# Timing Aware Information Flow Security for a JavaCard-like Bytecode

Daniel Hedin [1] David Sands [2]

*Department of Computing Science*
*Chalmers*
*Goteborg, Sweden*

**Abstract**

Common protection mechanisms fail to provide *end-to-end* security; programs with legitimate access to secret information are not prevented from leaking this to the world. Information-flow aware analyses track the flow of information through the program to prevent such leakages, but often ignore information flows through *covert channels* even though they pose a serious threat. A typical covert channel is to use the timing of certain events to carry information. We present a timing-aware information-flow type system for a low-level language similar to a non-trivial subset of a sequential Java bytecode. The type system is parameterized over the time model of the instructions of the language and over the algorithm enforcing low-observational equivalence, used in the prevention of implicit and timing flows.

*Key words:* covert channels, information flow, security, bytecode

## 1 Introduction

With the increasing adoption of mobile devices with extensible functionality, security becomes more and more important. Common protection mechanisms of today include *sandboxes*, which protect the system by running programs with limited capabilities, and *application firewalls*, which prevent two applications from accessing each others private data or code. The safety of such systems – in particular those targeting some variant of the Java virtual machine – often rely on type systems or other means of static verification that prevent the application from breaking the abstraction.

Common to these techniques is that they fail to provide what is known as *end-to-end security*. Nothing prevents a program with legal access to a secret to tell this secret to the world, by accident or malice. Recent research

[1] `http://www.cs.chalmers.se/~utter`
[2] `http://www.cs.chalmers.se/~dave`

has focused on the problem of statically determining when programs satisfy end-to-end secrecy properties [14].

The ways by which programs can disclose information are often categorised into *explicit* and *implicit* flows, together with flows through other *covert channels*. Explicit flows amounts to directly leaking a secret using some public communication channel. Implicit flows – a form of covert channel – may arise when the program control flow depends on secrets. If the values of secrets can effect the execution path through a program, then modifications to the environment and public output in such a path may be used to indirectly encode properties of the secrets.

Explicit flows are easily prevented using ordinary protection mechanisms, while implicit channels need some kind of information flow aware mechanism. A covert channel is the general name given to a medium not intended for communication which is used to transfer information [10]. The timing of certain observable events can be used as a covert channel, and this is considered in this paper. Covert channels are a delicate problem which can be dealt with using techniques similar to implicit channels, but tend to lead to very restrictive systems. Thus, they are often deliberately ignored in information flow security in order to appear more "practical". Nevertheless, covert channels pose a real and serious threat and cannot be ignored just for convenience. As an example, suppose that an attacker can only reliably observe variations in the timing of observable events when the difference is of the order of 0.5 seconds. Even with such a low bandwidth (corresponding to an attacker observing a system across a busy network) it is quite plausible to leak the equivalent of VISA card number in well under 2 minutes.

The protection mechanisms described in the majority of the research described in [14] are similar in the respect that they are all automatic. The untrusted code is statically verified prior to execution or monitored during execution without calling for user interaction. Indeed, it may be unreasonable to assume any user interaction when establishing security properties of dynamically downloaded code. However, any automated security checking that does not model timing will potentially accept a program which leaks information through timing, no matter how blatant the leak is. The program will be allowed to run and, even worse, the verification process will establish trust between the user and the program.

In this paper we explore the problem of end-to-end information-flow security for a simple bytecode language in which we model and prevent certain classes of timing attacks.

Analysis of lower level languages is difficult, but there are several potential benefits. Higher-level programs makes the compiler part of the trusted computing base. Development, implementation and maintenance of a trustworthy compiler is a huge undertaking. From the point of view of timing properties, low-level languages are much more detailed, and thus allow finer-grained timing models. This is often impossible to perform at the level of the source code, as many aspects of the timing behaviour determined by the compilation process.

The remainder of the paper is organised as follows. Section 2 considers related work, Section 3 introduces the language and its semantics, Section 4 discusses timing, time models and non-interference, Section 5 explores the more interesting parts of the type system and Section 6 concludes.

## 2 Related Work

Our work builds on ideas from type-based program analysis. Among the more complete treatments of type system based information flow security we find FlowCaml [15,13] and JIF[12], handling large subsets of OCaml and Java respectively.

Regarding type systems for low-level languages, Stata and Abadi[16] proposed a type system for a subset of Java bytecode focused on the guarantees provided by regular bytecode verification. For assembly languages, notable work has been done by the TAL[3] group at Cornell. With respect to Java bytecode the work by Morisett et al. in Stack-Based Typed Assembly Language (STAL) [11] is the most relevant.

Only recently has the problem of information-flow security been studied for low level languages. Barthe and Rezk[4,5] provide a flow sensitive type system for a sequential bytecode language. Their basic type structure is similar to that of Stata and Abadi's, extended to a *polyvariant* analysis. As is standard for most analyses (deriving from the approach originally outlined by Denning [6]), implicit flows are prohibited by forbidding modifications of parts of the environment with lower security type than the current context. For example, if the current program counter value depends on the value of a secret, then the code must not write down to a lower security level. A noninterference theorem is established for the type system.

With an emphasis on handling a full-scale language Genaim and Spoto[7] present a compositional information flow analysis for full Java bytecode. The analysis uses boolean functions to encode information flow.

At the other end of the scale, Kobayashi and Shirane[8] provide noninterference proofs for an analysis of information flow for a tiny subset of Java bytecode, excluding arrays, objects and methods. The main focus of their work is an extension to the type system of Stata and Abadi with information flow awareness. Unlike the above works there is also a brief consideration of timing channels. Kobayashi and Shirane suggest a method for dealing with timing derived from their main theorem, which says that if a program terminates with a public value in $m$ steps for one environment then it will terminate for any low-equivalent environment within $c * m$ steps for some constant $c$. To handle timing, Kobayashi and Shirane suggest that given this constant $c$ you measure the number of steps of execution when all secret values are set to 0,

[3] http://www.cs.cornell.edu/talc

and insert a delay before returning so that return always takes place after (c + 1)n steps. Their approach assumes both the possibility of measuring the execution time of statements in the program as well as delaying results a specified time. In many circumstances (e.g. multi-function smartcards) neither is possible.

With respect to timing aware systems Agat[2] presents a timing aware type system for a small While-language which includes a transformation which takes a program and transforms it into an equivalent program without timing leaks. In [1][Paper II] Agat implements and evaluates the transformation for a tiny subset of Java bytecode. The analysis is only informally specified, but is enough to test that certain timing leaks can indeed be eliminated, based on observations of a particular bytecode interpreter.

Our approach picks up from Agat's preliminary experiment, and seeks to extend it to a larger subset of Java bytecode, and to formalise it in a way that allows us to formulate (and ultimately, we hope, *prove*) its correctness specification.

The main contribution of this paper is a type system, which is parameterised over a timing and model of the instructions, allowing for various degrees of timing sensitivity to be handled, including simple forms of cache-sensitive timing.

## 3 Syntax and Semantics

The language is equivalent to a sequential subset of Java bytecode, including objects and arrays, but excluding exception handling and the **jsr/ret** instruction pair.

### *3.1 Syntax*

A program, ranged over by $P$, is a collection of classes. A class, ranged over by $C$, is zero or more fields and methods. A field, ranged over by $F$, is defined by its type and its identifier. A method, ranged over by $M$, is defined by its identifier, type and a list of instructions. Class, method and field identifiers are ranged over by $\mathcal{C}$, $\mathcal{M}$ and $\mathcal{F}$ respectively. The syntax contains references to the type language: $\tau$, $\mu$ are *security types* defined in Section 5.1. Let $l$ range over program labels.

$$\begin{array}{lll}
\textit{Programs} & P & ::= C_1 \ldots C_n \\
\textit{Classes} & C & ::= \textbf{class}\ \textit{cname}[\textbf{extends}\ \textit{scname}]\{F_1; \ldots F_n; M_1 \ldots M_n\} \\
\textit{Fields} & F & ::= \tau\ \textit{vname}; \\
\textit{Methods} & M & ::= \textit{mname}\ \mu\ \{[l_1]\ \textit{inst}_1 \ldots [l_n]\ \textit{inst}_n\}
\end{array}$$

Fig. 1. Syntax

The set of instructions, $\mathcal{I}$, ranged over by *inst*, is defined in figure 2.

$\mathbf{pop}, \mathbf{dup}, \mathbf{swap}, \mathbf{const}_{\mathbb{Z}}, \mathbf{load}_{\mathbb{N}}, \mathbf{store}_{\mathbb{N}}, \mathbf{add}, \mathbf{sub}, \mathbf{mul}, \mathbf{div}, \mathbf{rem}, \mathbf{if}@_{l}\, l,$
$\mathbf{goto}_{l}, \mathbf{new}_{\mathcal{C}}, \mathbf{getfield}_{\mathcal{F}}, \mathbf{putfield}_{\mathcal{F}}, \mathbf{invoke}_{\mathcal{M}}\ \tau, \mathbf{return}, \mathbf{output}_{H|L}, \mathbf{cast}_{\mathcal{C}},$
$\mathbf{anew}_{\tau}, \mathbf{aload}, \mathbf{astore}, \mathbf{length}$

where $@ \in \{eq, ne, le, ge, lt, gt\}$.

Fig. 2. Instructions

### *3.2 Semantics*

The semantics is an ordinary small-step operational semantics on configurations for a given program, with the addition of two explicit termination transitions corresponding to normal and abnormal termination. Transitions originating from instructions with publicly observable behaviour are annotated with that observable behaviour. We assume that the original program has been relabeled, so that each instruction has a unique label in the program. Let $PC$ be the set of instruction labels ranged over by $pc$ and $l$, let $p$ range over object pointers, $Ptr$, and let $ap$ range over and array pointers, $APtr$.

| | |
|---|---|
| *Primitive Values* | $v_p ::= \mathbb{Z} \mid p \mid ap$ |
| *Objects* | $o ::= \langle \mathcal{C}, \mathcal{F}_1 : v_{p1}, \ldots, \mathcal{F}_n : v_{p_n} \rangle$ |
| *Arrays* | $a ::= \langle \tau, v_{p1}, \ldots, v_{p_n} \rangle$ |
| *Heaps* | $oe ::= (p_1 \mapsto o_1, \ldots, ap_1 \mapsto a_1, \ldots)$ |
| *Environments* | $e ::= \langle st, r, oe, fs \rangle$ |
| *Operand Stacks* | $st ::= \epsilon \mid v_p \cdot st$ |
| *Register Banks* | $r ::= (v_{p_0}, \ldots, v_{p_{255}})$ |
| *Frame Stacks* | $fs ::= \epsilon \mid f \cdot fs$ |
| *Frames* | $f ::= (st, r, pc)$ |

Fig. 3. Values and environments

As given in Figure 3, the primitive values, ranged over by $v_p$, are integers, object pointers and array pointers. Objects and arrays are maps from field identifiers or natural numbers respectively to primitive values. Objects and arrays contain their security types, defined in Section 5.1. The heap is a map from pointers to heap entities, arrays and objects. The execution environment, ranged over by $e$, is a 4-tuple of the operand stack, the registers, the heap and the framestack. The operand stack, ranged over by $st$, is a stack of primitive values. A register bank, ranged over by $r$, is a map from register numbers to primitive values. The frame stack, ranged over by $fs$, is a stack of frames. A frame, ranged over by $f$, is composed of the saved operand stack, registers and the program counter. In the remainder of this text we refer to registers, stack locations and the fields of heap entities collectively as *environment locations*.

Let $O$ be the set of observable behaviours and let $\hat{o}$ denote the presence of a behaviour $o \in O$ or no observable behaviour. The semantics is formulated w.r.t. a given program, $P$, with rules of the form:

$$\frac{\ldots}{(pc_i, e_i) \xrightarrow{\hat{o}} (pc_e, e_e)} \qquad \frac{\ldots}{(pc_i, e_i) \rightarrow e_e} \qquad \frac{\ldots}{(pc_i, e_i) \downarrow}$$

representing one step of execution with possible observable behaviour, normal and abnormal termination from left to right. A program may be started in any static method taking an array of integers as parameter.

## 4 Timing and Non-interference

*Non-interference* is one of the most used concepts in information-flow security. The general idea of non-interference is that variations in secret values available to the program should not produce variations in the publicly observable behaviour. Roughly, the program environment is categorised into secrets (of *security level* $H$) and non-secrets (of security level $L$). Non-interference states that if you start the program in any two environments in which the non-secret parts are equal, then, when the program terminates, the non-secret parts of the resulting environments must also be equal. If this is the case, then the secret parts of the environment cannot interfere with the non-secret parts and thus no information about the secrets can be deduced from the non-secrets. The judgement that two environments have equal non-secret parts is often referred to as *low-equivalence*(Section 5.3), commonly formalised in terms of equivalence relations w.r.t. to an environment *security type*(Section 5.1) describing which parts of the environments are considered secret and which are not.

The success of non-interference comes from its generality, which allows instantiations to cover a wide array of information flows. Any behaviour which can be semantically modelled falls into the scope of non-interference – e.g. explicit and implicit flows, but also timing, cache aware timing and other covert channels.

The downside with non-interference is the strictness of the formulation. Leaks are not quantified only qualified, which makes the notion unusable for programs that naturally must contain leaks. This is beyond the scope of the present study however. We refer the reader to [14] for a survey of language based information-flow security.

To handle timing we need a time model, i.e. a model that describes the timing behaviour of the instructions of the language. For an assembly language, the timing of an instruction depends amongst other things on the computer architecture: the CPU and its surroundings. For an interpreted bytecode the situation is more complex. Not only does the underlying hardware affect the timing model, but also the implementation of the runtime environment. The timing model is very important; the security of the system is directly linked to the quality of the model. There are a number of possibilities:

**Step-Counting Model** The most simple possible model assigns the same unit time to all instructions. This is equivalent to counting the number of transitions in an operational semantics and is the type of timing model presented in e.g. [8].

**Constant Time Model** Obviously, not all instructions take the same time. One refinement is to assign different constant times to the different instructions. This is the model found in [1].

**Functional Time Model** The next refinement is to let the time model be a function of the parameters to the instruction. Instructions like *anew*, that creates an array of size $n$, are not constant time. In [2], this is handled by disallowing such instructions when they have secret values as arguments.

**History Sensitive Time** Unfortunately, the timing of certain instructions is not a function solely of the current state upon which the instruction operates. For instructions referencing memory, the state of the cache is an important parameter with a potentially dramatic effect on the time taken to execute an instruction.

Our formulation of time models covers these four cases.

## *4.1 Histories and Time Models*

To allow for different models we extend the standard semantics with histories, accumulating information about the execution. The histories are the base for the calculation of an abstract time, which is used to ensure that the time of publically observable is independent on secret values.

For a given program, a time model is a pair, $(SM, TM)$, of the semantic time model, $SM$, and the corresponding type time model, $TM$. The way to view this is that $SM$ provides the semantics for the time and $TM$ provides the type judgements, used in the type rules. The connection between $TM$ and $SM$ is a non-interference style demand, similar to the one for traces below. For example, if the time [4] of an instruction is depending on the first but not the second of its parameters, then the security type of the time of that instruction is affected by the security type of the first parameter but not the second.

A history is typically built from a number of sub-histories, e.g. the current time and the state of the data cache, related to each other by the instructions, but still relatively independent. [5] Because of this we define a history to be an ordered product of sub-histories, imposing the same structure on the security type of histories. This allows for an adequate separation of (partially) independent sub-histories.

### *4.1.1 Semantic Time Model*

Let $E$ be the set of environments, ranged over by $e$. The semantic time model is a quadruple $\langle H, T, hist, time\rangle$ where

- $H$ is the set representing execution histories, defined structurally in terms of an ordered product of sub-histories, each with a distinguished empty element and equality operation,

[4] According to the semantic time model

[5] It is possible to imagine situations where the state of the data cache reflects secret information and the time doesn't (the execution time of only a few instructions is depending on the state of the data cache) or the other way around (there are more things than the data cache that affect the execution time of instructions).

- $T$ is the abstract domain of times, equipped with an equality operation,
- $\mathit{hist} : \mathcal{I} \times E \times H \to H$ is a partial function which takes an instruction, the current state and the history so far, and produces the extended history corresponding to the next state of the computation.
- $\mathit{time} : \mathcal{I} \times E \times H \to T$ gives the observable time after executing the next step in the history $H$.

### 4.1.2 Type Interface

Let $ET$ be the set of environment types (Section 5.1). The interface to the type system is a triple $\langle HT, \phi, \psi\rangle$ where

- $HT$ is the set of history security types defined by the structure of $H$ by assigning a security level to each sub-history,
- $\phi : \mathcal{I} \times ET \times HT \to HT$, is a type function consistent with $\mathit{hist}$ as defined below,
- $\psi : \mathcal{I} \times ET \times HT \to \sigma$, is a type function extracting the type of time consistent with $\mathit{time}$ as defined below.

For example, let $\overline{\sigma}$ be the set of all security levels ranged over by $\sigma$ (Section 5.1). For a step counting approach, $H = T = \mathbb{N}$, $\mathit{hist} = \lambda(i, e, h).h + 1$, and $\mathit{time} = \lambda(i, e, h).h + 1$. As type interface we would get $HT = \overline{\sigma}$ and $\phi = \lambda(i, e_t, h_t).h_t$ and $\psi = \lambda(i, e_t, h_t).h_t$

To model simple (data-) cache behaviour we can take H to be a pair of the current time and the sequence of memory accesses performed so far, together with an arbitrary time function, depending only on the current instruction and the history. The type interface would be $HT = \overline{\sigma} \times \overline{\sigma}$ and $\phi$ and $\psi$ would be type functions corresponding to the time and history function in the way defined above.

### 4.1.3 Augmented Operational Semantics

The operational semantics is augmented to maintain the current history, which in turn enables the time to be computed and offered as part of the observable output. The extension of the standard semantics with time is specified as follows. For observable[6] transitions $(pc, e) \xrightarrow{o} (pc', e')$ we define

$$(pc, e, h) \xrightarrow{o,\mathit{time}(pc,e,h)} (pc', e', \mathit{hist}(pc, e, h))$$

Furthermore, for any vector of augmented observations $\widetilde{o,t}$, and any set of program points $P$, we define an auxiliary transition relation, $\overset{\widetilde{o,t}}{\rightsquigarrow}_P$, which describes sequences of observable steps which do not go beyond program points in $P$. More precisely, if $\widetilde{o,t} = (o_1, t_1), \ldots, (o_n, t_n)$, then

$$(pc_0, e_0, h_0) \overset{\widetilde{o,t}}{\rightsquigarrow}_P (pc_n, e_n, h_n)$$

[6] The extension for non-observable transitions is identical

if and only if

$$(pc_0, e_0, h_0) \rightarrow^* \overset{o_1,t_1}{\rightsquigarrow}_P \cdots \rightarrow^* \overset{o_n,t_n}{\rightsquigarrow}_P (pc_n, e_n, h_n)$$

where at most the final program $pc_n$ may be one of program points in $P$. The relation is used in the formulation of Low-observable equivalence, defined in Section 5.3 below.

### *4.2 Top-level Non-interference*

In this section we briefly state the semantic criterion which our type system aims to verify. To describe whether an attacker (an observer of the low events of the system) can learn anything about the high inputs to the system we need to be precise about what the attacker can observe. We assume that the only observable outputs are the values produced by the output instruction, together with the time at which the output was made. Implicit in the formulation lies the assumption that termination is not directly observable, nor the cause of the termination. To see this, consider three programs all free from public output of which the first terminates normally, the second crashes and the third diverges. Under the assumption that only the public output is observable the three programs are equivalent. This model is consistent with an attacker who observes the running of the system only through its outputs, and cannot see termination, normal or otherwise. Ignoring everything other than the directly observable output could, in some circumstances, allow up to one bit of information to be leaked per run. Ignoring timing leaks altogether, on the other hand, is a more serious matter as we argued in Section 1.

The assumption that the termination is not observable lets us handle partial instructions and loops on secret data in a more liberal way than would otherwise be possible (c.f. [2,17]) However, once we have looped on high data then no further low output can be allowed. In this sense we say that our security criterion is *weakly termination sensitive.*

Now we can set out our formulation of non-interference. We suppose that a method $M$ takes an array as parameter. The array is of public length, but contains secret contents. We say that $M$ is non-interfering if there are no variations in the public output or in the time at which output occurs for any pair of initial environments $e_1$ and $e_2$ which differ only on the values stored in the secret array:

$$\forall \widetilde{o}, \widetilde{t_1}, \widetilde{t_2}.(M_{init}, e_1, \epsilon) \overset{\widetilde{o,t_1}}{\rightsquigarrow} \iff (M_{init}, e_2, \epsilon) \overset{\widetilde{o,t_2}}{\rightsquigarrow} \wedge \widetilde{t_1} =_t \widetilde{t_2},$$

where $=_t$ is the equality operation on the abstract domain of times from the time model extended pointwise to vectors, and $\widetilde{o}$,$\widetilde{t_1}$, $\widetilde{t_2}$ are vectors of output and time respectively.

## 5 Type System

We present a compositional, timing aware information flow type system with the method as the unit of composition. The structure of the type system is

| | | |
|---|---|---|
| *Security Levels* | $\sigma$ | $::= L \mid H \mid \sqcup(\alpha_1, \ldots, \alpha_n)$ |
| *Prim. Security Types* | $\tau$ | $::= aptr_{\sigma,\sigma}\ \tau \mid int_\sigma \mid ptr_{\sigma,\sigma}\ \mathcal{C}$ |
| *Method Quant. Context* | $\Delta_m$ | $::= \cdot \mid \alpha, \Delta_m$ |
| *BB Quant. Context* | $\Delta_\varepsilon$ | $::= \cdot \mid \gamma$ |
| *Method Types* | $\mu$ | $::= \forall[\Delta_m].\widetilde{\tau}, \sigma, \xi \xrightarrow{\sigma} \tau, \sigma, \xi$ |
| *Operand Stack Types* | $s$ | $::= \epsilon \mid \gamma \mid \tau \cdot s$ |
| *Register Bank Types* | $r$ | $::= (\tau_0, \ldots, \tau_{255})$ |
| *BB Types* | $\varepsilon$ | $::= \forall[\Delta_\varepsilon].s, r, \sigma, \xi$ |

Fig. 4. Type Language

similar in spirit to that of STAL[11], with statically typed labels and polymorphic stack types, but less expressive. We don't model the original type system of bytecode, rather, we deviate from it in certain respects. However, we assume that all programs under consideration have been subjected to byte code verification and, thus, that certain invariants hold during program execution.

For simplicity we have excluded static fields, interfaces and exception handling. The two former would be a trivial extension. JIF shows us a way to handle exceptions (of which the solution in [5] can be seen as a special form) that probably can be adapted to our setting, although timing makes it a delicate matter.

Due to space constraints we can only present a small selection of the type system. For more information we refer the reader to the home page[7] of this paper, where the entirety of the type system and prototype implementations will be available.

Common to all information-flow aware type systems is that they must track implicit flows. As mentioned earlier implicit flows arise when the control flow is depending on secrets; i.e. from the conditional branches. To allow for a smooth formulation of the type system bytecode programs must be preprocessed to regain some structural information, e.g. the (least) merge point of branch instructions. Informally, the merge point is the first program point passed through by all traces starting at the targets of the instruction. For a given control flow graph, this is an easy graph problem. The lack of dynamic branches, i.e. branches where the possible targets are not statically known, makes the (inter-method) control flow statically decidable. See [7] for a more thorough discussion on this topic.

## 5.1 Type Language

Defined in Figure 4, the formal security levels, ranged over by $\sigma$, consists of the two actual security levels $H$(high or secret) and $L$(low or public), and a formal least upper bound of security variables $\sqcup(\alpha_1, \ldots, \alpha_n)$. The security levels form a lattice with $L$ as the least element, $H$ as the top element and the formal joins in between, ordered by set inclusion of the variables. We write $\alpha$

[7] http://www.cs.chalmers.se/~utter/bytecode-time/

for $\sqcup(\alpha)$.

The primitive security types, ranged over by $\tau$, are built from types of integers, object pointers and array pointers annotated with security levels. The security level on integers is associated with the value of the integer. Pointers have two security levels: the first is associated with the actual value of the pointer and the second is associated with the structure of the pointed object. The structure of arrays is the size and the structure of objects is the (principal) class. This differentiation allows for a better handling of pointer dereferencing instructions. The pointer types also carry the security types of the pointed object.

In the types of methods and basic blocks there is a special security level, referred to as the type of pc. This pc type is related to, but different from, the pc type of JIF. As we shall see, the pc type tracks information encoded by outcome of execution partial instructions. Successful completion tells you that the parameter to the partial instruction was not in the range of values that would have caused the instruction to fail and the other way around for a crash.

Basic block types, ranged over by $\varepsilon$, may be polymorphic in stack types as indicated by the universally quantified stack type variable context, $\Delta_\varepsilon$. From left to right, the type is built up by a stack type, ranged over by $s$, the register bank type, ranged over by $r$, the security level of the pc and the type of the history, ranged over by $\xi$. Stacks are either empty, a primitive security type followed by a stack or a stack variable, $\gamma$. The register bank type is a mapping from register numbers to primitive security types.

The method type is the most complicated of the types. Method types are polymorphic in the security levels defined by the security variable context $\Delta_m$. A method typed $\forall[\cdot].\widetilde{\tau}, \sigma_{ip}, \xi_i \overset{\sigma_{ctx}}{\to} \tau, \sigma_{op}, \xi_o$ accepts parameters of a subtype to $\widetilde{\tau}$, can be run in any environment with a pc type that is a subtype of $\sigma_{ip}$ and a history type that is a subtype of $\xi_i$, returns a value that is a subtype of $\tau$, a new pc type $\sigma_{op}$ and a new history type, $\xi_o$. The method type also has same kind of side effect constraint, $\sigma_{ctx}$ in the example type, as found in e.g. [5,3]. The side effect constraint constrains public output and side effects much like the pc, but is used to retain compositionality in the presence of dynamic invocations on secret objects.

## *5.2 Subtyping*

We define structural subtyping relations in the standard way. For example, if $\tau_1 <: \tau_2$, then $\tau_1$ is a subtype of $\tau_2$. Method subtyping is defined by contravariance in the (ordinary) parameters, the history parameter and the pc parameter, invariance in the side effect constraint and co-variance in all return types. The invariance in the side effect constraint ensures compositionality in the presence of dynamic invocation on secret objects, by ensuring that if one method is free from low side effects, then all methods with the same name must be free from low side effects.

$$\frac{\widetilde{\tau}_2 <: \widetilde{\tau}_1 \quad \begin{array}{c}\sigma_4 <: \sigma_1 \\ \xi_3 <: \xi_1\end{array} \quad \sigma_2 = \sigma_5 \quad \tau_1 <: \tau_2 \quad \begin{array}{c}\sigma_3 <: \sigma_6 \\ \xi_2 <: \xi_4\end{array}}{\forall[\cdot].\widetilde{\tau}_1, \sigma_1, \xi_1 \xrightarrow{\sigma_2} \tau_1, \sigma_3, \xi_2 <: \forall[\cdot].\widetilde{\tau}_2, \sigma_4, \xi_3 \xrightarrow{\sigma_5} \tau_2, \sigma_6, \xi_4}$$

## *5.3 Low Equivalence*

As described in Section 4 non-interference is typically formulated in terms of a low-equivalence relation on environments. Put simply, two environments are low-equivalent w.r.t. an environment type if all parts of the environment classified as public by the environment type are equal. For example, consider two environments $e_1$ and $e_2$ both containing a variable $a$. If variable $a$ is classified as public by the environment type then the value of $a$ should be equal in $e_1$ and $e_2$. The direct extension of this scheme to handle heaps is to demand the rooted low-reachable graphs of both environments be isomorphic. Locations related by this isomorphism are then demanded to have equal values.

Picture 5 illustrates the idea by depicting two environments, $e_1$ on the left half of the picture, and $e_2$ on the right, and the low-reachable tree rooted in the first few registers with the dashed arcs representing the isomorphism. The type of register $r1$ ($ptr\ L\ L$) dictates that the structure (class) of the object to be equal as well as the contents of the object. The type of register $r2$ forces the contents of the register to be equal. Going one step further into the tree the type of the first field of $C$ demands the (shared) low contents of the object to be equal but not structure of the pointed objects. The safety of this interpretation is depending on the opaqueness of bytecode pointers.

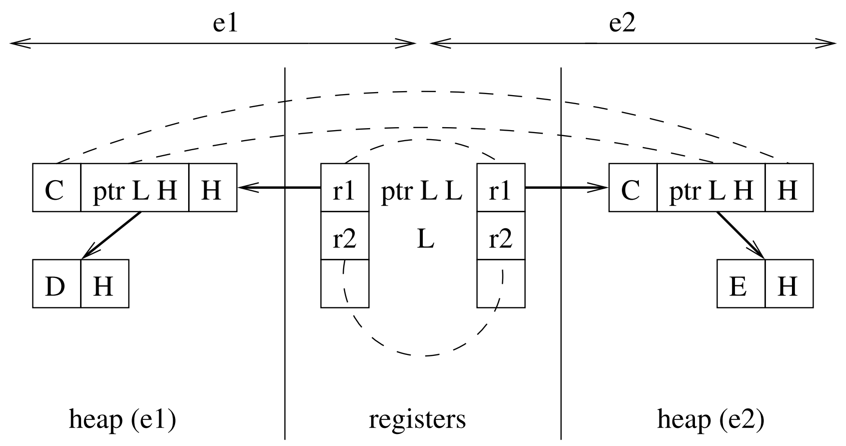


Fig. 5. Low Equivalence

## *5.4 Type Rules*

This section presents and discusses a small selection of the type rules, chosen to explain the key ideas of the type system. The rules for instructions are of the form:

$$\frac{p_1 \dots p_n}{\Sigma; \Gamma \vdash i \Rightarrow \Gamma'}$$

which is read as instruction $i$ is type correct in the static environment $\Sigma$, consisting of the program under consideration, the return types and other constraints of the method, and the environment $\Gamma$, producing the environment, $\Gamma'$. Let $\Gamma$ and $\Sigma$ refer to the initial environment and static environment in all the rules. $\Gamma$ ranges over quantifier free environment types.

Common to all rules is the the history premise of the form $\phi_i(\Gamma) = \xi_2$, for an instruction, $i$. This premise ties the rule to the timing model by producing a new history type from the initial environment type, $\Gamma$, which contains the old history type and, where necessary, types of the parameters of the instruction.

### *5.4.1 Direct Time and Termination Leakages*

We assume that that neither termination nor the cause of termination is directly visible to the attacker. This allows us to have partial instructions with secret parameters. If termination were directly observable, information about the (partial) parameters would leak. For example, if the execution of a division operation does not crash we know that the value of the divisor was distinct from 0. In our case non-termination can only be signalled to the outside using public outputs. Thus, as long as public output is prohibited after the execution of a partial instruction on secret information nothing is leaked, which gives us the possibility to continue with unconstrained computation on secret information after the low communication has ceased. The primary use of the pc type is in the type rule of the $\mathbf{output}_L$ instruction,

$$\frac{\psi_{(\xi_1)} <: L \quad \sigma_p <: L \quad \phi_{\mathbf{output}_L}(\Gamma) = \xi_2}{\Sigma; int_L \cdot s, r, \sigma_p, \xi_1 \vdash \mathbf{output}_L \Rightarrow s, r, \sigma_p, \xi_2}$$

which disallows low output if the pc, $\sigma_p$, is not $L$. This rule is interesting for other reasons as well. Being the only source of publicly observable behaviour, the $\mathbf{output}_L$ instruction is the only instruction to extract the time type from the history type. If the time type is $H$, this means that the time of the execution of the instruction may been affected by secrets, for instance by looping on a secret or by the creation of an array of secret size. Under such circumstances public output must be prohibited, since the time of the output would give away information on the secret responsible for the time differences. Thus, the type rule demands that the extracted time is $L$ in the premise $\psi_{(\xi_1)} <: L$. Obviously, the value to be output must also be public. For example, assume that the top of the stack is a secret integer, i.e. that the stack type is $int_H \cdot \gamma$, then the following programs would be ill typed:

```
                     anew (int L)       iconst 1
output L             const 1            div
                     output L           output L
```

### *5.4.2 Implicit Information Flows*

In the style of Agat[1] we use a semantic side condition, low-observable equivalence, in the type rule of **if**@-instructions to prevent implicit and timing flows. Intuitively, the side condition is used to make sure that it is impossible to tell which branch was taken by inspecting the publicly observable behaviour of the execution, or the low part of the contents of the resulting environment.

#### Low-Observable Equivalence

More precisely, we define low-observable equivalence in terms of trace equivalence. For a body to be low-observably equivalent it has to be that all traces through the body of the branch, e.g. all traces beginning at the top of either of the branches, produce the same observable output at the same time and, if terminating normally by reaching the end of the high security context, their resulting environments are low-equal w.r.t. the environment type of the merge point. To illustrate trace equivalence, consider the following for two traces starting in $l_1$ and $l_2$ respectively ending in the least merge point $l_e$.

$$\begin{array}{ccc} (l_1, e_1) \overset{o_1,t_1}{\rightsquigarrow} & \ldots & \overset{o_n,t_n}{\rightsquigarrow} (l_2, e_2) \\ =_{\theta,\Gamma} & \cdots & =_{\theta,\Gamma} \\ (l_e, e_1') \overset{o_1,t_1}{\rightsquigarrow} & \ldots & \overset{o_n,t_n}{\rightsquigarrow} (l_e, e_2') \end{array}$$

For high security contexts beginning at $l_1, l_2$ ending in any of the labels $l_e \in ls_e$ we define low-observational equivalence ($\simeq$), w.r.t. the initial and the final environment type of the body of the context, $\Gamma$ and $\Gamma_e$ respectively. To allow for arbitrary calling contexts we quantify over all partial bijections compatible with the bijection induced by the type on the initial environment.

$$l_1 \simeq^{\Gamma,\Gamma_e}_{ls_e} l_2 \equiv \forall e_1, e_2, \theta . e_1 =_{\theta,\Gamma} e_2 \Longrightarrow (l_1, e_1) \sim^{\Gamma_e}_{ls_e,\theta} (l_2, e_2)$$

Let $o$ denote a non-empty sequence of output and $\twoheadrightarrow_{ls_e}$ be a big-step reduction up to any label in $ls_e$ without observable output. Trace equivalence is formulated as follows:

$$(l_1, e_1) \sim^{\Gamma_e}_{ls_e,\theta} (l_2, e_2) \equiv \forall \widetilde{o}, \widetilde{t}, l_1', l_2', e_1', e_2'.$$
$$(((l_1, e_1) \overset{\widetilde{o,t}}{\rightsquigarrow}{}^{+}_{ls_e} (l_1', e_1') \iff (l_2, e_2) \overset{\widetilde{o,t}}{\rightsquigarrow}{}^{+}_{ls_e} (l_2', e_2')) \wedge (l_1', e_1') \sim^{\Gamma_e}_{ls_e,\theta} (l_2', e_2')) \vee$$
$$\exists \theta' . (((l_1, e_1) \twoheadrightarrow_{ls_e} e_1' \iff (l_2, e_2) \twoheadrightarrow_{ls_e} e_2') \wedge \theta \subseteq \theta' \wedge e_1' =_{\theta',\Gamma_e} e_2')$$

Naturally, since objects may be allocated within the high security context we allow the partial bijection to be extended to include them. For simplicity we are assuming that the domain of the environment is growing monotonically, i.e. that there is no garbage collector.

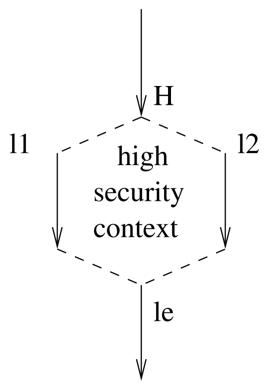
#### Instantiations of the Side Condition

The role of the side condition is to express the when secret branches are free from implicit flows in general terms without mandating the implementation of the enforcing algorithm. The benefits of using a side condition like this is that it allows you to instantiate the type system with different implementations as long as they guarantee the properties demanded by the side condition.

If we temporarily ignore timing, one way to approximate the side condition is the widely used technique to prohibit modification of public environment locations from high security contexts. This can easily be seen to approximate the semantic side condition (under the assumption that the whole program has been typed): if low modification is prohibited then so is low-output and, trivially, all low output will be equal. Furthermore, since the body of the high security context is prohibited from modifying the low parts of the memory no new low objects can be allocated and all public environment locations must be left untouched.

As shown in [1][Paper II] we can formulate "non-assignment" instructions, which are skip instruction sequence timing equivalent to a reformulation of the ordinary assignment instruction. Using this fact we can approximate the side condition by (syntactically) demanding that each high modification is matched by a high "non-modification", each low-modification (including low output) is matched by an equivalent low-modification in lock-step. While this may seem to be an unreasonable demand for real world programs, it can be very useful for programs transformed using the cross-copy idea from [2]. Although Agat only argued correctness for a functional time model, this approach is also sound for a simple history-based cache model, since it preserves the sequence of memory accesses.

Being the only source of implicit flows, the type rule for the **if**@-instruction is the only place where the side condition is used. Assuming a branch, $\mathbf{if}@_{l_1,l_2,l_e}$, which is the head of a high security context (i.e. with a secret parameter) implicit flows are prohibited by demanding low-observational equivalence for all traces from $l_1$ and $l_2$ up to the merge point $l_e$ using the semantic side condition. Let $\delta$ range over substitutions that closes both $\Gamma$ and $\Gamma_e$.



$$\frac{\begin{array}{c}\phi_{\mathbf{if}@}(\Gamma) = \xi_2 \\ \Gamma' = s, r, \sigma_p, \xi_2 \quad \mathit{typeof}(\Sigma, l_e) = \forall[\Delta_\varepsilon].\Gamma_e \\ \sigma \neq L \wedge \neg(\sigma <: \sigma_p) \Longrightarrow \\ \forall\delta.\delta(\sigma) = H \Longrightarrow l_1 \simeq^{\delta(\Gamma'),\delta(\Gamma_e)}_{\{l_e\}} l_2\end{array}}{\Sigma; \mathit{int}_\sigma \cdot s, r, \sigma_p, \xi_1 \vdash \ \mathbf{if}@_{l_1,l_2,l_e} \ \Rightarrow \Gamma'}$$

Fig. 6. Conditional Branch

To exemplify the use of the side condition and flowing security levels, consider the tiny program in Figure 7, in which register $r1$ is public inside the high security context but secret at the merge point since it takes on different values

```
     ...
 l1: (s: (int H) . a) (r1 : int L, r2 : int H) pc t
     if l2 l3 l4
 l2: (s: a) (r1 : int L, r2 : int H) pc t
     const 1
     dup
     store 1
     store 2
     goto l4
 l3: (s: a) (r1 : int L, r2 : int H) pc t
     const 2
     store 1
     const 1
     store 2
 l4: (s: a) (r1 : int H, r2 : int L) pc H
     ...
```

Fig. 7. Example of Conditional Branch

depending on which branch was taken, whereas register $r2$ may be considered public, even though it was secret inside the context, because it will always be equal to 1 when reaching $l4$. Because no partial instructions are used the piece of code is polymorphic in the type of pc. Assuming a constant time model, we see that the program will reach $l4$ at different times depending on which branch is taken, which is reflected by the type of time going $H$ on the type of label $l4$.

**Pointer Types**

Another of the distinct features of the type language is the type of pointers, which differentiates between the security level of the pointer value, and the structure of the pointed object, which is illustrated in the type rule of the **length** instruction,

$$\frac{\phi_{\mathbf{length}}(\Gamma) = \xi_2}{\Sigma; aptr_{\sigma_1,\sigma_2}\ \tau \cdot s, r, \sigma_p, \xi_1 \vdash \mathbf{length} \Rightarrow int_{\sigma_2} \cdot s, r, \sigma_p, \xi_2}$$

where the returned type of **length** only reflects the structure type of the array pointer, which is safe since the length instruction cannot distinguish between two similarly sized arrays. This feature is potentially useful together with the Agat style side condition, which allows for precise approximations of low-observability. The pc is not affected by the type of the pointer, which is safe if we define the length of a null-pointer to be 0.

### *5.4.3 Programs, Methods and Basic Blocks*

A program is type correct if all the methods of the program are type correct.

A method is type correct if all its basic blocks are type correct w.r.t. the side effect constraint and the return types of the method. Furthermore, the type of the initial basic block should be compatible with the execution environment provided by the type of the method.

A basic block is type correct if its instruction sequence is type correct in the entry type of the basic block, producing an exit type compatible with all the successors of the basic block.

## 6 Conclusion and Future Work

We have formulated a timing aware information-flow type system for a subset of a bytecode-like language. Our method generalizes previous attempts to model time by parameterizing the semantics and the type system with a time model. The project is still in its infancy and much work remains to be done. As mentioned above, the type system has been formulated with a correctness proof in mind, but any such proof remains an important part of the future work. We have created a prototype implementation and begun initial evaluation by a series of rudimentary tests, including the implementation of modular exponentiation [9]. The prototype implementation does type checking only, which does not scale well to real sized programs because of the size of the types. In this paper we present slightly simplified object types, with one security type per class. This is satisfactory for whole-program analyses but problematic in the light of compositionality and inferability since there is no most general security type for the field of a class. If a field is neither forced public or secret by the method of a class we could choose either, but neither choice would fit all programs. For instance, imagine we create a container class of some kind. If we choose the elements to be public we cannot store secrets into the container and vice versa. Without support from the type system we would end up having to implement two different containers: one for public and one for secret information. For a two level security lattice this may be acceptable but for a richer structure it certainly is not. One natural remedy to this problem is to allow object types to be polymorphic in the security levels restricted to uniform recursion to avoid problems with cyclic objects. We believe these types to be inferable, and an implementation of inference is forthcoming.

**Acknowledgements** Thanks to the referees, and to Ulf Norell, Andrei Sabelfeld and Kyle Ross for helpful suggestions. The work was partially funded by the Swedish Foundation for Strategic Research (SSF), VR and Vinnova.

## References

[1] J. Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation.* PhD thesis, Chalmers University of Technology and Gothenburg University, Gothenburg, Sweden, December 2000.

[2] Johan Agat. Transforming out timing leaks. In *POPL*, pages 40–53, 2000.

[3] A. Banerjee and D. Naumann. Secure information flow and pointer confinement

in a java-like language. In *Proc. IEEE Computer Security Foundations Workshop*, pages 253–267, June 2002.

[4] Gilles Barthe, Amitabh Basu, and Tamara Rezk. Security types preserving compilation. In *Proceeding of VMCAI'04*, volume 2937 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[5] Gilles Barthe and Tamara Rezk. Secure information flow for a sequential java virtual machine. TLDI'05: Types in Language Design and Implementation.

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

[7] Samir Genaim and Fausto Spoto. Information flow analysis for java bytecode. To appear in: Proc. of the Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05).

[8] Naoki Kobayashi and Keita Shirane. Type-based information flow analysis for low-level languages. English version in 3rd Asian Workshop on Programming Languages and Systems (APLAS'02), Japanese (full) version in Computer Software 20(2), Iwanami Press, pp.2-21, 2003.

[9] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113, 1996.

[10] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[11] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In Xavier Leroy and Atsushi Ohori, editors, *1998 Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52, Kyoto, Japan, March 1998. Springer-Verlag.

[12] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. Software release. http://www.cs.cornell.edu/jif, July 2001.

[13] Francois Pottier and Vincent Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 319–330, Portland, Oregon, January 2002.

[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] Vincent Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.

[16] Raymie Stata and Martin Abadi. A type system for java bytecode subroutines. In *POPL*, January 1998.

[17] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. *Proc. 10th IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.