Transforming out Timing Leaks

Johan Agat

Department of Computing Science
Chalmers University of Technology and Göteborg University
agat@cs.chalmers.se

Abstract

One aspect of security in mobile code is privacy: private (or secret) data should not be leaked to unauthorised agents. Most of the work on secure information flow has until recently only been concerned with detecting direct and indirect flows. Secret information can however be leaked to the attacker also through covert channels. It is very reasonable to assume that the attacker, even as an external observer, can monitor the timing (including termination) behaviour of the program. Thus to claim a program secure, the security analysis must take also these into account.

In this work we present a surprisingly simple solution to the problem of detecting timing leakages to external observers. Our system consists of a type system in which well-typed programs do not leak secret information directly, indirectly or through timing, and a transformation for removing timing leakages. For any program that is well typed according to Volpano and Smith [VS97a], our transformation generates a program that is also free of timing leaks.

1 Introduction

As the use of Internet and mobile code increases, prevention of security leakages in multilevel secure systems becomes a concern for the everyday user. In a multilevel secure systems, data with different security levels are processed and computed. The security levels are partially ordered and normally form a lattice. To maintain confidentiality in such a system, secret data must not be leaked to unauthorised users or flow downwards in the security lattice. This work is motivated by the need for privacy of secret data when code from an untrusted source is down-loaded and run on, typically, a naive users computer. Since virtually everybody stores data they consider private (or secret) on their computers, some kind of guarantee for the privacy of that data is needed when untrusted code is executed.

The scenario we imagine is the following: A user downloads a program from an untrusted web-site. To compute some information valuable to the user, the program needs access to the users private data. What is problematic is that the program might also need to access some databases over the Internet in order to function. Is it safe for the user to run the program or could the private data be leaked?

A concrete example of such a program could be an automated financial advisor that needs the users private financial information as input and will fetch stock market rates, bank loan interest rates etc. over the Internet while computing financial advice for the user.

In this paper, we will deal with sequential programs and security leakage to an attacker external to the system running the untrusted code. Secret or private information may be leaked to the external attacker in a multitude of different ways:

- *Direct leakage*, the simplest and most blunt way of leaking information, is when the secret data is just passed to the attacker as is.
- *Indirect leakage* (sometimes called leakage through a *covert storage channel*), is when the secret data is encoded in the observable behaviour of the program. The program might for instance perform different kinds of database accesses depending on the value of some secret data.
- *Timing leakage* (or leakage through a *covert timing channel*) occurs when the program encodes the secret data in its timing behaviour or manipulates some shared resource in such a way that the attacker can observe the manipulation by measuring the availability of the resource during a particular time interval.
- Programs may also leak information through their termination behaviour. That is given that the secret data satisfies some condition the program may terminate abnormally or go into a infinite loop. We consider leaking through nontermination to be a special case of timing leakage.

Use of covert timing channels is maybe the most cunning way to leak data and perhaps also the most difficult to detect and prevent. Luckily, network latencies etc., introduce a lot of noise that must be dealt with by an implementation of a timing channel over the Internet. This means that the capacity of such a channel is probably quite low. However, timing channels cannot be discarded when reasoning about the security of a system dealing with sensitive information. Even if the capacity of the timing channel is as low as 2 bits per minute, leaking a 16 digit VISA card number can

```
volvoValue := 0;
i := 1;
while (i<=DBsize) {
  let share := sharesDB[i].name  in
  let value := lookupVal(share)*sharesDB[i].no in
    if (isVolvoShare(share))
      volvoValue := volvoValue+value;
    i := i + 1
}
```

Figure 1: A program with a timing leak.

```
volvoValue := 0;
i := 1;
while (i<=DBsize) {
  let share := sharesDB[i].name  in
  let value := lookupVal(share)*sharesDB[i].no in
    if (isVolvoShare(share))
      volvoValue := volvoValue+value
    else
      skipAsn volvoValue (volvoValue+value);
    i := i + 1
}
```

Figure 2: A padded, secure version of the program.

be done in less than 30 minutes[1]. Also, combined with the ability to make repeated attacks, timing can disclose disastrously sensitive information to the attacker. For example, Kocher showed [Koc96], that some implementations of the RSA encryption algorithm leak information about the encryption key through their timing behaviour. By making a series of encryptions and measuring their times, the attacker could figure out the entire key.

1.1 Example

To illustrate how timing leaks can arise and also how they can be closed, we consider two program fragments. Figure 1 presents a program that computes the total value of the user's Volvo shares into the secret variable `volvoValue`. The program loops through a database, `sharesDB`, implemented as an array of records with two secret components: a string `name` and an integer `no`. The length of the database is public. No secret data will be leaked to the public variable `i`, but since the amount of computation performed in the loop is dependent on whether `sharesDB[i].name` is a Volvo share, the program will leak this information through its timing behaviour.

This timing leak can be closed simply by padding the program with dummy computation, as shown in Figure 2. The command `skipAsn x` e takes the same time to execute as `x :=` e but does not do any assignment. Assuming that the functions `lookupVal` and `isVolvoShare` both execute in constant time, the program is secure since the contents of `sharesDB` will not influence the execution time of the while-loops body.

[1]One decimal digit requires roughly 3.3 bits to encode ($2^{3.3} \approx 10$).

1.2 Contribution

We present a type-system and a type-directed transformation that removes timing leaks from programs to make them secure with respect to a semantic security condition based on bisimulation. The security condition is very strong and the programs that are considered secure do not leak any secret information directly, indirectly, by termination behaviour or through covert timing channels to external observers. Programs that are well typed in our type-system satisfy this security condition. Our type-system improves over existing ones [VS97a], in that it allows more secure programs according to our condition. The generality of our type-system makes type checking undecidable, but we provide a simple, decidable, type-directed transformation that gives well-typed (secure) programs. The transformation removes timing leaks from programs without direct and indirect leaks, by padding with dummy instructions where needed. Together, our transformation and type-system provide the first realistic solution for closing timing leaks to external observers.

2 A Semantic Security Condition

Essential to analysing the security properties of a program is to have a semantic notion of security. The condition of security used must be strong enough to capture all important leakages possible in a realistic implementation of the system. In this section, we first point to problems with the realism and strength of the security condition used in some related work. We then discuss what a realistic security condition should capture, present the language we use and finally define our condition of security.

To simplify the presentation of our system we consider only two security classes: L for low security (public) and H for high security (secret). We thus have a two-point security lattice: $\text{L} \leq \text{H}$.

2.1 Related Work

The most commonly used semantic security condition is that of *noninterference* [GM82], which has been adopted in many recent papers on secure flow analysis [VSI96, VS97b, HR98]. A program satisfies the noninterference property if its low security outputs do not depend on the high security inputs. This can be formulated as:

$$\forall E_1, E_2.\ E_1 =_{\text{low}} E_2 \Rightarrow P(E_1) =_{\text{low}} P(E_2)$$

where $E_1 =_{\text{low}} E_2$ means that if the environments E_1 and E_2 are defined, all their low-security components are equal. Here the program is seen as a function on environments.

With noninterference as the underlying notion of security, type-based analyses capable of detecting direct and indirect security leakages have been proposed by Volpano and Smith [VS97b] (a reformulation of Dennings work [Den76, DD77]), and Heintze and Riecke [HR98]. A big drawback with noninterference, however, is the extensional view of programs as functions from input to output. Thus a program that is secure in this sense can still leak information through its timing behaviour.

To close leakage through nontermination, Volpano and Smith enforce a condition of termination agreement, where both the termination behaviour and low output of a secure program are independent of the high inputs [VS97a]. Practically, this was done by disallowing looping conditions to depend on high security data. The same restriction was enforced in [SV98] to prevent nontermination to be used as a method of leaking information between parallel processes in a multi-threaded language. By forcing the arguments to partial primitive operators, like division, to be of lowest security, leakages trough abnormal termination can be closed [VS97a].

When it comes to detecting leakages through covert timing channels, Volpano and Smith have taken two different approaches [VS97a, VS98]. Neither of these deal with timing leakage to external observers in a feasible way. In [VS97a], a theorem on *timing agreement* is formulated. This theorem states that sequential programs with both looping- and branching conditions independent of high data will execute in lock-step with the low security part of the environment independent of the high security data. The consequences of disallowing also branching on high data will be discussed in Section 3. Although not a security condition in itself, the timing agreement theorem contains some of the necessary ingredients for a timing-aware security condition.

Internal timing leakages between concurrent threads arising from the probabilistic behaviour of the scheduler is studied in [VS98]. A type system and a semantics based on Markov chains of probabilistic states and stochastic transition matrices are presented, but no explicit security criterion is given. The system's security is based on the use of a '`protect` C' statement that ensures atomic execution of the command C. Properly implemented and used on all if-commands branching on high data, the `protect`-statement can eliminate internal timing leakage. However, (quote) *"if external observation of the running program is allowed, then of course covert channels of the kind discussed* [...] *remain possible"*.

The language JFlow, presented by Myers in [Mye99], is an extension of Java with annotations for secure information flow. The annotations can be checked mostly-statically but some tests need to be performed at runtime. Leakages through timing or nontermination are not detected in JFlow. As a way of escaping the restrictiveness of strict information flow, the system allows declassification of secret data, which makes formal reasoning about the security of the system hard. No semantic security condition or soundness theorem is given.

Banâtre, Bryce and Le Métayer [BBL94], present an information flow logic in which proofs of (potential) flows between variables are derived. The correctness criterion given is essentially noninterference on a variable by variable basis. The logic does not deal with external observers or flows resulting from timing or termination behaviour.

Methods of detecting potential timing channels when both the sending and receiving processes are available for analysis have been proposed by He and Gligor in [HG92]. This work is rather informal and does not contain any formal semantic condition of security.

A method described for example in [Heh84, Nie84], is that of using a *programmed counter* to measure execution time. The idea is to transform the program so that a special program variable introduced to record execution time is incremented after each statement. Rustan, Leino and Joshi [RLJ98], suggest using this approach to reason about covert flows involving timing behaviour. Introducing a low-security counter variable that is incremented after each command can certainly give a timing-aware semantic security condition but this approach works poorly in combination with existing type system for security. The problem is that incrementing a low-security counter variable after each command conflicts with the requirement that low-security variables are not assigned in the branches of high if-commands. This means that unless the incrementation of the counter variable is given some special treatment, all high if-commands will seem to have indirect leaks. We have thus chosen to build a notion of execution time into the semantics, as presented in Section 2.3.

2.2 Requirements on Secure Programs

To arrive at a realistic and adequately strong security condition, we must analyse which kinds of information leakages should be prevented. In our setting, code is down-loaded from an untrusted source, given secret data as input and also allowed to access resources over Internet. The attacker in this setting is the author of the code and possibly in control of the site from which it is down-loaded. The attacker is also external to the system running the code. When the program runs, it can communicate with the attacker in two different ways:

Immediate communication, where data is sent directly to the attacker, that is the site from which the program was loaded. This is the only communication allowed by Java applets (see e.g. [DFWB97]).

Indirect communication, by use of a third party. The communication to the attacker is made by sending data to or manipulating some another site on the net in a way that the attacker can monitor.

Since the attacker is external to the system running the program, and thus cannot be controlled by that system, we must assume that the attacker can observe both *what* is communicated by the program and *when* this communication is made. From the attackers view, the execution of the program can be seen as a (possibly infinite) sequence of time, value pairs that correspond to observable actions and when they are made. Our semantic security condition must thus force this sequence to be independent of the high security data that the program manipulates.

Forcing the observable actions of a program to be independent of high data poses no new problems. This is just the noninterference property. To make sure also that the time at which these actions are performed is independent of high data, we must in some way reason about the execution time of programs. It is hard to determine how fine-grained this reasoning should be. Obviously, the number of instructions and which operations that are performed affect the execution time and so does the hardware (and interpreter) on which the program is run. The kind of instructions executed must probably also be considered, but even this might not be enough. A subtle way of leaking would still be available: the program can exhibit different cache, and thereby time, behaviour depending on some high data. Consider the following piece of C code:

```
if (h)
  for(n=0; n<N; n++) xs[n]++; /* loop 1 */
else
  for(n=0; n<N; n++) ys[n]++; /* loop 2 */
for(n=N; n>=0; n--) xs[n]++;  /* loop 3 */
```

The same number and the same kind of instructions will be executed by both branches but if loop 1 is executed the

Operators	$op ::= \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{=} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=}$
Expressions	$e ::= l \mid e\ op\ e \mid le$
Initialisers	$ie ::= e \mid \texttt{mkarray}(e)\ ie \mid \{\texttt{x}_1 = ie_1, \ldots, \texttt{x}_n = ie_n\}$
Commands	$C, D ::= le \texttt{ := } e \mid \texttt{skipAsn}\ le\ e \mid \texttt{if}\,(e)\ C\ \texttt{else}\ D \mid \texttt{skipIf}\ e\,C \mid \texttt{let x :=}\ ie\ \texttt{in}\ C \mid \texttt{while}\,(e)\ C \mid C\,;D \mid \texttt{output x}$
Left-expressions	$le ::= \texttt{x} \mid le\texttt{.x} \mid le[e]$
Left-values	$lv ::= \texttt{x} \mid lv\texttt{.x} \mid lv[n]$
Values	$v ::= l \mid \{\texttt{x}_1 = v_1, \ldots, \texttt{x}_n = v_n\} \mid [v_0, \ldots, v_n]$
Basic Values	$l ::= n \mid \texttt{true} \mid \texttt{false}$

Figure 3: Syntax of expressions, commands and values.

array `xs` will probably be cached when loop 3 is reached. It is thus likely that the entire execution will be longer if `h` is false[2]. The variance in timing here might not be big but it is probably big enough to implement a timing channel! The code shown above is not secure with our criterion given in Definition 2, nor is it well-typed in our type-system. Our type system will only allow a high if-command when the two branches have exactly the same pattern of allocations and variable references. Thus the example above will be considered insecure even without loop 3.

To avoid building a limitation into the system, we have implicitly parameterised our semantics and security condition on the interpretation of time. This is discussed somewhat more in Sections 2.3 and 2.4.

2.3 The Language

The language we use, with syntax given in Figure 3, has assignments, sequencing, conditionals, local bindings, while loops, output commands and two kinds of skip-commands: `skipAsn` $le\ e$ and `skipIf` $e\,C$. The transformation described in Section 4 inserts these skip-commands to remove timing leakages. Values consists of records, arrays, integers and booleans. An expression cannot construct a record or array but this can be done by an initialising expression when a new binding is introduced. The `output` construct implements externally observable actions by outputting the value of an integer variable to the attacker.

To give the semantics of programs we use two partial functions: a big-step natural semantics for expressions and a labelled small-step transition semantics for commands. Semantic rules for expressions, initialisers and left-expressions are standard and given in Figure 4. Evaluation judgements for commands are of the forms

$$\langle E \,|\, C\rangle \xrightarrow{as} \langle E \,|\, D\rangle$$

$$\langle E \,|\, C\rangle \xrightarrow{ts\cdot\checkmark} E$$

where E is the environment, associating variables with values. The annotations on the transition arrow is a possibly empty sequence of time-expressions, ts, and for as also output actions. These describe the time and observable actions of making that transition. Output actions, ranged over by o, are integer values, n and the symbol $\checkmark$ which is used to signal termination. There is at most one output action on any given transition. We write $as_1 \cdot as_2$ for the concatenation of two sequences and we consider a single action or time-expression as a singleton sequence. We use ts to range over sequences consisting only of time-expressions, which are:

t_e (t_{le})	The time it takes to evaluate e (or le).
t_{asn}	Time for making an assignment.
t_{br}	Time to inspect and branch on a value.
$t_{pu\ ie}$	Time to evaluate ie and push a new binding of the resulting value on the evaluation environment.
t_{po}	Time to pop the innermost binding from the evaluation environment.

To capture the time of evaluating expressions, the time-expressions t and t_{pu} are indexed on expressions and initialisers. Since only basic values can be assigned (assignment of arrays- and record-values is not allowed) it is enough to use a constant t_{asn} for the time of an assignment.

Note that time-expressions do not specify time directly. They are only descriptions of time and have to be given some kind of interpretation if we need to know the real time they denote. Such an interpretation can be arbitrarily complex. By looking at the entire history of the execution, an interpretation could even model data-cache behaviour to some extent. The semantics is parameterised on the interpretation of time. Any interpretation can be used as long as primitive operations are given constant times. This is discussed in Section 2.4.

Rules for the evaluation of commands are given in Figure 5. The environment E is a sequence of bindings but for convenience we also treat it as a mapping, writing $E(\texttt{x})$ to access the value in the rightmost (innermost) binding of x. We write $E[lv = l]$ for updating E at the binding and component indicated by lv to the basic value l and we define $dom(E)$ as $\{lv | \exists l. E(lv) = l\}$.

The two skip-constructs, `skipAsn` and `skipIf`, are designed to have the same timing behaviour as an assignment and an if-branching respectively. In the type-system described in Section 3, `skipAsn` and `skipIf` are used in describing timing behaviour. Programs can be padded with appropriate skip-commands as done by the transformation presented in Section 4, thereby removing timing leakages.

All other rules for evaluation of commands are fairly standard. Evaluation of a `let` introduces a new local binding. We have chosen to represent local bindings syntactically with a let-like construct called `local`, and they are moved to and from the environment in the rule (Local). We use the notation $E, \texttt{x} = v$ for adding a new binding to E and also to match out the innermost binding of an environment. The time for pushing a binding on the evaluation environment is paid in the (Let)-rule when the binding is first introduced and the time for popping the binding is paid when the body of the `let` terminates. There are no time-annotations for the cost of shuffling bindings in and out of the environment made in the (Local) rule. This is of course due to the fact that the `local` construct is nothing but a notational trick to keep track of local and global variables and it has no corresponding implementation overhead. Another technical trick we use is that the `output` command evaluates to a `skip` that then simply terminates, thus we avoid the problem of more than one output action on any given transition.

[2]A simple test based on the code above showed that a difference of 2-3 seconds in execution time could be measured from a program that ran in 1 minute. The test was made on a 296 MHz Sun Ultra 4.

$$\text{(Lit)}\ \frac{}{E \vdash l \Downarrow l} \qquad \text{(Var)}\ \frac{E(\mathtt{x}) = v}{E \vdash \mathtt{x} \Downarrow v} \qquad \text{(Op)}\ \frac{E \vdash e_1 \Downarrow l_1 \quad E \vdash e_2 \Downarrow l_2}{E \vdash e_1\, op\, e_2 \Downarrow l_1 [\![op]\!] l_2} \qquad \text{(Ix)}\ \frac{E \vdash e_1 \Downarrow [v_0, \dots, v_n, \dots, v_m] \quad E \vdash e_2 \Downarrow n}{E \vdash e_1[e_2] \Downarrow v_n}$$

$$\text{(Sel)}\ \frac{E \vdash e \Downarrow \{\dots, \mathtt{x} = v, \dots\}}{E \vdash e.\mathtt{x} \Downarrow v} \qquad \text{(MkA)}\ \frac{E \vdash e \Downarrow n \quad E \vdash ie \Downarrow v}{E \vdash \mathtt{mkarray(}e\mathtt{)}\ ie \Downarrow [v, \dots, v]\ (n \text{ occurences})} \qquad \text{(L-Ix)}\ \frac{E \vdash le \Downarrow^{\mathrm{L}} lv \quad E \vdash e \Downarrow n}{E \vdash le[e] \Downarrow^{\mathrm{L}} lv[n]}$$

$$\text{(MkRec)}\ \frac{E \vdash ie_1 \Downarrow v_1 \ \dots\ E \vdash ie_n \Downarrow v_n}{E \vdash \{\mathtt{x}_1 = ie_1, \dots, \mathtt{x}_n = ie_n\} \Downarrow \{\mathtt{x}_1 = v_1, \dots, \mathtt{x}_n = v_n\}} \qquad \text{(L-Sel)}\ \frac{E \vdash le \Downarrow^{\mathrm{L}} lv}{E \vdash le.\mathtt{x} \Downarrow^{\mathrm{L}} lv.\mathtt{x}} \qquad \text{(L-Var)}\ \frac{}{E \vdash \mathtt{x} \Downarrow^{\mathrm{L}} \mathtt{x}}$$

Figure 4: Big-step semantics for expressions and initialisers ($\Downarrow$) and left-expressions ($\Downarrow^{\mathrm{L}}$).

$$\text{(Assign)}\ \frac{E \vdash e \Downarrow l \quad E \vdash le \Downarrow^{\mathrm{L}} lv}{\langle E \,|\, le \mathtt{\ :=\ } e\rangle \xrightarrow{t_e \cdot t_{le} \cdot t_{asn} \cdot \checkmark} E[lv = l]}\ lv \in dom(E) \qquad \text{(Let)}\ \frac{E \vdash ie \Downarrow v}{\langle E \,|\mathtt{let\ x\ :=\ } ie \mathtt{\ in\ } C\rangle \xrightarrow{t_{pu\,ie}} \langle E \,|\mathtt{local\ x\ :=\ } v \mathtt{\ in\ } C\rangle}$$

$$\text{(Seq)}\ \frac{\langle E \,|\, C\rangle \xrightarrow{ts \cdot \checkmark} E'}{\langle E \,|\, C; D\rangle \xrightarrow{ts} \langle E' \,|\, D\rangle} \qquad \frac{\langle E \,|\, C\rangle \xrightarrow{as} \langle E' \,|\, C'\rangle}{\langle E \,|\, C; D\rangle \xrightarrow{as} \langle E' \,|\, C'; D\rangle}$$

$$\text{(Local)}\ \frac{\langle E, \mathtt{x} = v \,|\, C\rangle \xrightarrow{as} \langle E', \mathtt{x} = v' \,|\, D\rangle}{\langle E \,|\mathtt{local\ x\ :=\ } v \mathtt{\ in\ } C\rangle \xrightarrow{as} \langle E' \,|\mathtt{local\ x\ :=\ } v' \mathtt{\ in\ } D\rangle} \qquad \frac{\langle E, \mathtt{x} = v \,|\, C\rangle \xrightarrow{ts \cdot \checkmark} E', \mathtt{x} = v'}{\langle E \,|\mathtt{local\ x\ :=\ } v \mathtt{\ in\ } C\rangle \xrightarrow{ts \cdot t_{po} \cdot \checkmark} E'}$$

$$\text{(If)}\ \frac{E \vdash e \Downarrow \mathtt{true}}{\langle E \,|\mathtt{if\ (}e\mathtt{)\ } C \mathtt{\ else\ } D\rangle \xrightarrow{t_e \cdot t_{br}} \langle E \,|\, C\rangle} \qquad \frac{E \vdash e \Downarrow \mathtt{false}}{\langle E \,|\mathtt{if\ (}e\mathtt{)\ } C \mathtt{\ else\ } D\rangle \xrightarrow{t_e \cdot t_{br}} \langle E \,|\, D\rangle}$$

$$\text{(While)}\ \frac{E \vdash e \Downarrow \mathtt{false}}{\langle E \,|\mathtt{while\ (}e\mathtt{)\ } C\rangle \xrightarrow{t_e \cdot t_{br} \cdot \checkmark} E} \qquad \frac{E \vdash e \Downarrow \mathtt{true}}{\langle E \,|\mathtt{while\ (}e\mathtt{)\ } C\rangle \xrightarrow{t_e \cdot t_{br}} \langle E \,|\, C; \mathtt{while\ (}e\mathtt{)\ } C\rangle}$$

$$\text{(SkipAsn)}\ \frac{E \vdash e \Downarrow l \quad E \vdash le \Downarrow^{\mathrm{L}} lv}{\langle E \,|\mathtt{skipAsn}\ le\ e\rangle \xrightarrow{t_e \cdot t_{le} \cdot t_{asn} \cdot \checkmark} E}\ lv \in dom(E) \qquad \text{(SkipIf)}\ \frac{E \vdash e \Downarrow l}{\langle E \,|\mathtt{skipIf}\ e\, C\rangle \xrightarrow{t_e \cdot t_{br}} \langle E \,|\, C\rangle}\ l \in \{\mathtt{true}, \mathtt{false}\}$$

$$\text{(Skip)}\ \frac{}{\langle E \,|\mathtt{skip}\rangle \xrightarrow{\checkmark} E} \qquad \text{(Output)}\ \frac{E(\mathtt{x}) = n}{\langle E \,|\mathtt{output\ x}\ \rangle \xrightarrow{t_{\mathtt{x}} \cdot n} \langle \mathtt{skip} \,|\, E\rangle}$$

Figure 5: Small-step semantics for commands.

2.4 A Bisimulation-based Security Condition

As a foundation for our semantic security condition, we use a partial bisimulation on commands, $\sim_\Gamma$. We index this relation on a typing environment, Γ, to distinguish low- and high-security left-values. Informally, two commands are Γ-bisimular if they behave stepwise identically with respect to execution time, outputs and the manipulation of the low variables (according to Γ). Also, this behaviour must be independent of the high data in the environment.

Definition 1 (Γ-bisimulation)
$\sim_\Gamma$ is the largest symmetric relation on commands that satisfies:

$C_1 \sim_\Gamma C_2$ if $\forall E_1, E_2$ such that $E_1 =_\Gamma E_2$ we have that

$$\begin{aligned} \langle E_1 \,|\, C_1\rangle \xrightarrow{as} \langle E_1' \,|\, D_1\rangle &\Rightarrow \langle E_2 \,|\, C_2\rangle \xrightarrow{as} \langle E_2' \,|\, D_2\rangle \wedge \\ &\qquad E_1' =_\Gamma E_2' \wedge D_1 \sim_\Gamma D_2 \\ \langle E_1 \,|\, C_1\rangle \xrightarrow{ts \cdot \checkmark} E_1' &\Rightarrow \langle E_2 \,|\, C_2\rangle \xrightarrow{ts \cdot \checkmark} E_2' \wedge E_1' =_\Gamma E_2' \end{aligned}$$

where $=_\Gamma$ is defined as

$$\begin{aligned} E_1 =_\Gamma E_2 \text{ iff } & dom(E_1) = dom(E_2) \wedge dom(E_1) \subseteq dom(\Gamma) \wedge \\ & \forall lv \in dom(E_1).\Gamma(lv) = \tau_{\mathrm{L}} \Rightarrow E_1(lv) = E_2(lv) \end{aligned}$$

Here, Γ is a typing environment and τ_{L} is a base type of low security level. Both will be introduced in Section 3.1

By relating commands to themselves by Γ-bisimulation we get a sort of timing-aware, stepwise noninterference property, which suits us well as the definition of security:

Definition 2 (Γ-security)

$$C \text{ is } \Gamma\text{-secure if } C \sim_\Gamma C$$

This security condition is semantically a lot stronger than actually necessary, due to the definition of Γ-bisimulation. For example, due to the statelessness of $\sim_\Gamma$ the program h := 0; l := h is not secure, although it would not leak any secret information in our sequential setting. Weaker and more elaborate versions of the bisimulation could be defined. The relation could be made stateful by dropping the local quantification of the environments and relate configurations instead of commands. Moreover, as it stands, $\sim_\Gamma$, requires the evaluation of $\langle E_1 | C_1 \rangle$ and $\langle E_2 | C_2 \rangle$ to both make a step with syntactically equal time-expression sequences, which essentially forces them to make the same computations (but not the same assignments!). This condition could be relaxed by making the interpretation of time-expression sequences explicit and by forcing the two commands to evaluate in times that have the same interpretation, possibly involving the entire history of the two executions. Any weaker version of the bisimulation would clearly be implied by $\sim_\Gamma$. To avoid any unnecessarily complicated definition and since the type-system and transformation presented in Sections 3 and 4 will be sound with respect to $\sim_\Gamma$, we have chosen this, stronger definition. Also, the statelessness of $\sim_\Gamma$ gives a composable security criterion and extends nicely to satisfy the Hook-up property in parallel programs [McC87], as shown by Sands and Sabelfeld [SS99].

It is of course very hard to statically predict cache behaviour. However by forcing two Γ-bisimular commands to have syntactically equal time-expression sequences, the two commands must obey the same pattern of allocations and variable references. This severely reduces the possibilities for timing leaks based on cache behaviour.

The only assumption on the interpretation of time inherent in the definition of $\sim_\Gamma$, is that a time-expression sequence *ts* must denote the same time regardless of the values bound to the free high-security variables in *ts*. This means that primitive operations must be performed in constant time, independent of the argument values of the operator, and that the length of an array cannot be secret.

It might be worth pointing out that $\sim_\Gamma$ is only a partial bisimulation in that it is *not* reflexive. Insecure commands, like l := h, are not related to themselves, which is also the reason that $\sim_\Gamma$ is not a congruence. Γ-bisimulation thus differs from ordinary bisimulation by not being an equivalence relation, but rather a partial such:

Lemma 1 ($\sim_\Gamma$ is a PER)
The relation $\sim_\Gamma$ is a Partial Equivalence Relation, i.e. it is symmetric, transitive but not necessarily reflexive.

Proof: $\sim_\Gamma$ is symmetric by definition, and transitivity is easy to show since $=_\Gamma$ is transitive.

Due to the symmetry and transitivity of $\sim_\Gamma$, if a command is Γ-bisimular to anything, it is also Γ-bisimular to itself, and hence Γ-secure.

2.5 Example of a Secure Program

Even though Γ-bisimulation is a very strong security criterion, it does not rule out useful programs. If we informally extend the language with functions and strings, the padded version of the Volvo shares example, presented in Figure 2, is Γ-secure. Looping through the database does not leak since the length of the database is public. Branching on the (secret) value of `isVolvoShare(share)` is also secure since the two branches are Γ-bisimular. Of course, the example is only Γ-secure provided that the two functions `lookupVal` and `isVolvoShare` are both Γ-secure.

$$\frac{s_1 \le s_2}{\bar{\tau}_{s_1} \le \bar{\tau}_{s_2}} \qquad \frac{\tau_1 \le \tau_2}{\mathrm{A}\,\tau_1 \le \mathrm{A}\,\tau_2}$$

$$\frac{\tau_1 \le \tau_1' \ \ldots\ \tau_n \le \tau_n'}{\{\mathtt{x}_1 : \tau_1, \ldots, \mathtt{x}_n : \tau_n\} \le \{\mathtt{x}_1 : \tau_1', \ldots, \mathtt{x}_n : \tau_n'\}}$$

Figure 6: The subtyping relation

3 Typing Secure Programs

In [VS97a], Volpano and Smith present a type-system in which well-typed programs are secure with respect to timing leaks to external observers. By a simple inductive reasoning, it can be shown that programs typed according to section 5 of [VS97a] are also Γ-secure. Our critique to this type-system is that it is too restrictive to be of any practical use: it requires the condition of both while-loops and if-commands to be of lowest security. With this restriction, high data can only be copied around and passed to (total) primitive operators, which means that only programs that are essentially parametric in their high inputs are considered secure by the type-system. Our observation to relax this restrictiveness is the following:

A secure program may safely branch on high data as long as the external observer cannot determine which branch was taken.

Our type-system closes timing and termination channels in well-typed programs by requiring looping conditions to be of low security and branches of high if-commands to have the same externally observable behaviour.

3.1 The Type System

We use the following language of types:

Security levels	$s ::= \mathrm{L} \mid \mathrm{H}$	(with $\mathrm{L} \le \mathrm{H}$ and $s \le s$)
Base types	$\bar{\tau} ::= \mathrm{Int} \mid \mathrm{Bool}$	
Security types	$\tau ::= \bar{\tau}_s \mid \mathrm{A}\,\tau \mid \{\mathtt{x}_1 : \tau_1, \ldots, \mathtt{x}_n : \tau_n\}$	

Security types, ranged over by τ, are base types annotated with a security level, array-, or record types. The components of records can all have different security types and thereby security levels. The type for arrays contains the type for the elements in the array, but there is no security level for the length of the array, since this must always be low. Allowing high values to specify the length of an array will open up for timing leakages of the same kind as loops on high data, since the array is initialised upon creation. Also, array update and indexing are non-total operations, so we cannot allow high-security values to be used in specifying the index in these operations. This problem, and our treatment of it, is analogous to that with the division operator in [VS97a]. In Figure 6, we extend the ordering on security levels to a subtyping relation.

The typing of expressions calculate an upper bound on the security levels of the variables in the expression. This typing has no time component since expressions evaluate atomically and are side-effect free in our semantics. Typing rules for expressions, initialisers and left-expressions are

$$\text{(LitInt)}\ \frac{}{\Gamma \vdash_{\lhd} n : \mathrm{Int}_s} \qquad \text{(LitBool)}\ \frac{}{\Gamma \vdash_{\lhd} \{\mathtt{true},\mathtt{false}\} : \mathrm{Bool}_s} \qquad \text{(Var)}\ \frac{\Gamma(\mathtt{x}) \lhd \tau}{\Gamma \vdash_{\lhd} \mathtt{x} : \tau} \qquad \text{(Op-Arithm)}\ \frac{\Gamma \vdash_{\le} e_1 : \mathrm{Int}_s \quad \Gamma \vdash_{\le} e_2 : \mathrm{Int}_s}{\Gamma \vdash_{\lhd} e_1\{\mathtt{+},\mathtt{-},\mathtt{*}\}e_2 : \mathrm{Int}_s}$$

$$\text{(Ix)}\ \frac{\Gamma \vdash_{\lhd} e_1 : \mathrm{A}\,\tau \quad \Gamma \vdash_{\le} e_2 : \mathrm{Int}_{\mathrm{L}}}{\Gamma \vdash_{\lhd} e_1[e_2] : \tau} \qquad \text{(Sel)}\ \frac{\Gamma \vdash_{\lhd} e : \{\ldots,\mathtt{x}:\tau,\ldots\}}{\Gamma \vdash_{\lhd} e.\mathtt{x} : \tau} \qquad \text{(Op-Cmp)}\ \frac{\Gamma \vdash_{\le} e_1 : \mathrm{Int}_s \quad \Gamma \vdash_{\le} e_2 : \mathrm{Int}_s}{\Gamma \vdash_{\lhd} e_1\{\mathtt{=},\mathtt{<=},\mathtt{<}\}e_2 : \mathrm{Bool}_s}$$

$$\text{(MkA)}\ \frac{\Gamma \vdash_{\le} e : \mathrm{Int}_{\mathrm{L}} \quad \Gamma \vdash_{\le} ie : \tau}{\Gamma \vdash_{\lhd} \mathtt{mkarray(}e\mathtt{)}\ ie : \mathrm{A}\,\tau} \qquad \text{(MkRec)}\ \frac{\Gamma \vdash_{\le} ie_1 : \tau_1 \ \ldots\ \Gamma \vdash_{\le} ie_n : \tau_n}{\Gamma \vdash_{\lhd} \{\mathtt{x}_1 = ie_1,\ldots,\mathtt{x}_n = ie_n\} : \{\mathtt{x}_1 : \tau_1,\ldots,\mathtt{x}_n : \tau_n\}}$$

Figure 7: Typing rules for expressions, initialisers and left-expressions. ($\lhd$ ranges over $=$ and $\le$)

$$\text{(Assign}_{\mathrm{H}}\text{)}\ \frac{\Gamma \vdash_{\le} e : \bar{\tau}_s \quad \Gamma \vdash_{=} le : \bar{\tau}_{\mathrm{H}} \quad s \le \mathrm{H}}{\Gamma \vdash le \mathtt{\ :=\ } e : \mathtt{skipAsn}\ le\ e} \qquad \text{(SkipAsn)}\ \frac{}{\Gamma \vdash \mathtt{skipAsn}\ le\ e : \mathtt{skipAsn}\ le\ e}$$

$$\text{(Assign}_{\mathrm{L}}\text{)}\ \frac{\Gamma \vdash_{\le} e : \bar{\tau}_{\mathrm{L}} \quad \Gamma \vdash_{=} le : \bar{\tau}_{\mathrm{L}}}{\Gamma \vdash le \mathtt{\ :=\ } e : le \mathtt{\ :=\ } e} \qquad \text{(SkipIf)}\ \frac{\Gamma \vdash C : C_{\mathrm{L}}}{\Gamma \vdash \mathtt{skipIf}\ e\,C : \mathtt{skipIf}\ e\,C_{\mathrm{L}}}$$

$$\text{(Seq)}\ \frac{\Gamma \vdash C : C_{\mathrm{L}} \quad \Gamma \vdash D : D_{\mathrm{L}}}{\Gamma \vdash C; D : C_{\mathrm{L}}; D_{\mathrm{L}}} \qquad \text{(Let)}\ \frac{\Gamma \vdash_{\le} ie : \tau \quad \Gamma, \mathtt{x} : \tau \vdash C : C_{\mathrm{L}}}{\Gamma \vdash \mathtt{let\ x\ :=\ } ie \mathtt{\ in\ } C : \mathtt{let\ x\ :=\ } ie \mathtt{\ in\ } C_{\mathrm{L}}}$$

$$\text{(If}_{\mathrm{H}}\text{)}\ \frac{\Gamma \vdash_{\le} e : \mathrm{Bool}_{\mathrm{H}} \quad \Gamma \vdash C : C_{\mathrm{L}} \quad \Gamma \vdash D : D_{\mathrm{L}}}{\Gamma \vdash \mathtt{if\ (}e\mathtt{)}\ C \mathtt{\ else\ } D : \mathtt{skipIf}\ e\,C_{\mathrm{L}}}\ C_{\mathrm{L}} \sim_{\Gamma} D_{\mathrm{L}} \qquad \text{(While)}\ \frac{\Gamma \vdash_{\le} e : \mathrm{Bool}_{\mathrm{L}} \quad \Gamma \vdash C : C_{\mathrm{L}}}{\Gamma \vdash \mathtt{while\ (}e\mathtt{)}\ C : \mathtt{while\ (}e\mathtt{)}\ C_{\mathrm{L}}}$$

$$\text{(If}_{\mathrm{L}}\text{)}\ \frac{\Gamma \vdash_{\le} e : \mathrm{Bool}_{\mathrm{L}} \quad \Gamma \vdash C : C_{\mathrm{L}} \quad \Gamma \vdash D : D_{\mathrm{L}}}{\Gamma \vdash \mathtt{if\ (}e\mathtt{)}\ C \mathtt{\ else\ } D : \mathtt{if\ (}e\mathtt{)}\ C_{\mathrm{L}} \mathtt{\ else\ } D_{\mathrm{L}}} \qquad \text{(Output)}\ \frac{\Gamma(\mathtt{x}) = \mathrm{Int}_{\mathrm{L}}}{\Gamma \vdash \mathtt{output\ x} : \mathtt{output\ x}}$$

Figure 8: Typing rules for commands

given in Figure 7. These rules are parameterised on whether sub typing is allowed for variables or not, using $\lhd$ to range over $=$ and $\le$. The Assign rules in Figure 8 use the parameterisation to prevent sub typing from being used in typing left-expressions. Apart from using a richer type language, our way of typing expressions is essentially identical to that of Volpano and Smith [VS97b].

Typing environments, mapping variables to security types, are ranged over by Γ and we write $\Gamma, \mathtt{x} : \tau$ for the environment that maps $\mathtt{x}$ to τ and otherwise behaves like Γ. We write $\Gamma(\mathtt{x}) = \tau$ if $\mathtt{x}$ is mapped to τ by Γ, and $\Gamma(\mathtt{x}) \le \tau$ if there exists some τ' such that $\Gamma(\mathtt{x}) = \tau'$ and $\tau' \le \tau$. We write $\Gamma(lv) = \tau$ as a shorter notation for $\Gamma \vdash_{=} lv : \tau$ and we define $dom(\Gamma)$ as $\{lv | \exists\tau.\Gamma(lv) = \tau\}$.

Typing rules for commands are given in Figure 8. The typing judgements are of the form:

$$\Gamma \vdash C : C_{\mathrm{L}}$$

The "type" of a command C is its *low-slice*, C_{L}. The low-slice is syntactically identical to C but only contains assignments to low security left-values. All assignments to high security left-values and branching on high security data are replaced with the appropriate skips. For example, with the environment $\Gamma = \{\mathtt{h} : \mathrm{Int}_{\mathrm{H}}, \mathtt{l} : \mathrm{Int}_{\mathrm{L}}\}$, we can derive:

$$\Gamma \vdash (\mathtt{h\ :=\ h+1; l\ :=\ l*4}) : (\mathtt{skipAsn\ h\ (h+1); l\ :=\ l*4})$$

As stated in Lemma 2, the low-slice has the same observable behaviour as the original command with respect to low left-values. Note that although there are similarities, the low-slice is not a program slice in the conventional sense (see e.g. [RT96]). The low-slice and the type-systems construction of it has more in common with the extraction of static program parts as described by Mogensen in [Mog89].

The typing rules are all rather straightforward. Direct leakage is prevented by the Assign$_{\mathrm{L}}$-rule. By forcing the looping condition to be of low security, the While-rule prevents leakage through nontermination and also blocks a simple kind of timing leaks. The most interesting rule is If$_{\mathrm{H}}$, which stops both indirect- and timing leaks. It allows branching on high data provided that the branches have Γ-bisimular low-slices and thus have the same externally observable behaviour. The low slice is $\mathtt{skipIf}\ e\,C_{\mathrm{L}}$ but could just as well have been $\mathtt{skipIf}\ e\,D_{\mathrm{L}}$ or $\mathtt{if\ (}e\mathtt{)}\ C_{\mathrm{L}}\mathtt{\ else\ }D_{\mathrm{L}}$ instead, since they are all Γ-bisimular. By using $\mathtt{skipIf}\ e C_{\mathrm{L}}$ instead of $\mathtt{if\ (}e\mathtt{)}\ C_{\mathrm{L}}\mathtt{\ else\ }D_{\mathrm{L}}$ in the transformation described in Section 4, we avoid getting an exponential blow-up in code size.

3.2 Usefulness in Practice

Our system guarantees that well-typed programs do not leak information to an external observer even through timing behaviour. Unlike the system described in [VS97a], ours is not too restrictive to be used in practice since we do not disallow well-typed programs to compute with, and branch on secret data. For example, the program in Figure 2 is well-typed in our system but not in [VS97a].

The side condition $C_L \sim_\Gamma D_L$ in the If$_H$-rule makes the type system undecidable since $\sim_\Gamma$ is undecidable. This does not prevent sound but incomplete and conservative type checking algorithms to be made though. Given an $O(n)$ method of computing $C_L \sim_\Gamma D_L$, type checking will not be harder than normal [VS97b].

It can be argued that no programmers write programs where all if-commands branching on high data have Γ-bisimular branches. As much as this is true, it is also necessary to produce such programs if we want to avoid timing leaks. To reduce the burden for the programmer, the next section presents a transformation that removes timing leaks by making branches of high if-commands Γ-bisimular.

3.3 Soundness of the Type System

The type system described in Figure 8 is sound with respect to Γ-bisimulation. The soundness theorem is a corollary of the following lemma, stating that the low-slice of a command has the same observable behaviour as the command itself.

Lemma 2 (Typing and Γ-bisimulation)
If $\Gamma \vdash C : C_L$ then $C \sim_\Gamma C_L$

A detailed proof of this lemma is given in Appendix A.

Theorem 1 (Well-typed programs are secure)

$$\text{If } \Gamma \vdash C : C_L \text{ then } C \text{ is } \Gamma\text{-secure.}$$

Proof: Follows from Lemma 2 and the symmetry and transitivity of $\sim_\Gamma$ according to Lemma 1.

4 Transforming out Timing Leaks

Paying with performance, we can transform out the timing leakages of programs by padding the branches of high if-commands with dummy instructions so that they get the same timing and otherwise externally observable behaviour. In Figure 9 we give an inductively defined algorithm that transforms out timing leakages. The algorithm subsumes the type-system and works by recursively performing a crosswise padding of each branch of a high if-commands with the low-slice of the other branch. Thereby producing a program where the low-slices of the branches of high if-commands are Γ-bisimular. Transformation judgements are of the form:

$$\Gamma \vdash C \hookrightarrow D \,|\, D_L$$

In a given type environment, the command C is transformed to an almost semantically equivalent Γ-secure command, D, and to the low-slice, D_L, of this command. The commands D and C are semantically equivalent in the sense that they will perform the same sequence of outputs and assignments to global variables given identical environments to start the evaluation in. They differ in that D might need more evaluation steps than C to do a particular assignment. Also, D might go into a nonterminating, non-productive loop even if C does not. We argue that this is acceptable since the extra nontermination in D will be due to potential information leaks by nontermination in C. We state the semantic soundness of the transformation formally in Theorem 3.

The crosswise padding with low-slices is done in the If$_H$-rule. The function $\text{ge}(C)$, inductively defined in Figure 10, is used to ensure that the low-slices of the transformed branches are free from outputs and assignments to other than variables local to the branches. Thus, the low-slices used for padding will add only time, not observable actions, to the computation – a requirement which is essential to the semantic soundness of the transformation. The premises $\text{ge}(D_{1L}) = \emptyset$ and $\text{ge}(D_{2L}) = \emptyset$ essentially requires both branches to be without output actions and assignments to low security variables bound outside the branches. This requirement stops programs with indirect leaks and is but a variant of that made in [VS97b], where only assignments to high variables are allowed in the branches of a high if-command. Since assignments to low variables locally let-bound in the branches is not restricted, they may contain arbitrary loop-structures (which then also will be present in the low-slices of the branches). The extra nontermination that may be introduced by the transformation is due to nonterminating loops in one branch being copied over into the other.

4.1 The Cost in Performance and Code Size

Performing a crosswise copying of the low-slices is a simple but not very refined way of making the branches Γ-bisimular. Ideally, we would like to pad each branch with the difference to the maximum execution time, rather than taking the sum of the execution times of the two branches. Computing the difference is undecidable in general but a more refined method of padding than cross copying can certainly be defined.

To try to give some kind of formal argument about the slow-down of the transformed program relative the original one is pointless. The best we can say is that the transformed program can take arbitrarily longer time to execute, since even code that was semantically dead in the original program may be executed in the transformed one. For a tree-structure of nested high if-commands, the original program would execute only one path in the tree whereas the transformed program will execute *all* paths. As discussed in the previous section, this can lead to nontermination of the transformed program.

One can easily be mislead to think[3] that a combination of nested high if-commands and the cross copying made in the If$_H$ rule of the transformation would lead to exponential blow-up in code size. However, the worst case blow-up in code size from cross copying is actually only by a factor linear in the nesting depth of high if-commands. What saves us from exponential blow-up in code size is that the `skipIf` introduced as low-slice of a transformed high if has roughly half the size of a corresponding if-command with Γ-bisimular branches. To reason formally about the code size blow-up, we begin by defining a simple measurement of code size.

Definition 3 (Code size)
Define ${}^{\#}C$ as:

$$\begin{array}{ll}
{}^{\#}(\texttt{if}\,(e)\ C\ \texttt{else}\ D) = 1 + {}^{\#}C + {}^{\#}D & {}^{\#}(le := e) = 1 \\
{}^{\#}(C; D) = {}^{\#}C + {}^{\#}D & {}^{\#}(\texttt{while}\,(e)\ C) = 1 + {}^{\#}C \\
{}^{\#}(\texttt{skipIf}\ e\, C) = 1 + {}^{\#}C & {}^{\#}(\texttt{let x :=}\ ie\ \texttt{in}\ C) = 1 + {}^{\#}C \\
{}^{\#}(\texttt{skipAsn}\ le\ e) = 1 & {}^{\#}(\texttt{output x }) = 1
\end{array}$$

[3]As the author was in an earlier draft.

(Assign$_{\text{H}}$) $$\frac{\Gamma \vdash_{\leq} e : \bar{\tau}_s \quad \Gamma \vdash_{=} le : \bar{\tau}_{\text{H}} \quad s \leq \text{H}}{\Gamma \vdash le \texttt{ := } e \hookrightarrow le \texttt{ := } e \mid \texttt{skipAsn}\ le\ e}$$

(Assign$_{\text{L}}$) $$\frac{\Gamma \vdash_{\leq} e : \bar{\tau}_{\text{L}} \quad \Gamma \vdash_{=} le : \bar{\tau}_{\text{L}}}{\Gamma \vdash le \texttt{ := } e \hookrightarrow le \texttt{ := } e \mid le \texttt{ := } e}$$

(Seq) $$\frac{\Gamma \vdash C_1 \hookrightarrow D_1 \mid D_{1\,\text{L}} \quad \Gamma \vdash C_2 \hookrightarrow D_2 \mid D_{2\,\text{L}}}{\Gamma \vdash C_1; C_2 \hookrightarrow D_1; D_2 \mid D_{1\,\text{L}}; D_{2\,\text{L}}}$$

(If$_{\text{H}}$) $$\frac{\Gamma \vdash_{\leq} e : \text{Bool}_{\text{H}} \quad \Gamma \vdash C_1 \hookrightarrow D_1 \mid D_{1\,\text{L}} \quad \Gamma \vdash C_2 \hookrightarrow D_2 \mid D_{2\,\text{L}} \quad \mathsf{ge}(D_{1\,\text{L}}) = \emptyset \quad \mathsf{ge}(D_{2\,\text{L}}) = \emptyset}{\Gamma \vdash \texttt{if}\,(e)\ C_1\ \texttt{else}\ C_2 \hookrightarrow \texttt{if}\,(e)\ D_1; D_{2\,\text{L}}\ \texttt{else}\ D_{1\,\text{L}}; D_2 \mid \texttt{skipIf}\ e\ (D_{1\,\text{L}}; D_{2\,\text{L}})}$$

(If$_{\text{L}}$) $$\frac{\Gamma \vdash_{\leq} e : \text{Bool}_{\text{L}} \quad \Gamma \vdash C_1 \hookrightarrow D_1 \mid D_{1\,\text{L}} \quad \Gamma \vdash C_2 \hookrightarrow D_2 \mid D_{2\,\text{L}}}{\Gamma \vdash \texttt{if}\,(e)\ C_1\ \texttt{else}\ C_2 \hookrightarrow \texttt{if}\,(e)\ D_1\ \texttt{else}\ D_2 \mid \texttt{if}\,(e)\ D_{1\,\text{L}}\ \texttt{else}\ D_{2\,\text{L}}}$$

(SkipAsn) $$\frac{}{\Gamma \vdash \texttt{skipAsn}\ le\ e \hookrightarrow \texttt{skipAsn}\ le\ e \mid \texttt{skipAsn}\ le\ e}$$

(SkipIf) $$\frac{\Gamma \vdash C \hookrightarrow D \mid D_{\text{L}}}{\Gamma \vdash \texttt{skipIf}\ e\,C \hookrightarrow \texttt{skipIf}\ e\,D \mid \texttt{skipIf}\ e\,D_{\text{L}}}$$

(Let) $$\frac{\Gamma \vdash_{\leq} ie : \tau \quad \Gamma, \mathtt{x} : \tau \vdash C \hookrightarrow D \mid D_{\text{L}}}{\Gamma \vdash \texttt{let x := } ie\ \texttt{in}\ C \hookrightarrow \texttt{let x := } ie\ \texttt{in}\ D \mid \texttt{let x := } ie\ \texttt{in}\ D_{\text{L}}}$$

(While) $$\frac{\Gamma \vdash_{\leq} e : \text{Bool}_{\text{L}} \quad \Gamma \vdash C \hookrightarrow D \mid D_{\text{L}}}{\Gamma \vdash \texttt{while}\,(e)\ C \hookrightarrow \texttt{while}\,(e)\ D \mid \texttt{while}\,(e)\ D_{\text{L}}}$$

(Output) $$\frac{\Gamma \vdash_{\leq} \mathtt{x} : \text{Int}_{\text{L}}}{\Gamma \vdash \texttt{output x} \hookrightarrow \texttt{output x} \mid \texttt{output x}}$$

Figure 9: An algorithm for transforming out timing leakages

$$\frac{}{\mathsf{ge}(\mathtt{x}) = \{\mathtt{x}\}} \qquad \frac{\mathsf{ge}(le) = \sigma}{\mathsf{ge}(le.\mathtt{x}) = \sigma} \qquad \frac{\mathsf{ge}(le) = \sigma}{\mathsf{ge}(le[e]) = \sigma} \qquad \frac{\mathsf{ge}(le) = \sigma}{\mathsf{ge}(le \texttt{ := } e) = \sigma}$$

$$\frac{\mathsf{ge}(C_1) = \sigma_1 \quad \mathsf{ge}(C_2) = \sigma_2}{\mathsf{ge}(C_1; C_2) = \sigma_1 \cup \sigma_2} \qquad \frac{\mathsf{ge}(C_1) = \sigma_1 \quad \mathsf{ge}(C_2) = \sigma_2}{\mathsf{ge}(\texttt{if}\,(e)\ C_1\ \texttt{else}\ C_2) = \sigma_1 \cup \sigma_2} \qquad \frac{\mathsf{ge}(C) = \sigma}{\mathsf{ge}(\texttt{let x := } ie\ \texttt{in}\ C) = \sigma \setminus \{\mathtt{x}\}}$$

$$\frac{\mathsf{ge}(C) = \sigma}{\mathsf{ge}(\texttt{while}\,(e)\ C) = \sigma} \qquad \frac{}{\mathsf{ge}(\texttt{skipAsn}\ le\ e) = \emptyset} \qquad \frac{\mathsf{ge}(C) = \sigma}{\mathsf{ge}(\texttt{skipIf}\ e\,C) = \sigma} \qquad \frac{}{\mathsf{ge}(\texttt{output x}\ ) = \{\bullet\}}$$

Figure 10: $\mathsf{ge}(C)$ – the global effects of C – free assigned variables and indication of output actions.

Our first observation is that the low slice computed by the transformation will always have the same size as the original command.

Lemma 3 (Size of computed low-slice)
If $\Gamma \vdash C \hookrightarrow D \,|\, D_{\rm L}$ then ${}^{\#}C = {}^{\#}D_{\rm L}$.

Proof: Simple by induction on the height of the transformation derivation.

We will write $\mathrm{nd}_{\Gamma}(C)$ for the maximal nesting depth of high (according to Γ) if-commands in C.

Theorem 2 (Code size blow-up wrt. nesting depth)
If $\Gamma \vdash C \hookrightarrow D \,|\, D_{\rm L}$ and $\mathrm{nd}_{\Gamma}(C) \leq n$ then ${}^{\#}D \leq (n+1){}^{\#}C$.

Proof: By induction on the height of the transformation derivation. Cases Assign$_{\rm H}$, Assign$_{\rm L}$, SkipAsn and Output are all trivial. Cases Seq, If$_{\rm L}$, SkipIf, Let and While all follow immediately from the induction hypothesis and some simple arithmetic. In case If$_{\rm H}$ we have that C is of the form `if` (e) C_1 `else` C_2 and $D =$ `if` (e) $D_1; D_{2\,\rm L}$ `else` $D_{1\,\rm L}; D_2$ where $\Gamma \vdash C_1 \hookrightarrow D_1 | D_{1\,\rm L}$ and $\Gamma \vdash C_2 \hookrightarrow D_2 | D_{2\,\rm L}$. From the assumption that $\mathrm{nd}_{\Gamma}(C) \leq n$ we know that $\mathrm{nd}_{\Gamma}(C_1) \leq n-1$ and $\mathrm{nd}_{\Gamma}(C_2) \leq n-1$. Now from Definition 3 we have

$$\begin{aligned}
{}^{\#}D &= 1 + {}^{\#}D_1 + {}^{\#}D_{2\,\rm L} + {}^{\#}D_{1\,\rm L} + {}^{\#}D_2 & \\
&\leq 1 + {}^{\#}D_{1\,\rm L} + {}^{\#}D_{2\,\rm L} + n{}^{\#}C_1 + n{}^{\#}C_2 & \text{(by ind. hyp.)}\\
&\leq 1 + {}^{\#}C_1 + {}^{\#}C_2 + n({}^{\#}C_1 + {}^{\#}C_2) & \text{(by Lemma 3)}\\
&\leq (n+1){}^{\#}C & \text{(by Definition 3)}
\end{aligned}$$

In the worst case scenario, the high if-commands in a command C are nested to a depth which is linear in the size of C. Such nesting occurs for example in a multiple choice implemented by nesting if-commands in the else-branch and the transformation will in that case blow up C to a command with size in the order of $O({}^{\#}C^2)$. To avoid such quadratic blow-up, deeply nested if-commands can often be flattened. Consider the following program:

```
if (e₁) C₁
else if (e₂) C₂
...
else Cₙ
```

Transforming this program will give a quadratic blow-up in code size, given that all C_i's are relatively small. By introducing a fresh high security variable `x` we can flatten the program as follows:

```
let x := true
in  if (e₁ && x) (C₁; x := false);
    if (e₂ && x) (C₂; x := false);
    ...
    if (x) Cₙ;
```

The size of this program is linear in the size of the original one and the transformation will only blow it up to roughly the double of its size. The amount of computation made in the transformed versions of these two programs is essentially the same. In both cases, one transformed C_i and the low-slices of all transformed C_j, where $i \neq j$, will be executed.

4.2 Semantic Soundness of the Transformation Algorithm

To reason about the semantic soundness of the transformation algorithm, we abstract from the evaluation time. We first introduce a multiple step evaluation $\twoheadrightarrow$.

Definition 4 (Multiple step evaluation)
We define $\twoheadrightarrow$ inductively as:

$$\frac{\langle E \,|\, C\rangle \xrightarrow{ts\cdot\checkmark} E'}{\langle E \,|\, C\rangle \overset{ts\cdot\checkmark}{\twoheadrightarrow} E'} \qquad \frac{\langle E \,|\, C\rangle \xrightarrow{as} \langle E' \,|\, D\rangle}{\langle E \,|\, C\rangle \overset{as}{\twoheadrightarrow} \langle E' \,|\, D\rangle}$$

$$\frac{\langle E \,|\, C\rangle \xrightarrow{as} \langle E' \,|\, D\rangle \quad \langle E' \,|\, D\rangle \overset{as'\cdot\checkmark}{\twoheadrightarrow} E''}{\langle E \,|\, C\rangle \overset{as\cdot as'\cdot\checkmark}{\twoheadrightarrow} E''}$$

$$\frac{\langle E \,|\, C\rangle \xrightarrow{as} \langle E' \,|\, C'\rangle \quad \langle E' \,|\, C'\rangle \overset{as'}{\twoheadrightarrow} \langle E'' \,|\, C''\rangle}{\langle E \,|\, C\rangle \overset{as\cdot as'}{\twoheadrightarrow} \langle E'' \,|\, C''\rangle}$$

To reason about observable evaluation steps we define $\overset{o}{\Longmapsto}$ to be a multiple step evaluation with exactly one output action, namely o.

Definition 5 (Observable evaluation)
Define $\Longmapsto$ as:

$$\begin{aligned}
&\langle E \,|\, C\rangle \overset{o}{\Longmapsto} \langle E' \,|\, D\rangle && \text{iff } \exists ts.\, \langle E \,|\, C\rangle \overset{ts\cdot o}{\twoheadrightarrow} \langle E' \,|\, D\rangle\\
&\langle E \,|\, C\rangle \overset{\checkmark}{\Longmapsto} E' && \text{iff } \exists ts.\, \langle E \,|\, C\rangle \overset{ts\cdot\checkmark}{\twoheadrightarrow} E'
\end{aligned}$$

Definition 6 (Weak simulation)
Define $\sqsubseteq$ as the largest relation that satisfies

$$\begin{aligned}
&C \sqsubseteq D \text{ if } \forall E.\\
&\quad \langle E \,|\, C\rangle \overset{o}{\Longmapsto} \langle E' \,|\, C'\rangle \Rightarrow \langle E \,|\, D\rangle \overset{o}{\Longmapsto} \langle E' \,|\, D'\rangle \wedge C' \sqsubseteq D'\\
&\quad \langle E \,|\, C\rangle \overset{\checkmark}{\Longmapsto} E' \qquad\quad \Rightarrow \langle E \,|\, D\rangle \overset{\checkmark}{\Longmapsto} E'
\end{aligned}$$

It is easy to show that $\sqsubseteq$ is reflexive, transitive and that it is preserved by contexts (i.e. that it is a precongruence).

Our statement of semantic soundness of the transformation is that the transformed program, D, will be simulated by the original program: $D \sqsubseteq C$. The transformation is not complete so $C \sqsubseteq D$ does not hold in general due to the extra nontermination that might be introduced into D. Completeness could be achieved if the transformation was assisted by some kind of termination analysis, so that the algorithm would fail rather than introduce extra nontermination. In Section 4.4, we prove the transformation complete for a certain class of programs.

To prove semantic soundness, we need a few lemmas expressing that only time can be observed from computations that do not perform any outputs or assignments to global variables.

Lemma 4 (Assigned variables)
Write $E \setminus \sigma$ for E with all bindings of variables in σ removed. If $\mathrm{ge}(C) = \sigma$, $\bullet \notin \sigma$ and either $\langle E \,|\, C\rangle \overset{as}{\twoheadrightarrow} \langle E' \,|\, D\rangle$ or $\langle E \,|\, C\rangle \overset{as\cdot\checkmark}{\twoheadrightarrow} E'$, then $E \setminus \sigma = E' \setminus \sigma$ and $\exists ts.\, as = ts$.

Proof: The proof is simple by induction on the height on the derivation of $\mathrm{ge}(C) = \sigma$.

Lemma 5
As a corollary to Lemma 4 we have that:

If $\mathrm{ge}(C) = \emptyset$ and either $\langle E\,|C\rangle \overset{as}{\rightarrow} \langle E'\,|D\rangle$ or $\langle E\,|C\rangle \overset{as\cdot\checkmark}{\rightarrow} E'$, then $E = E'$ and $\exists ts.\, as = ts$

Lemma 6 (Dummy computation)
If $\mathrm{ge}(C) = \emptyset$ then $D; C \sqsubseteq D$ and $C; D \sqsubseteq D$.

Proof: Follows from Lemma 5. We use the same technique as in the proof of Lemma 2 to show that $\{(D; C,\, D)\} \subseteq \sqsubseteq$ and $\{(C; D,\, D)\} \subseteq \sqsubseteq$.

Theorem 3 (Transformation & semantic soundness)

$$\text{If } \Gamma \vdash C \hookrightarrow D \mid D_{\text{L}} \text{ then } D \sqsubseteq C.$$

Proof: We use induction on the height of the transformation derivation. The cases Assign$_{\text{L}}$, Assign$_{\text{H}}$, SkipAsn, SkipIf and Output are all immediate from the reflexivity of $\sqsubseteq$. Cases Seq, If$_{\text{L}}$, Let and While follow from use of the induction hypothesis and the precongruence property of $\sqsubseteq$.

In the case of If$_{\text{H}}$ we know that C is of the form `if` (e) C_1 `else` C_2 and $D =$ `if` (e) $D_1; D_{2\,\text{L}}$ `else` $D_{1\,\text{L}}; D_2$. We also have that $\Gamma \vdash C_1 \hookrightarrow D_1 \mid D_{1\,\text{L}}$, $\Gamma \vdash C_2 \hookrightarrow D_2 \mid D_{2\,\text{L}}$, $\mathrm{ge}(D_{1\,\text{L}}) = \emptyset$ and $\mathrm{ge}(D_{2\,\text{L}}) = \emptyset$. By induction, $D_1 \sqsubseteq C_1$ and $D_2 \sqsubseteq C_2$. So by transitivity and Lemma 6 we have $D_1; D_{2\,\text{L}} \sqsubseteq C_1$ and $D_{1\,\text{L}}; D_2 \sqsubseteq C_2$. Now the precongruence property gives us our goal: $D \sqsubseteq C$.

4.3 Correctness of the Transformation Algorithm

The transformation algorithm is correct in the sense that transformed programs are secure. To prove this we need the idempotence property of the type system.

Lemma 7 (Low-slice idempotence)
If $\Gamma \vdash C : C_{\text{L}}$ then $\Gamma \vdash C_{\text{L}} : C_{\text{L}}$

Proof: Trivial by induction on the height of the typing derivation.

Lemma 8 (Transformation and well-typing)
If $\Gamma \vdash C \hookrightarrow D \mid D_{\text{L}}$ then $\Gamma \vdash D : D_{\text{L}}$.

Proof: A simple inductive reasoning on the depth of the transformation derivation proves this statement. Virtually all cases in the proof are immediate from use of the induction hypothesis and the typing rule corresponding to the transformation rule used.

The only case where some more elaborate reasoning has to be made is for the If$_{\text{H}}$ rule. We then have:

$$\frac{\begin{array}{c}\Gamma \vdash_{\leq} e : \mathrm{Bool}_{\text{H}} \quad \mathrm{ge}(D_{1\,\text{L}}) = \emptyset \quad \mathrm{ge}(D_{2\,\text{L}}) = \emptyset \\ \Gamma \vdash C_1 \hookrightarrow D_1 \mid D_{1\,\text{L}} \quad \Gamma \vdash C_2 \hookrightarrow D_2 \mid D_{2\,\text{L}}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{if}\,(e)\; C_1\; \texttt{else}\; C_2 \hookrightarrow \\ \quad \texttt{if}\,(e)\; D_1; D_{2\,\text{L}}\; \texttt{else}\; D_{1\,\text{L}}; D_2 \mid \texttt{skipIf}\; e\; (D_{1\,\text{L}}; D_{2\,\text{L}})\end{array}}$$

Using the induction hypothesis we get $\Gamma \vdash D_1 : D_{1\,\text{L}}$ and $\Gamma \vdash D_2 : D_{2\,\text{L}}$. From Lemma 7 and typing rule Seq we get $\Gamma \vdash D_1; D_{2\,\text{L}} : D_{1\,\text{L}}; D_{2\,\text{L}}$ and $\Gamma \vdash D_{1\,\text{L}}; D_2 : D_{1\,\text{L}}; D_{2\,\text{L}}$. By Lemma 2 and the symmetry and transitivity of $\sim_\Gamma$ according to Lemma 1, we thus have $D_{1\,\text{L}}; D_{2\,\text{L}} \sim_\Gamma D_{1\,\text{L}}; D_{2\,\text{L}}$. We can then use typing rule If$_{\text{H}}$ to get our goal.

Theorem 4 (Transformation gives secure programs)

$$\text{If } \Gamma \vdash C \hookrightarrow D \mid D_{\text{L}} \text{ then } D \text{ is } \Gamma\text{-secure.}$$

Proof: Follows from Theorem 1 and Lemma 8.

4.4 Completeness w.r.t. Volpano and Smith's system

In section 2 of [VS97a], Volpano and Smith present a type-system in which well typed programs are noninterfering, free of nontermination leakages but may contain timing leaks. Our transformation is complete with respect to this type-system in the sense that well typed programs can be transformed without inserting any nontermination:

> *If C is well-typed according to [VS97a], the transformation will succeed and yield a semantically equivalent program D that is free of timing leaks.*

To state this formally we first define the notion of semantic equivalence we will use.

Definition 7 (Weak bisimulation)
$C \sim D$ iff $C \sqsubseteq D$ and $D \sqsubseteq C$.

Typing environments in [VS97a] are split into two components, $\lambda; \gamma$, mapping *locations* and *identifiers* to security levels. We will not distinguish locations and identifiers, and refer to both as variables.

The language used by Volpano and Smith differs from ours in that integers are the only values used and that observable actions are modelled by assignments to global variables of low security. Moreover, our language has no construct corresponding to the `try` command that guard uses of the division operator. Let P range over programs in [VS97a] *not* containing `try` commands. Define $T(P)$ to be the translation to our syntax that takes each `if`- and `while` condition, e, to e`!=0` and each assignment to a global low variable, `x :=` e, to the sequence `x :=` e`; output x` .

Theorem 5
If $\lambda; \gamma \vdash P : s\,\mathrm{cmd}$ in the system of [VS97a], $C = T(P)$ and $\Gamma = \{\mathtt{x} : \mathrm{Int}_s | \lambda(\mathtt{x}) = s \vee \gamma(\mathtt{x}) = s\}$, then $\Gamma \vdash C \hookrightarrow D \mid D_{\text{L}}$ and $C \sim D$.

Proof: It suffices to show $\Gamma \vdash C \hookrightarrow D \mid D_{\text{L}}$ and $C \sqsubseteq D$. The well typedness of P ensures that all `while` loops have a condition of lowest security and that no loops exists in the branches of `if`-commands with a high condition. We thus use induction on the height of the typing derivation.

4.5 Example: RSA encryption

RSA encryption is based on computing $A = M^e \mod n$, where M is the clear text message and e is the encryption key. To decrypt, $M = A^d \mod n$ is computed, where d is the decryption key. To efficiently compute $X^k \mod n$, the modular exponentiation algorithm can be used, but as shown by Kocher [Koc96], a careless implementation will leak k through timing. In Figure 11, we give an example of such an implementation. We represent k as a `w` elements long array of secret booleans, with the most significant bit of k at k[0]. The program shown is not Γ-secure since the two branches of the `if` have different timing behaviours, neither is the program type correct in our system. The transformation algorithm presented in Figure 9 will close these timing

```
s := 1;
i := 0;
while (i < w) {
  if (k[i])
    r := (s*x) mod n
  else
    r := s;
  s := r*r;
  i := i+1
}              (The result is now in r)
```

Figure 11: An implementation of the modular exponentiation algorithm that leaks through timing.

```
s := 1;
i := 0;
while (i < w) {
  if (k[i])
    r := (s*x) mod n;
    skipAsn r s
  else {
    skipAsn r ((s*x) mod n);
    r := s
  };
  s := r*r;
  i := i+1
}
```

Figure 12: The output of our transformation: a secure implementation of the modular exponentiation algorithm.

leaks by transforming the program into the one given in Figure 12. The security of this, padded program relies on constant time execution of the operators `*` and `mod`. Since r, s and x are bignums in a realistic example, multiplication and modulo will be implemented by subroutines. To get secure implementations of these, the transformation can be used again.

5 Conclusions and Future Work

We have presented a simple and realistic solution to removing timing leaks to external observers, by using a combination of typing and transformation. Our solution is formalised and proved correct. It improves over previous work in that it relaxes the requirement that branching conditions must be of lowest security, thereby allowing a much larger class of programs to be typed.

By parameterising over the interpretation of time, our system becomes very precise. Even some timing leaks implemented by varying data-cache behaviour are detected and deemed insecure by our security criterion and type system. A key assumption built into our system is that primitive operators execute in constant time, regardless of the values given as arguments.

A minor oversimplification in our semantics is that it neglects the unconditional jumps made in the machine code that a compiler produces for if- and while-commands. The code generated for the command `if` (e) C_1 `else` $C2; D$ will typically have the following form:

	$[\![e]\!]$	Code for conditional expression
	`brt L1`	Branch to L1 if the condition was true
	$[\![C_2]\!]$	Code for else-branch
	`jmp L2`	Unconditional jump to L2
`L1:`	$[\![C_1]\!]$	Code for then-branch
`L2:`	$[\![D]\!]$	Code for the rest of the program

Thus, even if C_1 and C_2 execute with the same timing behaviour, the value of e can probably be leaked given that the time for executing `jmp L2` can be noticed. Although performing an unconditional jump often does not take any time at all for modern, heavily pipelined processors, the jump would certainly be noticeable for an interpreted language like Java byte-code. The simplest way to close this possibility to leak secret information is to generate a (redundant) `jmp L2` instruction also after the code for C_1.

As every semantic model abstracts from the real world, so does ours. This means that in an implementation of our system, there will always be well-typed programs that leak information by utilising some aspect of reality not covered by the semantic model. One weakness of our system is that the semantics has no concept of program counter or where in memory a piece of code is physically stored. It is thus impossible for the security condition to tell whether two syntactically identical pieces of code originate from the same location in memory. This sadly opens up for timing leaks implemented by utilising the behaviour of the instruction cache. If and how such leaks can be closed without completely turning off the cache remains to be investigated.

6 Acknowledgements

Thanks to David Sands, Andrei Sabelfeld, Jörgen Gustavsson and the Multi-group at Chalmers in general for encouragement and helpful comments during the development of this work. Also thanks to Pablo Giambiagi, Walid Taha and the anonymous reviewers for suggesting improvements and for spotting several typos and misplaced squiggles in preliminary versions of this paper.

References

[BBL94] J.-P. Banatre, C. Bryce, and D. Le Metayer. Compile-time detection of information flow in sequential programs. *Lecture Notes in Computer Science*, 875:55–73, 1994.

[DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[Den76] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[DFWB97] D. Dean, E. W. Felten, D. . Wallach, and D. Balfanz. Java security: Web browsers and beyond. Technical Report TR-566-97, Princeton University, Computer Science Department, February 1997.

[GM82] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, April 1982.

[Gor94] A. Gordon. A tutorial on co-induction and functional programming. In *Glasgow functional programming workshop*, pages 78–95. Springer Workshops in Computing, 1994.

[Heh84] E. C. R. Hehner. Predicative programming part 1. *Communications of the ACM*, 27(2):134–143, February 1984.

[HG92] J. He and V. D. Gligor. Formal methods and automated tool for timing-channel identification in tcb source code. In *In Proceedings 2nd European Symposium on Research in Computer Security*, LNCS 648, pages 57–75, November 1992.

[HR98] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 365–377. ACM, 1998.

[Koc96] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

[McC87] D. McCullough. Specifications for multi-level security and hook-up property. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 161–166. IEEE Computer Society Press, 1987.

[Mog89] T.Æ. Mogensen. Separating binding times in language specifications. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture '89, Imperial College, London*, pages 12–25, New York, NY, 1989. ACM.

[Mye99] A.C. Myers. JFlow: practical mostly-static information flow control. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, pages 228–241, New York, NY 10036, USA, 1999. ACM Press.

[Nie84] H.R. Nielson. *Hoare Logic's for Run-time Analysis of Programs.* Ph.D. thesis, CST-30-84, Edinburgh University, 1984.

[RLJ98] K. Rustan, M. Leino, and R. Joshi. A semantic approach to secure information flow. *Lecture Notes in Computer Science*, 1422:254–271, 1998.

[RT96] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glueck, and P. Thiemann, editors, *Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 409–429. Springer-Verlag, Feb 1996.

[SS99] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. Unpublished (http://www.cs.chalmers.se/~dave/papers/prob-sabelfeld-sands.ps), June; Revised October 1999.

[SV98] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, 19–21 January 1998.

[VS97a] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. *Proc. 10th IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

[VS97b] D. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT'97*, volume 1214 of *LNCS*, pages 607–621. Springer-Verlag, April 1997.

[VS98] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Proc. 11th IEEE Computer Security Foundations Workshop*, pages 34–43, June 1998.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):1–21, 1996.

A Proof of Lemma 2

To prove Lemma 2 we will need the following small additional lemmas.

Lemma 9 (High left-expressions)
If $\Gamma \vdash_{=} le : \bar{\tau}_{\text{H}}$, $E_1 =_\Gamma E_2$ and $E_1 \vdash le \Downarrow^{\text{L}} lv$
then $E_2 \vdash le \Downarrow^{\text{L}} lv$.

Proof: By induction over the height of the evaluation derivation. Uses the fact that only expressions of type Int_{L} may be used for indexing.

Lemma 10 (Well-typed expressions)
If $\Gamma \vdash_{\leq} ie : \tau$, $E_1 \vdash ie \Downarrow v$ and $E_1 =_\Gamma E_2$
then $\Gamma \vdash_{\leq} v : \tau$, $E_2 \vdash ie \Downarrow v'$ and $\Gamma \vdash_{\leq} v' : \tau$,

Proof: By induction over the height of the evaluation derivation.

Lemma 2 (Typing and Γ-bisimulation)

$$\text{If } \Gamma \vdash C : C_{\text{L}} \text{ then } C \sim_\Gamma C_{\text{L}}$$

Proof: We prove this lemma by induction on the height of the typing derivation and by cases on its structure. In each case we provide a set X, such that $(C, C_{\text{L}}) \in X$. To prove $X \subseteq \sim_\Gamma$ and thereby $C \sim_\Gamma C_{\text{L}}$, we use the standard technique of proving that $X \subseteq F_\Gamma(X \cup \sim_\Gamma)$, where F_Γ is the dense bisimulation operator that has $\sim_\Gamma$ as its largest fix-point (see e.g. [Gor94]). F_Γ is defined according to:

$$\begin{aligned}
F_\Gamma(X) = \{(C_1, C_2) \mid\ & \langle E_1 | C_1\rangle \xrightarrow{ts\cdot\checkmark} E_1' \Rightarrow \\
& \langle E_2 | C_2\rangle \xrightarrow{ts\cdot\checkmark} E_2' \wedge E_1' =_\Gamma E_2'\} \cup \\
\{(C_1, C_2) \mid\ & \langle E_2 | C_2\rangle \xrightarrow{ts\cdot\checkmark} E_2' \Rightarrow \\
& \langle E_1 | C_1\rangle \xrightarrow{ts\cdot\checkmark} E_1' \wedge E_1' =_\Gamma E_2'\} \cup \\
\{(C_1, C_2) \mid\ & \langle E_1 | C_1\rangle \xrightarrow{as} \langle E_1' | D_1\rangle \Rightarrow \\
& \langle E_2 | C_2\rangle \xrightarrow{as} \langle E_2' | D_2\rangle \wedge \\
& E_1' =_\Gamma E_2' \ \wedge\ (D_1, D_2) \in X\} \cup \\
\{(C_1, C_2) \mid\ & \langle E_2 | C_2\rangle \xrightarrow{as} \langle E_2' | D_2\rangle \Rightarrow \\
& \langle E_1 | C_1\rangle \xrightarrow{as} \langle E_1' | D_1\rangle \wedge \\
& E_1' =_\Gamma E_2' \ \wedge\ (D_1, D_2) \in X\}
\end{aligned}$$

The proof is now by induction on the height of the typing derivation, with a case analysis on the outermost typing rule used.

Assign$_{\text{L}}$: C is of the form $le := e$ and $C_{\text{L}} = C$. Choose $X = \{(C, C_{\text{L}})\}$.

Assign$_{\text{H}}$: C is of the form $le := e$ and $C_{\text{L}} =$ `skipAsn` $le\ e$. We also have that $\Gamma \vdash_{\leq} e : \bar{\tau}_s$ and $\Gamma \vdash_{=} le : \bar{\tau}_{\text{H}}$, where $s \leq \text{H}$. Given E_1 and E_2 such that $E_1 =_\Gamma E_2$, we assume

$$\langle E_1 | le := e\rangle \xrightarrow{t_e \cdot t_{le} \cdot t_{asn} \cdot \checkmark} E_1'$$

From semantic rule Assign we then have that $E_1 \vdash e \Downarrow l$, $E_1 \vdash le \Downarrow^{\text{L}} lv$, $lv \in dom(E_1)$ and $E_1[lv = l]$. The definition of $=_\Gamma$ gives us $lv \in dom(E_2)$. By Lemma 9 we get $E_2 \vdash le \Downarrow^{\text{L}} lv$. From Lemma 10 we get $E_2 \vdash e \Downarrow l'$, and semantic rule SkipAsn gives us

$$\langle E_2 | \texttt{skipAsn}\ le\ e\rangle \xrightarrow{t_e \cdot t_{le} \cdot t_{asn} \cdot \checkmark} E_2$$

Since lv is a high left-value w.r.t. Γ, we have that $E_1' =_\Gamma E_2$. Thus it is enough to choose $X = \{(C, C_{\text{L}})\}$.

Seq: C is of the form $C_1; C_2$ and $C_{\text{L}} = D_1; D_2$, where $\Gamma \vdash C_1 : D_1$ and $\Gamma \vdash C_2 : D_2$. By induction, $C_1 \sim_\Gamma D_1$ and $C_2 \sim_\Gamma D_2$. We can thus take:

$$X = \{(C_1; C_2, D_1; D_2) | C_1 \sim_\Gamma D_1, C_2 \sim_\Gamma D_2\}$$

If$_{\text{L}}$: C is of the form `if` (e) C_1 `else` C_2 and $C_{\text{L}} =$ `if` (e) D_1 `else` D_2, where $\Gamma \vdash_{\leq} e : \text{Bool}_{\text{L}}$, $\Gamma \vdash C_1 : D_1$ and $\Gamma \vdash C_2 : D_2$. By induction we have $C_1 \sim_\Gamma D_1$ and $C_2 \sim_\Gamma D_2$. Since the same branch will be taken by both C and C_{L} given that $E_1 =_\Gamma E_2$. We can thus choose $X = \{(C, C_{\text{L}})\}$

If$_{\text{H}}$: C is of the form `if` (e) C_1 `else` C_2 and $C_{\text{L}} =$ `skipIf` $e\ D_1$, where $\Gamma \vdash_{\leq} e : \text{Bool}_{\text{H}}$, $\Gamma \vdash C_1 : D_1$, $\Gamma \vdash C_2 : D_2$ and $D_1 \sim_\Gamma D_2$. Given E_1 and E_2, $E_1 =_\Gamma E_2$, we assume $\langle E_2 | C_{\text{L}}\rangle \xrightarrow{t_e \cdot t_{br}} \langle E_2 | D_1\rangle$. We then have $E_2 \vdash e \Downarrow l$ where $l \in \{\texttt{true}, \texttt{false}\}$ and from Lemma 10 that $E_1 \vdash e \Downarrow l'$, where also $l' \in \{\texttt{true}, \texttt{false}\}$. By semantic rule If we now have two cases:

If $\langle E_1 | C\rangle \xrightarrow{t_e \cdot t_{br}} \langle E_1 | C_1\rangle$ we can choose $X = \{(C, C_{\text{L}})\}$ since we have $C_1 \sim_\Gamma D_1$ by induction.

If $\langle E_1 | C\rangle \xrightarrow{t_e \cdot t_{br}} \langle E_1 | C_2\rangle$ we can also choose $X = \{(C, C_{\text{L}})\}$ since since we have $C_2 \sim_\Gamma D_2$ by induction, and $C_2 \sim_\Gamma D_1$ from Lemma 1.

Thus, we take $X = \{(C, C_{\text{L}})\}$.

SkipAsn: C is of the form `skipAsn` $le\ e$ and $C_{\text{L}} = C$. Choose $X = \{(C, C_{\text{L}})\}$.

SkipIf: C is of the form `skipIf` $e D$ and $C_{\text{L}} =$ `skipIf` $e D_{\text{L}}$ where $\Gamma \vdash D : D_{\text{L}}$. By induction we have $D \sim_\Gamma D_{\text{L}}$ and thus we can choose $X = \{(C, C_{\text{L}})\}$.

Let: C is of the form `let x :=` ie `in` D and $C_{\text{L}} =$ `let x :=` ie `in` D_{L}, where $\Gamma \vdash_{\leq} ie : \tau$ and $\Gamma, \texttt{x} : \tau \vdash D : D_{\text{L}}$. Assuming $E_1 =_\Gamma E_2$ and $\langle E_1 | C\rangle \xrightarrow{t_{pu\ ie}} \langle E_1 | \texttt{local x :=}\ v\ \texttt{in}\ D\rangle$, we have by Lemma 10 that $\langle E_2 | C_{\text{L}}\rangle \xrightarrow{t_{pu\ ie}} \langle E_2 | \texttt{local x :=}\ v'\ \texttt{in}\ D_{\text{L}}\rangle$, where v' differs from v only in the high parts according to τ. By induction, we have $D \sim_{\Gamma, \texttt{x}:\tau} D_{\text{L}}$, so we also have `local x :=` v `in` $D \sim_\Gamma$ `local x :=` v' `in` D_{L} and we can choose $X = \{(C, C_{\text{L}})\}$.

While: C is of the form `while` (e) D, and $C_{\text{L}} =$ `while` (e) D_{L}, where $\Gamma \vdash_{\leq} e : \text{Bool}_{\text{L}}$ and $\Gamma \vdash D : D_{\text{L}}$. By induction we have that $D \sim_\Gamma D_{\text{L}}$. With E_1 and E_2 such that $E_1 =_\Gamma E_2$, both C and C_{L} will either terminate or continue to loop, since e only depends on low variables. We have two cases:

If $\langle E_1 | C\rangle \xrightarrow{t_e \cdot t_{br} \cdot \checkmark} E_1$ then $\langle E_2 | C_{\text{L}}\rangle \xrightarrow{t_e \cdot t_{br} \cdot \checkmark} E_2$.

If $\langle E_1 | C\rangle \xrightarrow{t_e \cdot t_{br}} \langle E_1 | D; C\rangle$ then $\langle E_2 | C_{\text{L}}\rangle \xrightarrow{t_e \cdot t_{br}} \langle E_2 | D_{\text{L}}; C_{\text{L}}\rangle$

Choosing $X = \{(C, C_{\text{L}})\} \cup \{(D; C, D_{\text{L}}; C_{\text{L}}) | D \sim_\Gamma D_{\text{L}}\}$ we have that $X \subseteq F_\Gamma(X \cup \sim_\Gamma)$.

Output: C is of the form `output x` , and $C_{\text{L}} = C$. We can choose $X = \{(C, C_{\text{L}}), (\texttt{skip}, \texttt{skip})\}$, since $\Gamma(\texttt{x}) = \text{Int}_{\text{L}}$.