Improving language-based predictive mitigation for information-flow security

Jérémy Thibault[1] and Aslan Askarov[2]

[1] École Normale Supérieure de Rennes, France
[2] Aarhus University, Denmark

Abstract. Information-flow security analysis aims to prevent the leak of information in a system. Predictive mitigation tackles the issue of timing leaks in a general setting. We improve upon the performance and permissiveness of language-based predictive mitigation, a language-based instantiation of this technique. This work is carried in a setting of a simple imperative language with asynchronous I/O. More precisely, we leverage the asynchronous nature of the I/O to propagate mitigating delays to edges instead of delaying the whole computation.

1 Introduction

Ensuring segregation of private and public data is a key part of computer systems security. For many applications, such as cryptography, it is necessary that private information, such as cryptographic keys, or random seeds, can not be disclosed to an unauthorized observer.

Several techniques that disallow explicit access to private information exist: access control matrices [13], cryptography.

However, despite their efficiency, these models are not suited to prevent all information leaks. An observer having access to the public state of a program could deduce information on private data from this public state. It becomes apparent that there is a need to ensure no secret information can be retrieved this way. Information-flow analysis aims to prevent undesirable information flows from secret context to public context. Such flows can occur in several ways: explicitly (as in assignments or I/O), or implicitly, which includes covert channels (e.g. timing channels).

Predictive mitigation [3, 24] is a promising method to ensure no information flows via timing channels. It acts by delaying the timing of attacker-observable events.

In particular, language-based predictive mitigation [23] is an instantiation of this technique to a language-based setting that works by padding the execution time of secret-dependent sections in a given program.

However, predictive mitigation has the drawback on considerably slowing down the execution.

This work makes an attempt at solving this issue. We use asynchronous I/O to propagate the delays to I/O instead of delaying the whole execution. This

allows us to ignore some unnecessary delays in a multi-level setting. We also introduce "optimistic" mitigation that allows us to use less accurate predictions in a number of cases, which improves permissiveness.

2 Preliminaries

2.1 Attacker model

We assume that the attacker has access to the internal working of the system, typically the source code of the program. The attacker may be the provider of the program, for instance under the form of a plugin, or a library.

The attacker may observe events that are generated by the execution of the system. These events may be assignments to public variables or I/O operations for instance.

2.2 Security lattice

In this work, we try to prevent information flows from secret sources to public sinks.

We assume the information is classified according to a confidentiality level (or security level). The security levels form a hierarchy, where higher levels are more confidential. Moreover, we further restrict ourselves to security levels that form a lattice, proposed by Denning [7].

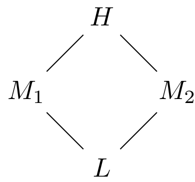

Fig. 1: A security lattice

In this representation, the security levels are ordered according to a relation $\sqsubseteq$ that reads "flows to". The "flows to" relation is a partial order on the set of security levels. For instance, the lattice described by Figure 1 represents the following relations: $L \sqsubseteq M_1$, $L \sqsubseteq M_2$, $M_1 \sqsubseteq H$, $M_2 \sqsubseteq H$, and by transitivity, $L \sqsubseteq H$.

The lattice representation also assume that there exists an unique least upper bound of a set of level. We note $\sqcup$ the binary operation that returns the least upper bound of two levels. In the previous example, we have for instance $M_1 \sqcup M_2 = H$, $L \sqcup M_1 = M_1$ and $H \sqcup M_1 = H$.

In the rest of the paper, we will use variables named $h, h_1, h_2, \ldots$ if they are of level H, $m, m_1, m_2, \ldots$ if they are of level M, etc.

2.3 Information flows

We consider a programming model, where our programs interact with outside parties over communication channels; the outside parties are also associated with security levels. For simplicity, we assume there is one channel per level.

The language consists of assignments, conditional branching, loops, and the `send` command which sends a value over a particular channel.

We are mainly interested in three kind of flows:

- The program `l := h` contains an *explicit flow*: the secret value of h is assigned to public variable l
- Figure 2a contains an *implicit flow* due to the structure of the program: the value of h is leaked into l, as one can be deduce only from reading the value of l which branch was taken
- Figure 2b contains an *implicit flow* due to a *timing channel*: if the observer L witnesses a short amount of time between the two messages, then the *else* branch had to be taken, hence $h = 0$; otherwise, $h = 1$.

```
if (h) {
  l := 1
} else {
  l := 0
}
```

(a) Implicit flow via control flow

```
send(L, 0)
if (h) {
  // Long computation
} else {
  skip
}
send(L, 1)
```

(b) Implicit flow via a timing channel

Fig. 2: Examples of undesirable flows

2.4 Noninterference

The principal concept of language-based information-flow security is the concept of *noninterference*. This concept has been introduced by Goguen in [9]. There are several variants of noninterference; see an overview by Hedin and Sabelfeld [10], and probabilistic noninterference [20].

Informally, we say that a program satisfies noninterference if given two initial states that differ only in their secret state (i.e. they are indistinguishable by the attacker), and two runs of the program starting from these initial states, the final states of the runs are likewise indistinguishable by the attacker.

2.5 Language-based information flow security

$$\frac{}{\Gamma \vdash n : l} \qquad \frac{\Gamma(x) = l}{\Gamma \vdash x : l} \qquad \frac{\Gamma \vdash e_1 : l_1 \qquad \Gamma \vdash e_2 : l_2}{\Gamma \vdash e_1 \star e_2 : l_1 \sqcup l_2}$$

$$\frac{}{\Gamma, pc \vdash \texttt{skip}} \qquad \frac{\Gamma \vdash e : l \qquad pc \sqcup l \sqsubseteq \Gamma(x)}{\Gamma, pc \vdash x := e} \qquad \frac{\Gamma, pc \vdash c_1 \qquad \Gamma, pc \vdash c_2}{\Gamma, pc \vdash c_1; c_2}$$

$$\frac{\Gamma \vdash e : l \qquad \Gamma, pc \sqcup l \vdash c_i, (i = 1, 2)}{\Gamma, pc \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2} \qquad \frac{\Gamma \vdash e : l \qquad \Gamma, pc \sqcup l \vdash c}{\Gamma, pc \vdash \texttt{while}\ e\ \texttt{do}\ c}$$

Fig. 4: Type system

$$
\begin{aligned}
e &::= n \mid x \mid e_1 \star e_2 \\
c &::= \texttt{skip} \\
&\quad \mid x := e \\
&\quad \mid c_1; c_2 \\
&\quad \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \\
&\quad \mid \texttt{while}\ e\ \texttt{do}\ c
\end{aligned}
$$

Fig. 3: Language grammar

For an overview of language-based information security, see the survey by Sabelfeld and Myers [19].

A first way of ensuring noninterference is static *Denning-style* security enforcement[8].

In programming languages implementing these techniques, a preemptive analysis of the program allows to reject code that contains explicit or implicit information flow. This is often done via type systems: each variable is associated a security type, that is the level of confidentiality of the variable; the type system ensures that a typable command can be executed without leaking information.

We consider as a basis for our work the language defined in Figure 3, and we use the formalization given by [2]. The semantics are given under the form of Structural Operational Semantics [18]; see Figure 14 in the appendix. The corresponding type system is described in Figure 4, based on the work of Volpano [21].

In this language, the semantics are given by a small-step relation $\langle c, m\rangle \rightarrow \langle c', m'\rangle$ where c is a command, m is a starting memory, m' is the updated memory and c' is the updated command, or $\texttt{stop}$ in the case of a terminal configuration. The rules for this relation are given Figure 14.

We can define noninterference for such a language, by defining memory agreement on public variables first:

Definition 1 (Memory agreement on public variables). *Given two memories m_1 and m_2, they agree on public variables, written $m_1 \sim m_2$, when*

$$\forall x, x \textit{ is public} \implies m_1(x) = m_2(x).$$

Definition 2 (Noninterference). *Program c is secure when for all memories m_1 and m_2 such that $m_1 \sim m_2$,*

$$\langle c, m_1\rangle \rightarrow^* \langle \texttt{stop}, m_1'\rangle \qquad \textit{and} \qquad \langle c, m_2\rangle \rightarrow^* \langle \texttt{stop}, m_2'\rangle$$

it holds that $m_1' \sim m_2'$.

This is a form of termination-insensitive definition, as it only consider runs where the execution terminates.

Noninterference can be ensured *via* security types. We suppose we have a security type environment Γ that maps variables to security level. The type system described in Figure 4 ensures noninterference. Typing rules for expression are of the form $\Gamma \vdash e : l$, which means expression e has security level l. Typing rules for commands are of the form $\Gamma, pc \vdash c$ where pc is the *program-counter level*, that carries information on the security context.

The purpose of the type system is two-fold. First, it ensures the absence of explicit flows in the form of assignment of high values to low variables. The

derivation rules ensure that the security context represent by the *pc* level is increased when entering a high-guarded section, and decreased upon leaving it. In such a section, assignments to public variables are then disallowed by the type system, to ensure the absence of control-flow based information leak.

However, this type system does not account for the timing behavior of the program. Indeed, a program such as the one described in Figure 2b is accepted by this type system. However, the measurement of the execution time of the program allows the attacker to deduce the private secret. Such an attack has been exhibited for some implementations of RSA [6, 11].

2.6 Predictive mitigation

One promising approach to timing leak is called *predictive mitigation* [3, 23]. Predictive mitigation tries to ensure observable events do not induce timing leaks by delaying them.

Language-based mitigation of timing channels [23] is a language-based approach to limit timing leaks. Timing mitigation is ensured by language construct that delays observable events.

This technique introduces a new command `pad`(e) `do` c. This executes the command c, then pads the execution so that e ticks are spent exactly in this section. The execution is stopped if the program spend too much time in it. This ensures that all possible runs of this command take the same amount of time, unless they timeout.

The technique has been shown to be a flexible, efficient way of limiting timing leaks [23, 3].

3 Contribution

Our contribution consists in an improvement to performance and permissiveness of language-based predictive mitigation.

3.1 Introduction of predictive mitigation

To improve the expressivity of the language described in Section 2.5, one may want to introduce a `send` operation that represents sending a message over an abstract channel, that can be a network channel, disk I/O, message passing between processes, etc.

To simplify the model, we assume that there exists one unique channel for each security level; extending the model to handle several channel per security level is immediate.

```
send (L, 0);
pad (50) {
  if (h) {
    // Long computation
  } else {
    skip;
  }
}
send (L, 1);
```

Fig. 5: Updated program with `pad` command

Timing channels in this augmented version However, this operation allows the receiver to observe timing channels, such as the one in Figure 2b. The observation of the timing of the call to `send` allows to deduce the value of a secret.

To ensure noninterference, the classical language-based technique is the introduction of a `pad` command whose role is to ensure that each the execution of a section while take the same amount of time, whatever the run.

The updated program is given Figure 5. In a run of this program, after executing the content of the padded section, the program will wait an appropriate amount of time.

Issues of classical predictive mitigation This addition to the language raise several issues.

First, if the execution follows a short path, a lot of CPU time is wasted waiting if the implementation is using busy waiting. Instead, if we allow another program to be executed, performances and security may be lost.

Indeed, context switching is not a free operation. Frequent context switch can severely hinder performance.

Moreover, interleaving of programs can lead to new security issues. Consider the program described in Figure 6. During a run of this program, if $h = 0$, then there is no waiting phase, and no possibility of interleaving another program. However, if $h = 1$, then it is possible to execute a program during the waiting phase that lasts 8 ticks. Then, after these 8 ticks, our program needs to resume instantly by the scheduler; indeed, if it is not the case, the message will be delayed and the observer will be able to deduce that another thread was executed. This means that the program had to wait in this section, hence $h = 1$.

```
pad(10) {
  if (h) {skip}
  else   {skip * 9}
}
send(0, L)
```

Fig. 6

Second, some messages to higher levels will be delayed for no reason by the predictive mitigation solution. Figure 7 is an example of such a case: suppose there is three security levels $L \sqsubseteq M \sqsubseteq H$. Then, the observer H has the right to access the value of variable m. However, predictive mitigation will still delay the execution of the $\texttt{send}(0, H)$ command even if it is not needed.

3.2 Optimized predictive mitigation

We propose a solution that solve these issues, but which requires asynchronicity of communication.

Model assumptions In our model, the attacker does not have access to the internal state of the program. Therefore, “pure” (in the sense that they do not contain communication commands) computations can not be observed by the

attacker; the only observation the attacker can make are those of messages they can receive. Then, it makes sens to only delay the timing of sent messages instead of the whole computation.

```
if (m) {
  // Long computation
} else {
  skip
}
send (0, H)
send (0, L)
```

Fig. 7

To handle this task, we suppose that there exists a runtime mechanism that acts as a buffer where message to be sent are placed, and send them at a later time.

Expected timing behavior Figure 8 represents the abstract utilization of a CPU during the execution of a program.When the processor is executing an instruction, the line is plain and when the processor has nothing to execute, it is dashed.

The first line describes the network communication thread, that must always be active in order to send messages at the right time.

The other two lines represent the utilization of the processor during the execution of the same program in both classical and optimized mitigation. Arrows represent the addition of messages to the message pool, and points from the time at which the `send` command is encountered to the time at which the message must be sent.

The time during which the processor is active is the same; however, the execution of the optimized version is not interrupted.

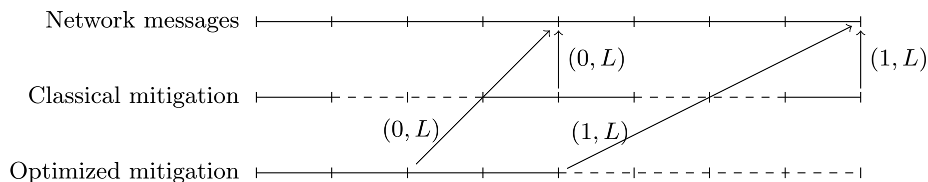

Fig. 8: Timing

3.3 Formalization

We consider the simple imperative language described in Section 2.5, augmented with operation allowing communication with external source, and a padding instruction used for predictive mitigation. The grammar of this language is given Figure 9.

The commands `pad` and `padr` are used for predictive mitigation: the first command is a source-level command, while the second is a runtime-only representation allowing to store information in case of nested paddings. The parameters of the first command are an expression e representing the allowed time for the execution, and a level l that describes the security level of the commands executed inside the padded section. The meaning of the runtime padding command is detailled later.

$$
\begin{aligned}
c ::= \ & \dots \\
& \mid \texttt{send}(l, e) \\
& \mid x \leftarrow \texttt{recv}(e, l) \\
& \mid \texttt{pad}(e, l) \ \texttt{do} \ c \\
& \mid \texttt{padr}(n_0, l, p, n_1, n_2) \ \texttt{do} \ c
\end{aligned}
$$

Fig. 9: Updated language

Representation of time We consider an abstract time represented by integers. Each instruction is executed in exactly one tick, with the exception of the `recv` instruction.

Representation of messages A program can send and receive information to/from a channel. Such information is described by a triplet (v, l, t) where v is the information being sent, l is the security level representing the channel, and t is the time at which the message was received or sent.

The messages that are received on a channel are determined by the previous history of messages sent and received on this channel, that is we suppose there exists a function s_l for each channel l that takes an history of messages (r, l) as an input and returns a value v.

Small-step relation The semantics are given by the small-step relation

$$\langle c, m, t, s, r, p\rangle \rightarrow \langle c', m', t', s', r', p'\rangle$$

where $\langle c, m, t, s, r, p\rangle$ and $\langle c', m', t', s', r', p'\rangle$ are *configurations*, that is a tuple $\langle c, m, t, s, r, p\rangle$ where:

- m is a memory: map from the variable names to $\mathbb{R}$.
- t is the current time ($t \in \mathbb{N}$)
- s is the set of sent messages, as described in the previous section. We do not require that if $(v, l, t') \in s$, then $t' \leq t$: a message that does not satisfy this is a message that will be sent in the future, as soon as the given time is reached
- r is the set of received messages
- p is a map of padding delays: map from the security levels to $\mathbb{N}$ which stores delay information regarding predictive mitigation. These delays correspond to the time that must be waited at the end of a padded section in classical predictive mitigation. These delays must be negative, which means the program is running late.

The updated rules for the basic language (without communication or predictive mitigation) are given in appendix Figure 15. The `stop`, sequential composition, assignment, conditions, and loops are the same as in a classical imperative language.

Communication and predictive mitigation semantics We introduce the element p from the configuration, that stores the time left at the end of padded sections, as well as the commands $\texttt{pad}(e, l)$ `do` c and $\texttt{padr}(v, l, p_0, v_0, t_0)$ `do` c.

The semantics of communication operations are given Figure 10a, and the semantics of mitigation are given Figure 10b.

$$\frac{\langle m, e\rangle \Downarrow v \qquad p(l) = t' \qquad s' = \{(l, v, t + t')\} \cup s \qquad p(l) \geq 0}{\langle \mathtt{send}(l, e), m, t, s, r, p\rangle \to \langle \mathtt{stop}, m, t + 1, s', r, p\rangle}$$

$$\frac{\begin{array}{c} s' = \{(l', v', t') \in s, l' = l\} \qquad r' = \{(l', v', t') \in r, l' = l\} \\ s_l(s', r') = v \qquad p(l) \geq 0 \qquad p' = l' \mapsto \begin{cases} 0 & \text{if } l \sqsubseteq l' \\ p(l') - p(l) & \text{otherwise} \end{cases} \end{array}}{\langle x \leftarrow \mathtt{recv}(l), m, t, s, r, p\rangle \to \langle \mathtt{stop}, m[x \leftarrow v], t + p(l) + 1, s, \{(l, v, t + p(l))\} \cup r, p'\rangle}$$

$$\frac{\begin{array}{c} s' = \{(l', v', t') \in s, l' = l\} \\ r' = \{(l', v', t') \in r, l' = l\} \qquad s_l(s', r') = v \qquad p(l) < 0 \end{array}}{\langle x \leftarrow \mathtt{recv}(l), m, t, s, r, p\rangle \to \langle \mathtt{stop}, m[x \leftarrow v], t + p(l) + 1, s, \{(l, v, t + p(l))\} \cup r, p\rangle}$$

(a) Semantics of communication operations

$$\frac{\langle m, e\rangle \Downarrow v}{\langle \mathtt{pad}(e, l) \ \mathtt{do} \ c, m, t, s, r, p\rangle \to \langle \mathtt{padr}(v, l, p, v, t + 1) \ \mathtt{do} \ c, m, t + 1, s, r, p\rangle}$$

$$\frac{\Delta t = t' - t \qquad v - \Delta t \geq 0 \qquad c' \neq \mathtt{stop} \qquad \langle c, m, t, s, r, p\rangle \to \langle c', m', t', s', r', p'\rangle}{\langle \mathtt{padr}(v, l, p_0, v_0, t_0) \ \mathtt{do} \ c, m, t, s, r, p\rangle \to \langle \mathtt{padr}(v - \Delta t, l, p_0, v_0, t_0) \ \mathtt{do} \ c', m', t', s, r', p'\rangle}$$

$$\frac{\begin{array}{c} \Delta t = t' - t \qquad v - \Delta t \geq 0 \qquad \langle c, m, t, s, r, p\rangle \to \langle \mathtt{stop}, m', t', s', r', p'\rangle \\ p'' = l' \mapsto \begin{cases} p_0(l') + v_0 - (t' - t_0) & \text{if } l \not\sqsubseteq l' \\ p'(l') & \text{otherwise} \end{cases} \end{array}}{\langle \mathtt{padr}(v, l, p_0, v_0, t_0) \ \mathtt{do} \ c, m, t, s, r, p\rangle \to \langle \mathtt{stop}, m', t', s', r', p''\rangle}$$

(b) Semantics of predictive mitigation

Fig. 10: Semantics of communication and predictive mitigation

These commands raise the security level to at least l, and executes the command c in this context.

We use a runtime-level `padr` command to ensure nested paddings are correctly handled. When entering a padded section via the command `pad`, the time the execution is allowed to spend in the section v_0, the current time t_0 and the current padding structure p_0 are stored in the runtime command. They are constant during the execution of the padded section. At the end of the padded section, these informations are used to construct the new padding structure in the following manner:

- if the section's level flows to another level, then this level must not be delayed further.
- otherwise, the value from p_0 is restored, and increased by the time differential.

When sending a message via the command $\mathtt{send}(e, l)$, the expression is evaluated immediately and the message is to be sent at later time $t + p(l)$, which is

exactly the time at which the message would have been sent had the execution waited at the end of the previous padded sections.

The reception of message from l is synchronous, as it waits for all message to l to be sent (that is it waits for $p(l)$ ticks), and then receive the message. The advantage of this is that it allows easier synchronization mechanism. Since the reception waits, we can decrease the values stored in p.

"Optimistic" mitigation Our semantics allow $p(l)$ to be negative. Since the only observable events are the messages, we allow the execution to continue even if it exceeds the time limit of a padding, and as long as no send operation is encountered. Then, the program can catch up on the delay when encountering another padded section.

If a `send` instruction is encountered while $p(l)$ is negative, this means that, to ensure timing security, one would have had to send the message before the instruction is encountered. Hence, we chose to stop the execution at this point; however, a particular implementation could allow sending the message late, allowing some information to be leaked.

We call this "optimistic" mitigation, as it allows more lax timing prediction: one could, via an appropriate mechanism, adapt dynamically the duration of a padded section that is found inside a loop.

Type system To ensure noninterference, we use the type system described in Figure 11.

$$\frac{}{\Gamma, pc \vdash \texttt{skip}} \qquad \frac{\Gamma \vdash e : l \qquad pc \sqcup l \sqsubseteq \Gamma(x)}{\Gamma, pc \vdash x := e} \qquad \frac{\Gamma, pc \vdash c_1 \qquad \Gamma, pc \vdash c_2}{\Gamma, pc \vdash c_1 ; c_2}$$

$$\frac{\Gamma \vdash e : l \qquad l \sqsubseteq pc \qquad \Gamma, pc \vdash c_i, (i = 1, 2)}{\Gamma, pc \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2} \qquad \frac{\Gamma \vdash e : l \qquad l \sqsubseteq pc \qquad \Gamma, pc \vdash c}{\Gamma, pc \vdash \texttt{while}\ e\ \texttt{do}\ c}$$

$$\frac{\Gamma \vdash e : l' \qquad pc \sqcup l' \sqsubseteq l}{\Gamma, pc \vdash \texttt{send}(e, l)} \qquad \frac{pc \sqcup l \sqsubseteq \Gamma(x) \qquad pc \sqsubseteq l \qquad \Gamma \vdash e : l' \qquad l' \sqsubseteq pc}{\Gamma, pc \vdash x \leftarrow \texttt{recv}(e, l)}$$

$$\frac{\Gamma \vdash e : l' \qquad l' \sqsubseteq pc \qquad \Gamma, pc \sqcup l \vdash c}{\Gamma, pc \vdash \texttt{pad}(e, l)\ \texttt{do}\ c} \qquad \frac{\Gamma, pc \vdash \texttt{pad}(v, l)\ \texttt{do}\ c}{\Gamma, pc \vdash \texttt{padr}(v, l, p_0, v_0, t_0)\ \texttt{do}\ c}$$

Fig. 11: Type system for optimized mitigation

This type system is similar to the classical type system described previously. The differences mostly lie in two points:

- Sending and receiving a message can be seen as slightly more complex assignments.
- Instead of the commands `if` and `while` raising the security context, it is now the padding command that raise pc.

Noninterference We prove a form of noninterference. In two different runs starting from initial configurations that are indistinguishable for an attacker l_{adv}, if both runs end, then the two final configuration are indistinguishable for the same attacker l_{adv}.

Definition 3 (Memory agreement at security level l). *Let $l \in \mathcal{L}$ be a security level. Two memories m_1 and m_2 agree at security level l, denoted $m_1 \sim_l m_2$, when:*

$$\forall l' \sqsubseteq l, \forall x, \Gamma(x) = l' \implies m_1(x) = m_2(x)$$

Definition 4 (Message agreement at security level l). *Let $l \in \mathcal{L}$ be a security level. Two sets of messages agree at security level l, denoted $s_1 \sim_l s_2$, when:*

$$\forall l' \sqsubseteq l, \forall v, \forall t, (v, l', t) \in s_1 \iff (v, l', t) \in s_2$$

This definition holds for both received and sent messages.

Definition 5 (Equivalence of commands). *Let l_{adv} be a security level.*
$c_1 \sim_{l_{adv}} c_2$ if and only if at least one of those is true:

- $c_1 = c_2$
- $c_1 = d_1; d'_1$, $c_2 = d_2; d'_2$, $d_1 \sim_{l_{adv}} d_2$, *and* $d'_1 = d'_2$
- $c_1 = \texttt{padr}(v, l, p_1, v_1, t_1)$ $\texttt{do}$ d_1, $c_2 = \texttt{padr}(v, l, p_2, v_2, t_2)$ $\texttt{do}$ d_2, $d_1 \sim_{l_{adv}} d_2$, *and for all* $l' \sqsubseteq l_{adv}$, $p_1(l) + t_1 + v_1 = p_2(l) + t_2 + v_2$

Definition 6 (Equivalence of configuration at level l_{adv}).
Let l_{adv} be a security level. Two configurations are equivalent at level l_{adv}, denoted:

$$\langle c_1, m_1, t_1, s_1, r_1, p_1 \rangle \sim_{l_{adv}} \langle c_2, m_2, t_2, s_2, r_2, p_2 \rangle$$

when

1. $c_1 \sim_{l_{adv}} c_2$
2. $m_1 \sim_{l_{adv}} m_2$
3. $s_1 \sim_{l_{adv}} s_2$
4. $r_1 \sim_{l_{adv}} r_2$
5. *for all* $l \sqsubseteq l_{adv}$, $p_1(l) + t_1 = p_2(l) + t_2$

Theorem 1 (Noninterference). *Let c be such that $\Gamma, pc \vdash c$. Then for all $\langle c, m_1, t_1, s_1, r_1, p_1 \rangle \sim_{l_{adv}} \langle c, m_2, t_2, s_2, r_2, p_2 \rangle$ such that:*

$$\langle c, m_1, t_1, s_1, r_1, p_1 \rangle \rightarrow^{n_1} \langle \texttt{stop}, m'_1, t'_1, s'_1, r'_1, p'_1 \rangle$$

and

$$\langle c, m_2, t_2, s_2, r_2, p_2 \rangle \rightarrow^{n_2} \langle \texttt{stop}, m'_2, t'_2, s'_2, r'_2, p'_2 \rangle$$

it holds that:

$$\langle \texttt{stop}, m'_1, t'_1, s'_1, r'_1, p'_1 \rangle \sim_{l_{adv}} \langle \texttt{stop}, m'_2, t'_2, s'_2, r'_2, p'_2 \rangle$$

This is a slightly weaker version of the previous noninterference definition, where the observable events are the message that are sent at all level below of equal to l_{adv}.

4 Evaluation

We implemented both classical predictive mitigation and our improvement for this simple language in OCaml.

The implementation abstracts away the communication system.

To improve our capacity to analyze this work, our implementation supports two kind of time model:

- First, the abstract time extracted from the semantics of the language.
- Then, a real-time model where time roughly corresponds to real execution times of the instructions.

We measured the timing of messages and the duration of the execution of two programs: the square-and-multiply modular exponentiation described in [14] and that is used in RSA, and a login system [5]. These programs are given in Figure 16 and Figure 17 (appendix).

As expected, the messages are sent at the same time in both version. Results (averaged over 1000 executions) regarding the execution time are given Figure 12, in milliseconds.

	Classical mitigation	Optimized mitigation	Improvement
Modular exponentiation	1.80	0.28	6.4×
Login	0.23	0.19	1.2×

Fig. 12: Execution time

This results are in line with our expectations. In particular, the modular exponentiation consists in a padded section inside a loop. The classical mitigation forces to wait each time the loop is entered, while the optimized do not, which explains these results.

We also evaluated the performances of a program that compute share values [1], for array size ranging from 1 to 200, averaged over 200 executions. The code is given Figure 18, and the results are given Figure 13

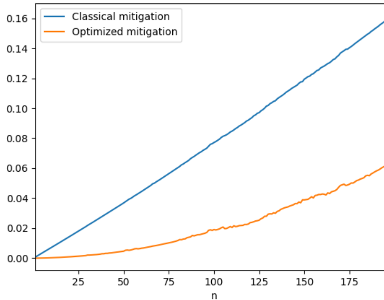

Fig. 13: Average duration of a run of the ShareValue program

As expected, the optimized version is faster than the classical version. In both version, the message are again sent at the same time.

Moreover, after a certain size, we notice that some padded sections begin to timeout. This cause the classical mitigation version to stop the execution, but is not an issue in the optimized version, as on average the timeout is not reached.

Further work This work can be extended in several manners.

– Provide a more realistic implementation of this work, including the runtime mechanism of semi-asynchronous model of I/O. This may turn out to be a difficult task, as it requires a fine management of time. This is an important step to provide usability.
– Extend the model to allow parallelism. Observational determinism [22] provides a security property for a parallel functional language. This language could be extended with predictive mitigation techniques.

5 Related work

This section discusses several solutions to information-flow security and in particular to timing channels from the literature, and compare them to our work.

Code transformation Code transformation eliminates some of the timing channels, by using techniques such as cross-copying [1] which pads branches with dummy statements, unification [12] which improves the previous technique by only inserting statements if they are needed, conditional assignment [15] which performs both computations, the result being encoded with bit masks and bitwise operations, and transactional branching [4], which wraps branches in transactions and commit only one of them, the other being aborted.

A study of their performance [14] shows that these transformations can significantly decrease the performance of programs. Contrary to our approach, these code transformations decrease the expressiveness of the language, as they disallow loops with secret guards.

Secure multi-execution Secure multi-execution consists in executing the program several times, once for every security level. Only outputs from their security level are kept, and inputs are replaced by default inputs in execution that is below their security level.

However, performance is affected by the need of multiple executions, especially in cases with a very high number of security levels.

Decentralized label model The Decrentalized Label Model [16] is a flexible model that allows the weakening of security policies by the owner of the information. This model is put to use in the language Jif [17], an extension to Java that provides static information-flow analysis.

6 Conclusion

Language-based predictive mitigation is an effective method to limit information-flow leaks. This method consists in delaying the execution of the program so that every run take the same time.

We proposed an optimization of this method, that leverages asynchronous I/O to move the delays from the computation to the inputs and outputs. More precisely, instead of delaying the whole execution of a program, only I/O operations are delayed.

We showed that a type system soundely prevents timing channels in this context. We also implemented this technique and showed that in several applications, its performance are indeed better.

References

[1] J. Agat. "Transforming Out Timing Leaks". In: *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*. 2000, pp. 40–53.

[2] A. Askarov. *Lecture notes for Language-based information-flow security*. 2017. URL: `https://github.com/aslanix/SmallStepNI/blob/master/infoflow-basics.pdf`.

[3] A. Askarov, D. Zhang, and A. C. Myers. "Predictive Black-box Mitigation of Timing Channels". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: ACM, 2010, pp. 297–307.

[4] G. Barthe, T. Rezk, and M. Warnier. "Preventing Timing Leaks Through Transactional Branching Instructions". In: *Electr. Notes Theor. Comput. Sci.* 153.2 (2006), pp. 33–55.

[5] A. Bortz and D. Boneh. "Exposing private information by timing web applications". In: *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*. 2007, pp. 621–628.

[6] D. Brumley and D. Boneh. "Remote Timing Attacks Are Practical". In: *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. 2003.

[7] D. E. Denning. "A Lattice Model of Secure Information Flow". In: *Commun. ACM* 19.5 (May 1976), pp. 236–243.

[8] D. E. Denning and P. J. Denning. "Certification of Programs for Secure Information Flow". In: *Commun. ACM* 20.7 (1977), pp. 504–513.

[9] J. A. Goguen and J. Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. 1982, pp. 11–20.

[10] D. Hedin and A. Sabelfeld. "A Perspective on Information-Flow Control". In: *Software Safety and Security - Tools for Analysis and Verification*. 2012, pp. 319–347.

[11] P. C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *Advances in Cryptology - CRYPTO ’96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. 1996, pp. 104–113.

[12] B. Köpf and H. Mantel. “Transformational typing and unification for automatically correcting insecure programs”. In: *Int. J. Inf. Sec.* 6.2-3 (2007), pp. 107–131.

[13] B. W. Lampson. “Protection”. In: *Operating Systems Review* 8.1 (1974), pp. 18–24.

[14] H. Mantel and A. Starostin. “Transforming Out Timing Leaks, More or Less”. In: *Proceedings, Part I, of the 20th European Symposium on Computer Security – ESORICS 2015 - Volume 9326*. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 447–467.

[15] D. Molnar et al. “The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks”. In: *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*. 2005, pp. 156–168.

[16] A. C. Myers and B. Liskov. “Protecting privacy using the decentralized label model”. In: *ACM Trans. Softw. Eng. Methodol.* 9.4 (2000), pp. 410–442.

[17] A. C. Myers et al. *Jif 3.0: Java information flow*. July 2006. URL: `http://www.cs.cornell.edu/jif`.

[18] G. D. Plotkin. “A structural approach to operational semantics”. In: (1981).

[19] A. Sabelfeld and A. C. Myers. “Language-based information-flow security”. In: 21.1 (Jan. 2003), pp. 5–19. URL: `http://www.cs.cornell.edu/andru/papers/jsac/sm-jsac03.pdf`.

[20] A. Sabelfeld and D. Sands. “Probabilistic Noninterference for Multi-Threaded Programs”. In: *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW ’00, Cambridge, England, UK, July 3-5, 2000*. 2000, pp. 200–214.

[21] D. M. Volpano, C. E. Irvine, and G. Smith. “A Sound Type System for Secure Flow Analysis”. In: *Journal of Computer Security* 4.2/3 (1996), pp. 167–188.

[22] S. Zdancewic and A. C. Myers. “Observational determinism for concurrent program security”. In: *16th IEEE Computer Security Foundations Workshop (CSFW)*. June 2003, pp. 29–43. URL: `http://www.cs.cornell.edu/andru/papers/csfw03.pdf`.

[23] D. Zhang, A. Askarov, and A. C. Myers. “Language-based Control and Mitigation of Timing Channels”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’12. Beijing, China: ACM, 2012, pp. 99–110.

[24] D. Zhang, A. Askarov, and A. C. Myers. “Predictive mitigation of timing channels in interactive systems”. In: *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*. 2011, pp. 563–574.

Appendix

6.1 Simple imperative language

$$\textsc{E-Const}\quad \frac{}{\langle m, n\rangle \Downarrow n} \qquad \textsc{E-Var}\quad \frac{}{\langle m, x\rangle \Downarrow m[x]} \qquad \textsc{E-Op}\quad \frac{\langle m, e_1\rangle \Downarrow n_1 \qquad \langle m, e_2\rangle \Downarrow n_2}{\langle m, e_1 \star e_2\rangle \Downarrow n_1 \star n_2}$$

(a) Semantics of expressions

$$\textsc{S-Skip}\quad \frac{}{\langle \texttt{skip}, m\rangle \to \langle \texttt{stop}, m\rangle} \qquad \textsc{S-Assign}\quad \frac{\langle m, e\rangle \Downarrow v}{\langle x := e, m\rangle \to \langle \texttt{skip}, m[x \leftarrow v]\rangle}$$

$$\textsc{S-Seq1}\quad \frac{\langle c_1, m\rangle \to \langle \texttt{stop}, m'\rangle}{\langle c_1; c_2, m\rangle \to \langle c_2, m'\rangle} \qquad \textsc{S-Seq2}\quad \frac{\langle c_1, m, \rangle \to \langle c_1', m'\rangle \qquad c_1' \neq \texttt{stop}}{\langle c_1; c_2, m, \rangle \to \langle c_1'; c_2, m'\rangle}$$

$$\textsc{S-If}\quad \frac{\langle m, e\rangle \Downarrow v \qquad v \neq 0 \implies i = 1 \qquad v = 0 \implies i = 2}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m\rangle \to \langle c_i, m\rangle}$$

$$\textsc{S-While-Cont}\quad \frac{\langle m, e\rangle \Downarrow v \qquad v \neq 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m\rangle \to \langle c; \texttt{while}\ e\ \texttt{do}\ c, m\rangle} \qquad \textsc{S-While-Break}\quad \frac{\langle m, e\rangle \Downarrow v \qquad v = 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m\rangle \to \langle \texttt{stop}, m\rangle}$$

(b) Semantics of commands

Fig. 14: Semantics for a simple imperative language

6.2 Optimized mitigation

$$\frac{}{\langle \texttt{skip}, m, t, s, r, p\rangle \to \langle \texttt{stop}, m, t+1, s, r, p\rangle}$$

$$\frac{\langle m, e\rangle \Downarrow v}{\langle x := e, m, t, s, r, p\rangle \to \langle \texttt{skip}, m[x \leftarrow v], t+1, s, r, p\rangle}$$

$$\frac{\langle c_1, m, t, s, r, p\rangle \to \langle \texttt{stop}, m', t', s', r', p'\rangle}{\langle c_1; c_2, m, t, s, r, p\rangle \to \langle c_2, m', t', s', r', p'\rangle}$$

$$\frac{\langle c_1, m, t, s, r, p\rangle \to \langle c_1', m', t', s', r', p'\rangle \qquad c_1' \neq \texttt{stop}}{\langle c_1; c_2, m, t, s, r, p\rangle \to \langle c_1'; c_2, m', t', s', r', p'\rangle}$$

$$\frac{\langle m, e\rangle \Downarrow v \qquad v \neq 0 \implies i = 1 \qquad v = 0 \implies i = 2}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m, t, s, r, p\rangle \to \langle c_i, m, t+1, s, r, p\rangle}$$

$$\frac{\langle m, e\rangle \Downarrow v \qquad v \neq 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m, t, s, r, p\rangle \to \langle c; \texttt{while}\ e\ \texttt{do}\ c, m, t+1, s, r, p\rangle}$$

$$\frac{\langle m, e\rangle \Downarrow v \qquad v = 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m, t, s, r, p\rangle \to \langle \texttt{stop}, m, t+1, s, r, p\rangle}$$

Fig. 15: Updated operational semantics for the base language without communication or predictive mitigation

6.3 Programs

The following programs are given in the syntax used by our implementation. In particular, the implementation does not allow `if` without an `else` branch, which explains why a `skip` is added to each of these branches. The operator `@` performs declassification, that is an expression `@`e is always public. Private variables begin by h, the other variables are public.

In the Square-and-multiply modular exponentiation Figure 16, informations about the private key h_k can be leaked by the timing behavior of the program.

In the login system Figure 17, an attacker can learn if the entered username exists in the database, because the comparison of the hash of the stored password and of the provided password takes time, here represented by 5 `skip`. For the experiments, we used the same predefined sequence of input values in all executions.

In the program ShareValue, Figure 18, the variable s represents the size of the array.

```
h_k := 65535;
n := 573;

y := 74;
h_r := 1;

i := 0;
while (i - 32) {
  pad (0.00005, H) {
    if ((h_k % 2) = 1) {
      h_r := (h_r * y) % n; skip;
    } else {
      skip;
    }
  }
  y := (y * y) % n;
  h_k := h_k / 2;
  i := i + 1;
}
h_res := h_r % n;

send(L, @h_res);
```

Fig. 16: Square-and-multiply modular exponentiation

```
h_username := 3;
h_pass := 5;

keep_looping := 1;
while (keep_looping) {

  u <- L;
  p <- L;

  pad (0.00003, H) {
    if (u = h_username) {
      skip; skip; skip; skip; skip;
      if (p = h_pass) {
        h_login := 1;
      } else {
        h_login := 0;
      }
    } else {
      h_login := 0;
    }
  }

  send (L, @h_login);

  if (u = 5) {
    keep_looping := 0;
  } else {
    skip;
  }
}
```

Fig. 17: Login system

```
s := 50

while (s - i) {
  h_id[i] <- H;
  h_id[i] := h_id[i] % 32;
  h_qty[i] <- H;
  h_qty[i] := h_id[i] % 32;
  i := i + 1;
}

h_shareVal := 0;
i := 0;

while (s - i) {
        pad(0.0007, H) {
                if (h_id[i] = h_special_share) {
                        h_shareVal := h_shareVal + (val * h_qty[i]); skip;
                } else { skip; }
        }
        i := i + 1;
}

send(H, h_shareVal);
```

Fig. 18: ShareValue program