal{C}) \in \mathcal{SOL}(P)$ and $S \models \mathcal{C}$.*

The soundness of the type system follows from the equivalence between our constraint-based type system, and a standard type system.

Proposition 1. *If P is typable, i.e. $S, se, \mathcal{C} \vdash P$, then P is non-interfering.*

$$\frac{P[i] = \mathsf{binop}\ op}{i \vdash k_1 :: k_2 :: st, se, \mathcal{C} \Rightarrow (k_1 \sqcup k_2 \sqcup se(i)) :: st, se, \mathcal{C}} \quad \frac{P[i] = \mathsf{push}\ n}{i \vdash st, se, \mathcal{C} \Rightarrow se(i) :: st, se, \mathcal{C}} \quad \frac{P[i] = \mathsf{ifeq}\ j}{i \vdash k :: \epsilon, se, \mathcal{C} \Rightarrow \epsilon, \mathrm{lift}_k(se, \mathsf{region}(i)), \mathcal{C}}$$

$$\frac{P[i] = \mathsf{pop}}{i \vdash k :: st, se, \mathcal{C} \Rightarrow st, se, \mathcal{C}} \quad \frac{P[i] = \mathsf{nop}}{i \vdash st, se, \mathcal{C} \Rightarrow st, se, \mathcal{C}} \quad \frac{P[i] = \mathsf{goto}\ j}{i \vdash st, se, \mathcal{C} \Rightarrow st, se, \mathcal{C}} \quad \frac{P[i] = \mathsf{return}}{i \vdash k :: st, se, \mathcal{C} \Rightarrow \mathcal{C}} \quad \frac{P[i] = \mathsf{load}\ x}{i \vdash st, se, \mathcal{C} \Rightarrow (se(i) \sqcup \Gamma(x)) :: st, se, \mathcal{C}}$$

$$\frac{P[i] = \mathsf{store}\ x}{i \vdash k :: st, se, \mathcal{C} \Rightarrow st, se, \mathcal{C} \cup \{st_i[0] \sqsubseteq \Gamma(x)\}} \quad \frac{P[i] = \mathsf{dup}}{i \vdash k :: st, se, \mathcal{C} \Rightarrow st, (se(i) \sqcup k) :: k :: se, \mathcal{C}}$$

Figure 5: CONSTRAINT-BASED TRANSFER RULES

3.2 Program Improvement

This section formalizes improvements as program transformations that preserve typing. Intuitively, improvements are functions that replace program fragments by other program fragments that generate weaker sets of constraints.

DEFINITION 7. *Let P be a program. A program fragment consists of a set $\mathcal{F} \subseteq \mathcal{P}$ of program points, a distinguished program point $i \in \mathcal{F}$, and a set $\mathcal{O} \subseteq \mathcal{F}$ such that:*

- *$\mathcal{F}$ is acyclic, i.e. $\mapsto_{|\mathcal{F}\times\mathcal{F}}$ does not contain any cycle;*
- *i is an entry point, i.e. for every $j \notin \mathcal{F}$ and $k \in \mathcal{F}$, if $j \mapsto k$ then $i = k$;*
- *$\mathcal{O}$ are exit points, i.e. for every $i \in \mathcal{O}$ and $j \in \mathcal{F}$, we have $i \not\mapsto j$.*

Let $(\mathcal{F}, i, \mathcal{O})$ be a program fragment. By composing the transfer rules of the type system, and since $\mathcal{F}$ is acyclic, we can define an $\mathcal{O}$-indexed family of functions $T_o^P : \mathbb{S}\times\mathbb{E}\times\mathbb{C} \rightharpoonup \mathbb{S}\times\mathbb{E}\times\mathbb{C}$.

DEFINITION 8. *A program improvement is a map I from programs to programs such that for every program P,*

- *P and $I(P)$ have the same set of program points and the same control flow graph;*
- *there exists a program fragment $\langle \mathcal{F}, i, \mathcal{O}\rangle$ such that P and $I(P)$ are equal on $\mathcal{P} \setminus \mathcal{F}$, i.e. $P[j] = I(P)[j]$ for every $j \notin \mathcal{F}$, and for every $o \in \mathcal{O}$, $T_o^P \sqsubseteq T_o^{P'}$, where $\sqsubseteq$ is the componentwise extension of the partial orders on $\mathbb{S}$, $\mathbb{E}$ and $\mathbb{C}$.*

We write $P' \preccurlyeq^I P$ if $P' = I(P)$ for some improvement I.

The assumption that P and $I(P)$ have the same control flow graph is made for the sake of simplicity.

PROPOSITION 2. *If P is typable, i.e. $S, se, \mathcal{C} \vdash P$, and P' improves P, i.e. $P' \preccurlyeq^I P$, then P' is typable, and $S, se, \mathcal{C}' \vdash P'$ for some $\mathcal{C}'$.*

3.3 Program Optimizations as Improvements

Not all program optimizations preserve typing: for example, it has been observed in [1] that common subexpression elimination transforms the typable program[1] $h := v_1 + v_2$; $l := v_1 + v_2$ into the untypable program $h := v_1 + v_2$; $l := h$. Likewise, partial dead code elimination transforms the typable program $l := e$; **if** (h) **then** $h := l + e'$ **else skip fi** into the untypable program **if** (h) **then** $l := e$; $h := l + e'$ **else skip fi**. Nevertheless, several common program optimizations improve programs, i.e. that preserve typability of programs.

Figure 6 presents a set of semantics preserving program optimizations rules that can be viewed as program improvements. The rules are generally of the form

$$\frac{P[i] = ins \quad constraints}{P[i] = ins'} \qquad \frac{P[i, i+n] = \vec{ins} \quad constraints}{P[i, i+n] = \vec{ins}'}$$

In the rules, we use $\mathcal{F}$ to denote a stack-preserving sequence of instructions, i.e. a sequence of instructions such that the stack is the same at the beginning and the end of $\mathcal{F}$ execution, which we denote as $\mathcal{F} \in \mathsf{StackPres}$ in the rules. We also assume that there are no jumps from an instruction in $\mathcal{F}$ outside $\mathcal{F}$, so that all executions must flow through the immediate successor of $\mathcal{F}$, and that there are no jumps from an instruction outside $\mathcal{F}$ inside $\mathcal{F}$, so that all executions enter $\mathcal{F}$ through its immediate predecessor. In other words, we assume that $ins :: \mathcal{F} :: ins'$ is a program fragment, where ins and ins' are the instructions preceding and following $\mathcal{F}$. In some rules we use $\mathcal{OS}(i)$ to denote a safe approximation of the operand stack state previous execution of intruction i. We write $\mathcal{OS}(i) = v_1 :: v_2 :: \cdots$ to say that at program point i the top two values on the operand stack are v_1 and v_2. Rule 12 uses $\mathcal{VAL}(x, i)$ to denote the safe approximation of the value of x at program point i. These approximations can be statically computed through, e.g., symbolic analysis [5].

In some cases however, the rules are of the form

$$\frac{P[i, n+m] = \vec{ins} \quad constraints}{P[i, n+m'] = \vec{ins}'}$$

with $m \neq m'$. Therefore such rules do not preserve the number of instructions, and the transformations must recompute the targets of jumps. Besides, such rules do not fit directly

[1]In the examples, we use h and l to name *secret* and *public* variables respectively.

1 – Push/Load - Pop Elimination

$$\frac{P[i,i+n+2] = \mathsf{instr}_i :: \mathcal{F} :: \mathsf{pop} \qquad \mathsf{instr}_i \in \{\mathsf{load}\ x, \mathsf{push}\ n\} \qquad \mathcal{F} \in \mathsf{StackPres}}{P[i,i+n] = \mathcal{F}}$$

2 – Binop - Pop Elimination

$$\frac{P[i,i+n] = \mathsf{binop}\ op :: \mathcal{F} :: \mathsf{pop} \qquad \mathcal{F} \in \mathsf{StackPres}}{P[i,i+n] = \mathsf{pop} :: \mathcal{F} :: \mathsf{pop}}$$

3 – Algebraic simplification I

$$\frac{P[i] = \mathsf{binop}\ \times \qquad \mathcal{OS}(i) = v_1 :: v_2 :: \cdots \qquad v_1 = 0 \vee v_2 = 0}{P[i,i+2] = \mathsf{pop} :: \mathsf{pop} :: \mathsf{push}\ 0}$$

4 – Algebraic simplification II

$$\frac{P[i] = \mathsf{binop}\ op \qquad op \in \{\times, /\} \qquad \mathcal{OS}(i) = v_1 :: v_2 :: \cdots \qquad v_1 = 1}{P[i] = \mathsf{pop}}$$

5 – Algebraic simplification III

$$\frac{P[i] = \mathsf{binop}\ op \qquad op \in \{+, -\} \qquad \mathcal{OS}(i) = v_1 :: v_2 :: \cdots \qquad v_1 = 0}{P[i] = \mathsf{pop}}$$

6 – Dead Store Elimination

$$\frac{P[i] = \mathsf{store}\ x \qquad x \text{ is dead at } P[i]}{P[i] = \mathsf{pop}}$$

7 – Duplicating Load Elimination

$$\frac{P[i,i+n] = \mathsf{load}\ x :: \mathcal{F} :: \mathsf{load}\ x \qquad \mathsf{store}\ x \notin \mathcal{F} \qquad \mathcal{F} \in \mathsf{StackPres}}{P[i,i+n] = \mathsf{load}\ x :: \mathcal{F} :: \mathsf{dup}}$$

8 – Store/Load Elimination

$$\frac{P[i,i+n] = \mathsf{store}\ x :: \mathcal{F} :: \mathsf{load}\ x \qquad \mathsf{store}\ x \notin \mathcal{F} \qquad \mathcal{F} \in \mathsf{StackPres}}{P[i,i+n] = \mathsf{dup} :: \mathsf{store}\ x :: \mathcal{F}}$$

9 – Redundant Store/Load Elimination

$$\frac{P[i,i+2+n] = \mathsf{store}\ x :: \mathsf{load}\ x :: \mathcal{F} :: \mathsf{store}\ x \qquad \mathsf{store}\ x, \mathsf{load}\ x \notin \mathcal{F} \qquad \mathcal{F} \in \mathsf{StackPres}}{P[i,i+n] = \mathcal{F} :: \mathsf{store}\ x}$$

10 – Reduction in Strength

$$\frac{P[i] = \mathsf{binop}\ \times \qquad \mathcal{OS}(i) = v_1 :: v_2 :: \cdots \qquad v_1 = 2}{P[i,i+2] = \mathsf{pop} :: \mathsf{dup} :: \mathsf{binop}\ +}$$

11 – Constant Folding

$$\frac{P[i,i+2] = \mathsf{push}\ c_1 :: \mathsf{push}\ c_2 :: \mathsf{binop}\ op}{P[i] = \mathsf{push}\ c_1\ \underline{op}\ c_2}$$

12 – Load Elimination

$$\frac{P[i] = \mathsf{load}\ x \qquad \mathcal{VAL}(x,i) = n}{P[i] = \mathsf{push}\ n}$$

Figure 6: Optimizing transformation rules

(a)	(b)	(c)	(d)	(e)
1 load h	1 load h	1 load h	1 load h	1 nop
2 push 1	2 push 1	2 push 1	2 push 1	2 nop
3 binop $\times$	3 binop $\times$	3 binop $\times$	3 binop $\times$	3 nop
4 load h	4 load h	4 load h	4 load h	4 nop
5 binop $-$	5 binop $-$	5 binop $-$	5 $\underline{\text{pop}}$	5 nop
6 load l_1	6 load l_1	6 $\underline{\text{nop}}$	6 nop	6 nop
7 binop $\times$	7 $\underline{\text{pop}}$	7 $\underline{\text{nop}}$	7 nop	7 nop
8 store l_1	8 $\underline{\text{pop}}$	8 pop	8 pop	8 nop
	9 $\underline{\text{push 0}}$	9 push 0	9 push 0	9 push 0
	10 store l_1	10 store l_1	10 store l_1	10 store l_1

Figure 7: Example of program optimization sequence

in Definition 7. Although it is possible to extend the definition of program improvement to account for such rules, by relaxing the requirement that the original and transformed program have the same set of program points, we have refrained from doing so since it would clutter the technical development without adding further insight.

The first two optimizations deal with program fragments that put a value on the operand stack to then pop it without using it. We can avoid this useless stack alteration by deleting push-pop, load-pop, and binop-pop pairs. The first rule targets push-pop, and load-pop pairs. The rule constraint reads as: there is a program fragment starting with a load/push instruction that is followed by a, possible empty, program fragment $\mathcal{F}$ that is followed by a pop, and $\mathcal{F}$ leaves the operand stack as it was after the execution of the initial load/push instruction.

If this constraint is satisfied, the initial push or load, and the final pop are removed. The second transformation rule is similar to the first one except that binop is not changed to nop but to pop in order to preserve the original operand stack length. Transformation rule 3 uses $n \times 0 = 0 \times n = 0$ property to optimize a binop $\times$ that operates over a null operand. Rules 4 and 5 use the identity property for $+, -, \times$, and $/$. Transformation rule 6 targets dead stores, this is, store instructions that affect a variable that is not used after the affectation nor before a new affectation. A dead store is useless thus it can be eliminated. Rule 7 targets the cases when before a load x instruction, the value of x is already on top of the stack; if this is the case, the load x instruction can be replaced with a dup instruction to reduce the traffic between memory and stack. Transformation rule 8 targets store x instructions followed by a load x instruction to the same stack position (before any new definition of x). The load x instruction can be deleted and keep the store x preceded by a dup instruction, therefore traffic between variables in memory and the operand stack is reduced (Palsberg et al., in [18], provides a collection of these kind of transformation rules). Transformation rule 9 targets redundant store/load pairs; usually produced by the compilation of command sequences like x:= e;x:=x.... Reduction in strength, rule 10, is a typical peephole optimization. Transformation 11 is a very simple kind of constant folding. If the value n of variable x at program point i is know at compile-time, rule 12 can be used to replace load x at i by push n.

Further optimizations like unreachable code elimination and code motion can be used to improve programs.

Example 1. *Figure 7 shows a sequence of programs resulting from the application of some of the above defined transformation rules. Using symbolic analysis is possible to know that one of the operands used by* binop $\times$ *(instruction 7) is null, thus optimizing transformation rule 3 is applied to obtain the program at Figure 7(b). This program has a* load – pop *sequence (instructions 6 and 7) that can be eliminated by applying rule 1. The resulting program, at Figure*

```
1 load h        1 load h        1 push 0
2 ifeq 6        2 ifeq 6        2 store l
3 push 0        3 push 0        3 load h
4 store l       4 store l       4 ifeq 7
5 goto 8        5 goto 8        5 nop
6 load h        6 push 0        6 goto 8
7 store l       7 store l       7 nop
8 ...           8 ...           8 ...

   (a)             (b)             (c)
```

Figure 8: Optimization-based program improvement

7(c), has a binop − pop *sequence (instructions 5–8) that is transformed by rule 2 to obtain the program at Figure 7(d). Then, using rules for* load/push − pop *and* binop − pop *sequences (rules 1 and 2) it is possible to get the final program at Figure 7(e).*

Notice that the original program is not typable because instruction 8 stores a secret *value into the* public *variable l_1. However, all optimized versions of the program are typable since the instruction* store l_1 *stores a constant value into l_1.*

Example 2. *Figure 8(a) shows a JVM$_\mathcal{I}$ version of the program used in [7] to show the incompleteness of flow-sensitive type systems. Our approach can alleviate this limitation by transforming the program to make it typable. For example, applying optimization rule 12 we get the program at Figure 8(b), still untypable; if then we apply* code motion *we get the typable program at Figure 8(c).*

3.4 Non-Interfering Program Slices

If the program is still interfering after applying semantics preserving improvements, we extract a secure slice from it by deleting interfering instructions. In JVM$_\mathcal{I}$ programs, invalid flows only occur when executing a store x instruction, where x is a public variable, and: the value on top of the operand stack is secret (*direct flow*), the security environment is secret (*indirect flow*).

Thus, one can obtain a non-interfering program by eliminating those dependencies. A naive approach is to replace the store instructions of the above form by pop instructions. While such a replacement is possible and in some cases unavoidable, we are interested in obtaining final programs with a semantics as close as possible to that of the original program. Thus, we use the constraint-based type system to detect which store instructions should be replaced by a pop instruction.

Given a solution $(S, se, \mathcal{C})$ of P, one can find whether a store x instruction at program point i yields an invalid flow by checking $S \not\models st_i[0] \sqsubseteq \Gamma(x)$. This suggests to adopt the following transformation rule:

$$\frac{P[i] = \mathsf{store}\ x \qquad S \not\models st_i[0] \sqsubseteq \Gamma(x)}{P[i] = \mathsf{pop}}$$

Example 3. *The slice at Figure 9(b) is extracted from the interfering program at Figure 9(a) by applying the above transformation to instructions 6 and 8. The slice is typable, thus is a non-interfering program.*

```
                                      1 load h1
1 load h1          1 load h1          2 ifeq 5
2 ifeq 5           2 ifeq 5           3 nop
3 nop              3 nop              4 goto 7
4 goto 7           4 goto 7           5 push 0
5 push 0           5 push 0           6 pop
6 store l1         6 pop              7 load h2
7 load h2          7 load h2          8 pop
8 store l2         8 pop              9 push ⊠
9 ...              9 ...              10 store l2
                                      11 ...

(a) Program        (b) Slice I        (c) Slice II
```

Figure 9: Interfering program and its typable slices

While the repeated application of the above rule yields a secure and typable program, it is also possible to refine the transformation and use a weaker rules for store x instructions that do not yield an indirect flow, i.e. s.t. $se(i) \sqsubseteq \Gamma(x)$, where i is the program point of the instruction considered. For such instructions, one can replace the top of the stack by a default value $\boxtimes$:

$$\frac{P[i] = \mathsf{store}\ x \quad se(i) \sqsubseteq \Gamma(x) \quad S \not\models st_i[0] \sqsubseteq \Gamma(x)}{P[i, i+2] = \mathsf{pop} :: \mathsf{push}\ \boxtimes :: \mathsf{store}\ x}$$

Example 4. *The typable slice at Figure 9(c) is extracted from the interfering program at Figure 9(a) by applying the above transformation rules to instructions 6 and 8.*

3.5 Example of Transformation

Figure 10 illustrates how an insecure program, taken from [10], may be transformed into a secure one. We perform code rewriting at the bytecode level, but we still provide the source code of the initial program and of the (decompilation of) the rewritten program.[2]

The program on the left of the figure is compiled to the bytecode sequence at Figure 10(b). The program is not secure, and thus it is rightly rejected by the information flow verifier of [2], or by its constraint-based variant, described at the beginning of this section. The insecurity is caused by the store instruction at program point 14, which stores a *secret* value into *output* variable (direct flow), and by the store instructions at program points 21 and 23, which are in the control region of the branching instruction at program point 16 that depends on the secret variable h. One obvious way to transform the program into a typable one is to replace the store instructions at program points 14, 21 and 23 by pop instructions. However, it is desirable to minimize the changes in the behavior of the program, and thus it is preferable to perform, in as much as possible, semantics-preserving transformations.

[2]Command **output**(e) is compiled as an instruction sequence that stores e value into *output*, a special public variable that models outputs to user visible channels.

```
1 l0 := l1 + 3;
2 if (l0 = 10) then
3    output (l1);
4    output (h);
5    if (h) then
6       l3 := 0;
7       output (l0);
8    else
9       h := 1;
10   fi
11 else
12   skip
13 fi;
14 output (l0)
```

(a)
Original program

```
1 load l1
2 push 3
3 binop +
4 store l0
5 load l0
6 push 10
7 binop −
8 ifeq 11
9 nop
10 goto 24
11 load l1
12 store output
13 load h
14 store output
15 load h
16 ifeq 20
17 push 1
18 store h
19 goto 24
20 push 0
21 store l3
22 load l0
23 store output
24 load l0
25 store output
```

(b)
Compiled program

```
1 load l1
2 push 3
3 binop +
4 store l0
5 load l0
6 push 10
7 binop −
8 ifeq 11
9 nop
10 goto 24
11 load l1
12 store output
13 load h
14 store output
15 load h
16 ifeq 20
17 push 1
18 store h
19 goto 24
20 nop
21 nop
22 load l0
23 store output
24 load l0
25 store output
```

(c)
Optimized program

```
1 load l1
2 push 3
3 binop +
4 store l0
5 load l0
6 push 10
7 binop −
8 ifeq 11
9 nop
10 goto 24
11 load l1
12 store output
13 push ⊠
14 store output
15 load h
16 ifeq 20
17 push 1
18 store h
19 goto 24
20 nop
21 nop
22 nop
23 nop
24 load l0
25 store output
```

(d)
Bytecode-level typable slice

```
1 l0 := l1 + 3;
2 if (l0 = 10) then
3    output (l1);
4    output (⊠);
5    if (h) then
6       skip;
7       skip;
8    else
9       h := 1;
10   fi
11 else
12   skip
13 fi;
14 output (l0)
```

(e)
High-level typable slice

Figure 10: A source program and its compiled, optimized, and secure versions

For example, store instruction at program point 21 may be replaced by nop instruction by successively applying dead assignment elimination (notice that l_3 is assigned and then not used), that transforms the store instruction by a pop, and a substitution of the basic block push :: pop by two nop instructions (for the simplicity of exposition, we only consider transformations that do not modify the structure of the program). The resulting program is given in Figure 10(c); note that the implicit flow at program point 21 has been eliminated by the transformation. The remaining invalid flows (at program points 14 and 23) cannot be eliminated by semantics-preserving transformations, and they are thus transformed using the two rules defined above. Figure 10(d) provides the resulting program; note that the program is already secure. The sequence load h :: store *output* was replaced by load ⊠ :: store *output*, and load l_0 :: store *output* by load l_0 :: pop and then optimized to nop :: nop. Figure 10(e) shows the transformed program decompiled in the high-level language (we have inserted skip instructions to preserve the structure of the original program to ease comparison).

4. EXTENSION TO A SEQUENTIAL JVM

An important and distinguishing feature of the transformation process presentend in this paper is its scalability to a sequential fragment of the JVM that includes dynamic object creation, instance fields access and modification, program exceptions, and method calls.

First, one can derive a constraint based variant of the type system of [2], as has been shown in this paper for a core JVM. Then, the transformation proposed in this paper extends directly to the extended language. Note however that:

- since the constraint-based type system requires methods must be annotated with security signatures, the transformation will only be applicable to programs that carry such annotations;
- in order to avoid implicit flows caused by high instructions that may raise exceptions, the notion of region is extended and parameterized by an exception type, and the type system rejects programs of the form $h.f := v; l := v'$ where $h.f$ is a reference to the field f of the (high) object h. There are two ways to make the program typable: one can either eliminate the low assignment, since it is in a high region, or eliminate the high assignment, in which case the low assignment is not any longer in a high region. We choose not to restrict the control flow graph of the program by removing high branching statements, but rather stick to the initial method of removing low assignments.

Thus, it is possible to use a constraint-based variant of the type system of [2] in a tool that improves typability of sequential JVM programs. We have not carried such an implementation, although we dispose of a Coq and Caml implementations of an information flow checker for the original type system of [2]. Nevertheless, we have been experimenting the transformation with JBLIF, an implementation of an information flow checker for Java and JVM programs based on the Indus tool [14].

Example 5. *Consider the Java program of Figure 11 Security annotations —from a security lattice Low $\sqsubset$ High— are given as program comments. Parameter* sec *is annotated as High and all other variables are considered Low.*

Lines 13, 14, and 15 declare 3 Low objects: o1*,* o2*, and* o3*. Line 16 assigns the value of* pub *to the field* o1.x*. Line 17 assigns the value of* sec *to the field* o2.x*. Then if* sec *is null and the parameter* a *is "007", a new object is created and assigned to* o3*, and the field* o1.x *is incremented in 1.*

```
1 public class A {
2   int x;
3   void set(){x=0;}
4   void set(int i){x=i;}
5   int get() {return x;}
6 }
7 public class InfFlow {
8   public static void procObj(
9     String[] a,
10    int sec; /* High */ )
11    {
12    int pub = 1;
13    A o1 = new A();
14    A o2 = new A();
15    A o3;
16    o1.set(pub);
17    o2.set(sec)
18    if (sec==0 && a[0].equals("007")){
19      o3 = new A();
20      o1.set(o1.get()+1);
21 } } }
```

Figure 11: Java program with invalid flows

```
7 public class InfFlow {
8   public static void procObj(
9     String[] a,
10    int sec; /* High */ )
11    {
12    int pub = 1;
13    A o1 = new A();
14    A o2 = new A();
15    A o3;
16    o1.set(pub);
17    o2.set()    /* modified */
18    if (sec==0 && a[0].equals("007")){
19      /* o3 = new A(); */
20      /* o1.set(o1.get()+1); */
21 } } }
```

Figure 12: Secure version of the program at Fig. 11

Information flow analysis with the extended type system will show that lines 17, 19, and 20 do not respect the information flow policy: line 17 assigns a secret value to a public field, and the execution of lines 19 and 20 is controlled by a secret variable thus their execution can leak secret information (for example, if the final value of o1.x *is not* pub+1*, then the value of* sec *is not null).*

Applying the transformation rules described in the previous section, interfering commands can be modified to prevent invalid flows. Fig. 12 shows a (detail of the) secure version of the example program; line 17 was modified to assign a default public value to o1.x*, and lines 19 and 20 were deleted.*

5. RELATED WORK

We have presented the first static approach to transform untypable, and thus potentially insecure programs, into typable, and thus secure ones. The method is heavily inspired from recent work on information flow monitoring, which we describe below. Our comparison takes into account three aspects: code size and runtime overhead, soundness, and scalability.

There are two implemented approaches for dynamically tracking information flow: one can track information flows dynamically either by relying on purpose-specific Java virtual machines, see e.g. [3, 13], or by appealing to custom classes [21] or C function libraries [8]. All these approaches are based on a preliminary static instrumentation of the target program. That is, target program is rewritten to augment it with custom code that provides the necessary information and functionality to track information flows at runtime. Contrarily to our method, these approaches produce an increase of the target program code size and execution time. This increase is not negligible: according to the information provided in some of the cited works, it can be in between of 60 and 280% in the case of code size, and between 82 and 1800% of execution time overhead.

Soundness is other important difference of our method w.r.t. these approaches. While our technique guarantees the absence of invalid flows for all program executions, these monitors halt program execution if the leakage of information is imminent; and this halt can produce an information leak. For example, in the following program:

```
1 if (secret=0) then
2    output(1)
3 else skip fi;
4 output(0)
```

the execution of command 2 is insecure because it leaks the value of variable *secret* (if command 2 is executed then the value of *secret* is 0). If after detect the imminent information leak, the monitor aborts program execution it will also leak the value of *secret* because the attacker knows that the monitor only halts if *secret* is 0.

Besides the implemented approaches, there are other theoretical developments to mention.

Shroff et al. [17] propose a runtime information flow control system based on the dynamic tracking of indirect dependencies between program statements. They target a higher-order functional language with mutable state, conditional branching and let-binding. Unfortunately the system is unsound and secret information flowing to public channels may not be detected the first time the flow occurs. As a sound alternative they define an information flow analysis based on statically-computed dependence information. However, this approach mitigates the benefits of runtime monitoring, since it looses precision.

In [10], Le Guernic et al. describe a runtime monitoring system for a simple sequential while language. This monitor uses an automaton to track information flows and alters the program behavior (instead of interrupting execution) to keep the program execution safe. More recently, Le Guernic [9] extends the automaton to cope with concurrent programs.

While our method is inspired from runtime monitoring, its realization builds upon information flow type systems, which has its roots in the work of Denning and Denning work [4], and more recently in the work of Volpano and Smith [19]. To date, the use of non standard type systems to enforce confidentiality remains dominant in the field, see [15] for a survey of the field. For example, Jif [12] provides a prominent example of information flow typed extension of Java that builds upon the *decentralized label model* and supports flexible and expressive information flow policies.

Although our method, as presented here, only addresses the enforcement of non-interference, many realistic systems need to declassify some kind of confidential information as part of their normal behavior; the actual challenge is to differentiate between proper and improper declassification of confidential information. Sabelfeld and Sands [16] provide a survey of current research on information flow policies and enforcement mechanisms in presence of declassification.

6. CONCLUSION

We have defined a general method to transform a wide class of insecure programs into typable, and hence secure programs, using a combination of semantics-preserving and type-improving transformations and of semantics-modifying transformations that remove insecure assignments.

Currently, our method is based on the flow insensitive type system of [2]. In order to extend the applicability of our method, we would like to rely on a more flexible type system where local variables and fields are allowed to change security levels during execution. Since there are currently no such flow sensitive type system for the JVM, our first task is to design such a type system, and prove its soundness; one attractive solution would be to adapt some of the ideas of Hunt and Sands [7] to the JVM. Furthermore, the adoption of a flow sensitive type system raises a number of interesting questions w.r.t. our method. Firstly, there might be more than one way to remove assignments to make programs secure, as witnessed by the following program:

$$l{:=}\ h;\ \textbf{if}\ (l)\ \textbf{then}\ l'{:=}\ 0\ \textbf{else skip fi}$$

Rather than operating non-deterministic transformations on the program (or making them deterministic by removing all assignments!), it seems preferable to deduce from the constraint set generated by the type system which fragments of the program will never leak information, which will always leak information, and which may leak information, and statically transform program fragments of the second kind and dynamically monitors program fragments of the third kind.

A much easier goal is to extend the method to type systems that support some form of controlled information release. We believe that the information flow type system of [2] can be extended to allow declassification along the "what" and "where" dimensions for the sequential JVM. Such an extension will be useful to, for example, handle a more complex version of the peer review process presented in Section 2.

Finally, we believe that the method proposed in this paper can also be applied usefully in the context of type systems for resource control.

7. REFERENCES

[1] G. Barthe, D. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for Java. In *Symposium on Security and Privacy*. IEEE Press, 2006.

[2] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. In *Proc. of 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, 2007.

[3] D. Chandra and M. Franz. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. to appear in *23rd Annual Computer Security Applications Conference (ACSAC 2007)*, Miami Beach, Florida; December 2007.

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

[5] T. Fahringer and B. Scholz. *Advanced Symbolic Analysis for Compilers.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[7] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 79–90, New York, NY, USA, 2006. ACM Press.

[8] L. C. Lam and T. C. Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 463–472. IEEE Computer Society, 2006.

[9] G. Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, pages 218–232, 2007.

[10] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based confidentiality monitoring. In *Proceedings of the Annual Asian Computing Science Conference*, 2006.

[11] T. Lindholm and F. Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[12] A. C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM Press.

[13] S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. In *First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007)*, pages 1–11, Dresden, Germany, 2007.

[14] V. Ranganath and J. Hatcliff. Slicing concurrent Java programs using Indus and Kaveri. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5):489–504, October 2007.

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

[16] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW '05: Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 255–269, Washington, DC, USA, 2005. IEEE Computer Society.

[17] P. Shroff, S. Smith, and M. Thober. Dynamic Dependency Monitoring to Secure Information Flow.

In *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 203–217, Washington, DC, USA, 2007. IEEE Computer Society.

[18] T. Vandrunen, A. Hosking, and J. Palsberg. Reducing loads and stores in stack architectures, 2000.

[19] D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 607–621, London, UK, 1997. Springer-Verlag.

[20] M. D. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[21] S. Yoshihama, T. Yoshizawa, Y. Watanabe, M. Kudo, and K. Oyanagi. Dynamic information flow control architecture for web applications. In J. Biskup and J. Lopez, editors, *Computer Security ESORICS 2007*, volume 4734 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2007.