Language-based predictive mitigation for systems with asynchronous I/O

Jérémy Thibault[1] and Aslan Askarov[2]

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

Abstract. Predictive mitigation is a general system technique for controlling timing channels that works by delaying timing of the attacker-observable system events in a deterministic manner. Language-based predictive mitigation is an instantiation of this technique to a language-based setting. This paper observes that in settings with asynchronous I/O it is possible to improve the performance and permissiveness of the language-based predictive mitigation by propagating mitigating delays to the I/O edges instead of delaying the whole computation, without losing soundness. This technique brings two advantages. First, we can avoid some of the otherwise unnecessary delays in a multi-level setting. Second, mitigating delays can be accumulated allowing for less accurate predictions.

1 Introduction

Ensuring that computer systems do not leak confidential information via their timing behavior remains an important challenge in computer security. The problem of timing channels is known to be dangerous in practice. Security literature contains numerous examples of sophisticated timing attacks demonstrating the feasibility of remote timing attacks [9], with recent examples showing how timing attacks can compromise user behavior [21, 23], and can be further exploited to leak contents of large databases [15]. Recent approaches show how effictive timing attacks be automatically synthesized given a source of an application [20]. Moreover, in systems permitting third-party code to access sensitive data, timing channels can be obviously be used to efficiently launder secrets.

In this work, we focus on the timing channels that have control flow representation, such as the one in Figure 1. Here, an attacker observing messages on channel `L` can infer which of the branches is taken based on the timing difference of the two messages. We do not consider runtime side channels that originate via hardware or runtime aspects of the system such as caches [6, 19], lazy evaluation [10, 22], or memory management [18]. Our approach is complementary to many existing techniques that focus on these side channels.

Most techniques for mitigating timing channels impose some form of predictable behavior on the timing of the secret-dependent computations. While this typically affects system performance, the trade-offs are justified by the resulting strong security guarantees.

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

Fig. 1: Example timing channel

In this paper, we examine one such technique, namely the language-based predictive mitigation of [24]. This technique requires the programmer to wrap secret-dependent computations with mitigating padding statements which enforces constant-time behavior and hence removes the possible timing leaks.

We observe that, in a programming model with asynchronous I/O the language-based predictive mitigation can be improved to avoid some of the unnecessary delays. Rather than delaying the whole computation we propose to continue the execution, but remember the amount of the delay and propagate the delay to the network runtime. We show how the delays need to be recorded in the program semantics and formally prove the soundness of our proposal. An added bonus of the new technique that it allows for the predictions to accumulate throughout the program execution, requiring less precise bounds from the programmer.

We evaluate our proposal using a prototype implementation demonstrating that it leads to efficient CPU utilization on a few examples without compromising security.

The rest of the paper is structured as follows. Section 2 presents our system model of a programming language with asynchronous I/O. Section 3 recalls the language-based predictive mitigation and describes how it may be improved under our attacker model. Section 4 formalizes our language. In particular, we show how a carefully set up semantics can address some of the shortcoming of the classical predictive mitigation. Section 5 proves the soundness of our approach, and Section 6 reports on the preliminary evaluation. We discuss the related work in Section 7, and conclude in Section 8.

2 System model

We investigate mitigation of timing channels in a setting of a simple imperative language extended with asynchronous I/O. The core of the mitigation is based on the ideas from [24].

The main deviation from the prior work is that our attacker model is weaker. Rather than assuming a co-resident adversary that is capable to observe the timing of the intermediate assignments, we consider a network-level attacker who only observes the I/O effects and their timing.

2.1 Security lattice

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 [11].

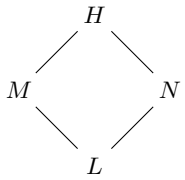

Fig. 2: A security lattice

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 2 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 $M \sqcup N = H$, $L \sqcup M = M$ and $H \sqcup M = H$.

In the rest of the paper, we 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.

2.2 The language

Figure 3 presents the syntax of our language. Expressions e range over integer constants, variables, and total arithmetic operations. Commands c include the standard imperative constructs, extended with two communication primitives $\texttt{send}(l, e)$ and $\texttt{recv}(e, l)$ and a nonstandard command for padding $\texttt{pad}(e, l)$ `do` c.

$$
\begin{aligned}
e &::= n \mid x \mid e_1 \star e_2 \\
c &::= \texttt{skip} \mid x := e \mid c_1; c_2 \mid \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } e \texttt{ do } c \\
&\quad \mid \texttt{send}(l, e) \mid x \leftarrow \texttt{recv}(l) \\
&\quad \mid \texttt{pad}(e, l) \texttt{ do } c
\end{aligned}
$$

Fig. 3: Language grammar

Communication in the language occurs over a set of channels. For simplicity, we conflate the channels with security levels l. The semantics of the send primitive is asynchronous; the semantics of receive is blocking. This setup allows us to explore the potential of the asynchronous communication without losing generality.

We explain the padding in Section 3.1 and present the formal semantics of the language in Section 4.1.

2.3 Attacker model

We consider a family of attackers, parametrized by their security levels. We assume that attackers have access to the internal working of the system, typically the source code, and observe the I/O interactions at their corresponding levels, including the timing of the communications. The attackers, however, cannot observe the order of the internal events in the program or communication at levels different from theirs. In particular, the attacker at level L can observe the I/O at the channel L but not on the channels with level l such that $L \not\sqsubseteq l$.

3 Mitigating timing leaks for asynchronous I/O

3.1 Language-based predictive mitigation

An effective way to control a timing channel as the one in Figure 1 is to ensure that the timing of the secret branch does not depend on the secret. This is the idea that is at the core of predictive mitigation [3, 25] that delays attacker-observable events in a systematic way. Language-based predictive mitigation [24] is an instantiation of this idea, where the delays can be provided as program expressions evaluated at runtime, which allows programming with fine-grained bounds. A security type system guarantees that the expressions specifying the bounds do not themselves reveal any information.

The parameters of the command `pad`(e, l) `do` c 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.

Figure 4 is a version of the example from Figure 1 rewritten with predictive mitigation. Here, the padding command ensures that the secret-dependent computation takes `50` units of time regardless of the value of the secret variable `h`. We refer to this delaying semantics of the padding command as *classical mitigation*.

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

Fig. 4: Updated program with `pad` command

3.2 A shortcoming of classical mitigation

While the classical mitigation padding soundly mitigates timing attacks, we observe that it also introduces unnecessary delays for high channels.

To see this, consider Figure 5. Suppose there are three security levels $L \sqsubseteq M \sqsubseteq H$. Then, the observer H can access the value of variable m. However, the padding also delays the execution of the `send`$(0, \texttt{H})$ command even though strictly speaking it is unnecessary. Only the low send needs to be delayed.

```
pad (50, L) {
  if (m) {
    // Long computation
  } else {
    skip
  }
}
send (0, H)
send (0, L)
```

Fig. 5: Example demonstrating limitations of the classical mitigation

3.3 Addressing the shortcoming

Our proposal to address this shortcoming is to propagate the delays from the main execution to the communication runtime. We achieve this through the following design.

- We assume that handling of network messages is handled in a parallel communication runtime process that does not interfere with the timing of the main execution.
- The padding command does not delay the execution at the end of its block. Instead, it immediately proceeds to the next command, remembering the amount of the delay that it would have otherwise spent at the end of the `pad` block.
- The accumulated delays are recorded in the semantics, and are propagated to the communication runtime. Sending of a message does not happen immediately. Instead, we record when in the future the message needs to be sent. This avoids unnecessary delays on higher channels, such as the `H`-communication in the earlier example. This is also a natural place for introducing additional delays, if the programming model assumes asynchronous semantics for sending.
- Because receive on a channel is blocking, we need to wait through the accumulated delay on the levels that are as restrictive as the level of the channel on which the message is received.

We refer to our proposed technique as optimized mitigation. We observe that this proposal does not require any changes to the syntax of the source language. The next section describes the semantics of the padding and the communication primitives in detail.

On the CPU utilization Note that the optimized technique can change the way the CPU is utilized. Consider program in Figure 6 that involves padding in a loop. Figure 7 presents the abstract utilization of a CPU during the execution of this program. Here, the solid lines correspond to the processor executing an instruction, and the dashed line correspond to the idling execution.

Three are three timelines in this figure. The first timeline depicts the communication process that is always active in order to send messages at the right time. The second and third timelines correspond to the utilization of the processor during the execution of the same program under both classical and the optimized mitigation strategies. The arrows represent pushing messages to the message pool, and connect the times of encountering the `send` commands in the program with the the times at which the messages must be sent.

While the overall time during which the processor is utilized (the sum of the solid lines) is the same, there are fewer interruptions in the execution of the optimized mitigation. We conjecture that fewer interruptions may lead to better CPU utilization, due to the reduced overhead of the OS-level context-switching and consequently better cache behavior.

```
i = 0;
while (i < 2) do {
  padTime = 3;
  pad (padTime, L) {
    skip; // placeholder for a high computation
  }
  send (i, L);
}
```

Fig. 6: A program with padding in a loop

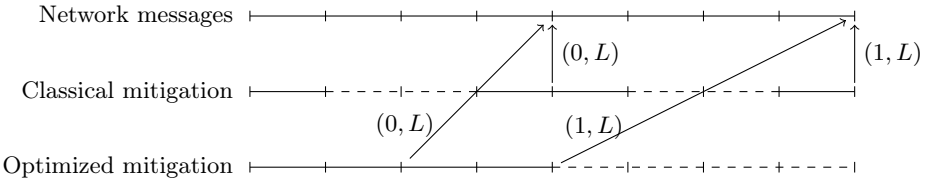

Fig. 7: Timing of the messages and the CPU utilization for example in Fig. 6

4 Formalization

This section defines the formal semantics for the language introduced in Section 2.2. We extend the grammar of the language with an auxiliary command $\texttt{padr}(n_0, l, p, n_1, n_2)\ \texttt{do}\ c$. This command does not appear in the source programs, but is used at runtime to store information about nested paddings.

$$\begin{aligned} c ::= {} & \ldots \\ & \mid \texttt{padr}(n_0, l, p, n_1, n_2)\ \texttt{do}\ c \end{aligned}$$

We explain the runtime padding command in detail in Section 4.1.

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 sent and received messages A program can send and receive information to/from a channel. Every message 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 an input strategy [17] function s_l for each channel l that takes the history of messages (s, r) on the channel as an input and returns a value v.

4.1 Semantics

The semantics of expressions is given by the big-step relation $\langle m, e\rangle \Downarrow n$ that specifies that expression e evaluates to value n in memory m. Here, m is a memory represented as a map from the variable names to values in $\mathbb{R}$.

The semantics of commands is 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*, which are tuples of the form $\langle c, m, t, s, r, p\rangle$ where:

- m is the memory
- t is the current time ($t \in \mathbb{N}$)
- s is the set of sent messages, as described in the previous subsection. 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. It maps security levels to $\mathbb{N}$, effectively storing delay information per level. These delays correspond to the time that must be spent idling at the end of a padded section in classical predictive mitigation. These delays may be negative, which means the program is running late.

Figures 8and 9 provides the transition rules for the basic language (without communication or predictive mitigation).

Communication and predictive mitigation semantics The semantics of communication operations are given in Figure 10a, and the semantics of mitigation are given in Figure 10b. These commands raise the security level to l, and execute the command c in this context.

$$\langle m, n\rangle \Downarrow n \qquad \langle m, x\rangle \Downarrow m[x] \qquad \frac{\langle m, e_1\rangle \Downarrow n_1 \qquad \langle m, e_2\rangle \Downarrow n_2}{\langle m, e_1 \star e_2\rangle \Downarrow n_1 \star n_2}$$

Fig. 8: Evaluation relation for expressions

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

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

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

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

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

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

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

Fig. 9: Operational semantics: the basic commands

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

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

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

(a) Semantics of communication operations

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

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

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

(b) Semantics of predictive mitigation

Fig. 10: Semantics of communication and predictive mitigation

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, this information is 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 $\texttt{send}(e, l)$, the expression is evaluated immediately and the message is set 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 messages to l to be sent (that is it waits for $p(l)$ ticks), and then proceeds to receive the message. The advantage of this is that it allows easier synchronization mechanism. Since receiving a message induces blocking, we decrease the values stored in p.

"Opportunistic" 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, as long as no low send operation is encountered. This gives the program an opportunity to catch up on the delay if another generous padded section is encountered before a low send.

If a `send` instruction is encountered while $p(l)$ is negative, this means that our padding values were incorrect. In our setup, for simplicity, we chose to stop the execution at this point. From the point of view of predictive mitigation, this corresponds to the prediction miss [3], and can be addressed by introducing penalty for future predictions.

4.2 Type system

Our enforcement is a standard Denning-style security type system. Figures 11 and 12 present the typing rules for expressions and commands, respectively. The typing judgment for commands is of the form $\Gamma \vdash e : l$. This judgment assigns security level l to expression e in a typing environment Γ that maps variable names to security levels. The typing judgement for commands has the form $\Gamma, pc \vdash c$. This judgment says that program c is well typed in environment Γ when the program counter label is at the level pc.

This type system eliminates standard direct and indirect flows. The non-standard aspect of this type system is that the program counter label can be raised explicitly only with the padding command. Finally, note also that the type system is termination-insensitive. Termination-insensitivity is further reflected in our noninterference condition in the next section.

5 Soundness

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}.

5.1 Low-equivalences

To formulate our noninterference theorem, we introduce a few technical definitions. Memory agreement at a level relates attacker's initial observations.

Definition 1 (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:*

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

$$\text{T-NAT}\ \frac{}{\Gamma \vdash n : l} \qquad \text{T-VAR}\ \frac{\Gamma(x) = l}{\Gamma \vdash x : l} \qquad \text{T-OP}\ \frac{\Gamma \vdash e_1 : l_1 \qquad \Gamma \vdash e_2 : l_2}{\Gamma \vdash e_1 \star e_2 : l_1 \sqcup l_2}$$

Fig. 11: Typing rules for expressions

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

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

$$\text{T-SEND}\ \frac{\Gamma \vdash e : l' \qquad pc \sqcup l' \sqsubseteq l}{\Gamma, pc \vdash \texttt{send}(e, l)} \qquad \text{T-RECV}\ \frac{pc \sqcup l \sqsubseteq \Gamma(x) \qquad pc \sqsubseteq l \qquad \Gamma \vdash e : l' \qquad l' \sqsubseteq pc}{\Gamma, pc \vdash x \leftarrow \texttt{recv}(l)}$$

$$\text{T-PAD}\ \frac{\Gamma \vdash e : l' \qquad l' \sqsubseteq pc \qquad \Gamma, pc \sqcup l \vdash c}{\Gamma, pc \vdash \texttt{pad}(e, l)\ \texttt{do}\ c} \qquad \text{T-PADR}\ \frac{\Gamma, pc \vdash \texttt{pad}(v, l)\ \texttt{do}\ c}{\Gamma, pc \vdash \texttt{padr}(v, l, p_0, v_0, t_0)\ \texttt{do}\ c}$$

Fig. 12: Typing rules for commands

Throughout execution, the attacker also obtains access to the messages that are sent and received on the attacker's channel. This is captured by the definition of *message agreement.*

Definition 2 (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:*

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

Note that we use definition for both received and sent messages.

We lift the definitions of equivalence from memories and messages to commands and configurations. For commands, we note that because of the nested paddings, their equivalence is slightly more complicated than simple syntactic matching.

Definition 3 (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 these conditions holds:*

- $c_1 = c_2$
- $c_1 = d_1; d_1'$, $c_2 = d_2; d_2'$, $d_1 \sim_{l_{adv}} d_2$, *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* $l' \sqsubseteq l_{adv}$, $p_1(l) + t_1 + v_1 = p_2(l) + t_2 + v_2$

The equivalence of configurations pulls all of the above definitions together.

Definition 4 (Equivalence of configuration at level $l_{\mathbf{adv}}$). *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* $l \sqsubseteq l_{adv}$, $p_1(l) + t_1 = p_2(l) + t_2$

The property 5 above is the key nonstandard ingredient of this definition, and is indeed at the heart of the noninterference proof. Two configurations are allowed to differ on their concrete timing as long as the accumulated delays count for the difference in their executions.

5.2 Noninterference

Our main technical result is that well-typed programs do not leak information via their timing behavior under our runtime. For simplicity, we formulate our top-level definition as a variant of termination-insensitive noninterference [4], noting that the proof technique in Appendix B fully supports a progress-insensitive condition. Termination-insensitivity is also a technical convenience for dealing with incorrect paddings, without having to introduce the complexity of penalizing predictions [24].

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:

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

Proof. The proof of this Theorem is in Appendix B

6 Initial evaluation

We have implemented both classical predictive mitigation and our improvement for a simple language as an interpreter in OCaml, abstracting the communication layer. While the results of this evaluation are preliminary, we believe they demonstrate that the proposed technique has a potential that warrants further studies.

Timing models To improve our capacity to analyze this work, our implementation supports two kind of time models: the abstract and the concrete. The abstract time is measured as the steps in the interpreter, and is extracted from the semantics of the language. The concrete time measures the actual time on the system, and roughly corresponds to the the real execution time of the instructions.

6.1 Modular exponentiation and Login

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 [8]. These programs are given in Figure 15 and Figure 16in Appendix A.

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 13, in milliseconds.

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

Fig. 13: Execution time for Modular exponentiation and Login examples

This results are in line with our expectations. In particular, because the modular exponentiation contains a padded section in loop, the optimized mitigation shows a considerable improvement. The classical mitigation forces to wait each time the loop is entered, while the optimized one does not, which explains these results.

6.2 Computation of a share's value

We also evaluated the performances of a program that compute share values [2], for array size ranging from 1 to 200, averaged over 200 executions. The code is given Figure 17 in Appendix A. Figure 14 depicts the relative performance of the classical and improved mitigation using real time.

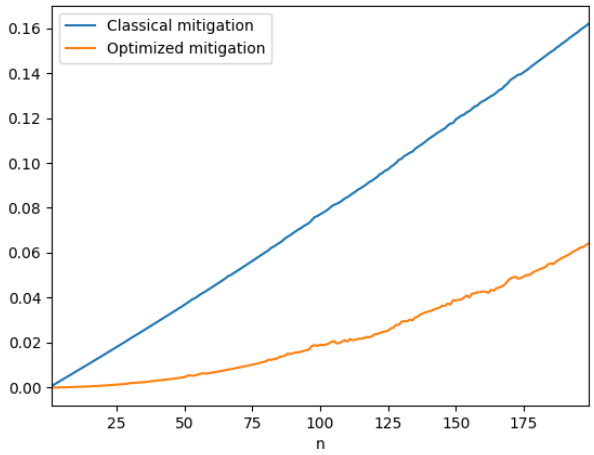

Fig. 14: Average duration of a run of the ShareValue program

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.

7 Related work

We briefly review the related work on mitigation of timing channels that have source-level representation.

Code transformation Code transformation eliminates some of the timing channels, by using techniques such as cross-copying [2] which pads branches with dummy statements, unification [13] which improves the previous technique by only inserting statements if they are needed, conditional assignment [16] which performs both computations, the result being encoded with bit masks and bitwise operations, and transactional branching [5], 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 [12] ensures timing by ensuring that outputs on low channels are produced from computations that never access the real secrets. The downside of the technique is the overhead in the number of runs, which increases with the size of the security lattice.

8 Conclusion

Language-based predictive mitigation is an effective method to control timing channels. We propose an optimization of this method, that leverages asynchronous I/O to move the delays from the computation to the inputs and outputs. We show how a combination of the static type system and a carefully crafted runtime semantics ensure a form of noninterference. Our initial evaluation suggests that this technique is promising and can yield significant performance improvements in certain cases.

Further work A prominent direction of future is extending our approach to reactive programming [7], and introduce the asynchronous version of the receive statement with callbacks. Other directions include efficient implementation of the runtime mechanism with more real-world evaluation.

References

[1] *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 2017. URL: `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8048777`.

[2] 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.

[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] A. Askarov et al. "Termination-Insensitive Noninterference Leaks More Than Just a Bit". In: *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*. Ed. by S. Jajodia and J. López. Vol. 5283. Lecture Notes in Computer Science. Springer, 2008, pp. 333–348. URL: `https://doi.org/10.1007/978-3-540-88313-5`.

[5] 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.

[6] D. J. Bernstein. *Cache-timing attacks on AES*. Tech. rep. 2005.

[7] A. Bohannon et al. "Reactive noninterference". In: *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*. Ed. by E. Al-Shaer, S. Jha, and A. D. Keromytis. ACM, 2009, pp. 79–90. URL: http://doi.acm.org/10.1145/1653662.1653673.

[8] 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.

[9] 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.

[10] P. Buiras and A. Russo. "Lazy Programs Leak Secrets". In: *Secure IT Systems - 18th Nordic Conference, NordSec 2013, Ilulissat, Greenland, October 18-21, 2013, Proceedings*. Ed. by H. R. Nielson and D. Gollmann. Vol. 8208. Lecture Notes in Computer Science. Springer, 2013, pp. 116–122. URL: https://doi.org/10.1007/978-3-642-41488-6_8.

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

[12] D. Devriese and F. Piessens. "Noninterference through Secure Multi-execution". In: *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010, pp. 109–124. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5504620.

[13] 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.

[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] H. Meer and M. Slaviero. "It's all about the timing..." In: *Black Hat USA*. 2007.

[16] 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.

[17] K. R. O'Neill, M. R. Clarkson, and S. Chong. "Information-Flow Security for Interactive Programs". In: *Proceedings of the 19th IEEE Computer Security Foundations Workshop*. Piscataway, NJ, USA: IEEE Press, July 2006, pp. 190–201.

[18] M. V. Pedersen and A. Askarov. "From Trash to Treasure: Timing-Sensitive Garbage Collection". In: *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 693–709. URL: https://doi.org/10.1109/SP.2017.64.

[19] C. Percival. *Cache missing for fun and profit*. 2005.

[20] Q. Phan et al. "Synthesis of Adaptive Side-Channel Attacks". In: *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 2017, pp. 328–342. URL: https://doi.org/10.1109/CSF.2017.8.

[21] I. Ray, N. Li, and C. Kruegel, eds. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*. ACM, 2015. URL: http://dl.acm.org/citation.cfm?id=2810103.

[22] M. Vassena, J. Breitner, and A. Russo. "Securing Concurrent Lazy Programs Against Information Leakage". In: *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 2017, pp. 37–52. URL: https://doi.org/10.1109/CSF.2017.39.

[23] P. Vila and B. Köpf. "Loophole: Timing Attacks on Shared Event Loops in Chrome". In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by E. Kirda and T. Ristenpart. USENIX Association, 2017, pp. 849–864. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/vila.

[24] 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.

[25] 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.

A Example programs used in the evaluation

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 15, informations about the private key h_k can be leaked by the timing behavior of the program.

In the login system Figure 16, 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

```
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. 15: Square-and-multiply modular exponentiation

the experiments, we used the same predefined sequence of input values in all executions.

In the program ShareValue, Figure 17, the variable s represents the size of the array.

```
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. 16: 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. 17: ShareValue program

B Soundness

Lemma 1 (noninterference for expressions). *Let Γ be a typing environment, l a security level, m_1 and m_2 two memories such that $m_1 \sim_l m_2$, and e an expression such that $\Gamma \vdash e : l'$ and $\langle e, m_1\rangle \Downarrow v_1$ and $\langle e, m_2\rangle \Downarrow v_2$.*

Then we have:

$$l' \sqsubseteq l \implies v_1 = v_2$$

Proof. By induction on the typing derivation $\Gamma \vdash e : l'$.

First rule: $v_1 = v_2 = n$

Second rule: $e = x$, where x is a variable. Suppose $l' \sqsubseteq l$. We have $\Gamma(x) = l'$, therefore by the definition of $\sim_l$, $m_1(x) = m_2(x)$. Hence $v_1 = v_2$.

Third rule: $e = e_1 \star e_2$ and $l' = l_1 \sqcup l_2$. Suppose $l' \sqsubseteq l$, then $l_1 \sqsubseteq l$ and $l_2 \sqsubseteq l$. Therefore by induction hypothesis applied to both l_1 and l_2, $v_1 = v_2$.

□

Lemma 2 (Transitivity of $\sim_l$). *$\sim_l$ is transitive.*

Proof. Immediate by definition of $\sim_l$. □

Lemma 3 (Public updates preserve $\sim_l$). *Let Γ be a typing environment, l a security level, m_1 and m_2 two memories such that $m_1 \sim_l m_2$ and x a variable such that $\Gamma(x) \sqsubseteq l$. Then for all values v, $m_1[x := v] \sim_l m_2[x := v]$*

Proof. Immediate by the definition of $\sim_l$. □

B.1 Well-formedness and preservation

Definition 5 (Well-formedness of configurations).

We say that a configuration $\langle c, m, t, s, r, p\rangle$ is well-formed w.r.t. a typing environment Γ and a level pc when either c is `stop`, or the program is well-typed, i.e. $\Gamma, pc \vdash c$.

Lemma 4 (Preservation of well-formedness).

Let Γ be a typing environment, pc a level, and $\langle c, m, t, s, r, p\rangle$ a configuration, such that the configuration is well-formed w.r.t. Γ and pc.

Supppose this configuration takes a step

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

Then the resulting configuration $\langle c', m', t', s', r', p'\rangle$ is also well-formed w.r.t. Γ and pc.

Proof. By induction on c.

Case $c =$ `skip`**:** Can only step to `stop`.

Case $c = x := e$**:** Can only step to `stop`.

Case $c = c_1; c_2$**:**
If $\langle c_1, m, t, s, r, p\rangle \to \langle \texttt{stop}, m', t', s', r', p'\rangle$ then $\langle c_1; c_2, m, t, s, r, p\rangle \to \langle c_2, m', t', s', r', p'\rangle$ which is well-formed because $c_1; c_2$ is well-formed.
Otherwise, there is c_1' such that $\langle c_1, m, t, s, r, p\rangle \to \langle c_1', m', t', s', r', p'\rangle$ and $\langle c_1; c_2, m, t, s, r, p\rangle \to \langle c_1'; c_2, m', t', s', r', p'\rangle$.
By induction hypothesis applied to $\langle c_1, m, t, s, r, p\rangle$, $\langle c_1', m', t', s', r', p'\rangle$ is well-formed, therefore $\langle c_1'; c_2, m', t', s', r', p'\rangle$ is well-formed as well.

Case $c = \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$**:** Can either step to c_1 or c_2, which are well-typed.

Case $c = \texttt{while}\ e\ \texttt{do}\ c'$**:** Can either step to `stop`, which gives a well-formed configuration, or to $c'; c$ which is well-typed.

Case $c = \texttt{send}(e, l)$**:** Can only step to `stop`

Case $c = \texttt{recv}(l)$**:** Can only step to `stop`

Case $c = \texttt{padr}(v, l, p_0)\ \texttt{do}\ d$**:** Can either step to `stop`, in which case the resulting is well-formed, or to another configuration of the form $\langle \texttt{padr}(v', l, p_0)\ \texttt{do}\ d', m', t', s', r', p'\rangle$.
Since the first configuration is well formed, it means that $\Gamma, pc \vdash \texttt{pad}(v, l)\ \texttt{do}\ c'$.
It is enough to prove that $\Gamma, pc \sqcup l \vdash d'$.
By applying the induction hypothesis to d, this is the case.

Case $c = \texttt{pad}(e, l)\ \texttt{do}\ c'$**:** This is true by the rule T-Padr.

□

B.2 Auxiliary semantics

We define an auxiliary semantics that represents an instrumentation of the original semantics with additional information. Because our noninterference is proven for all levels l_{adv}, our instrumentation is parameterized by the security level l_{adv}.

First, we define auxiliary semantics with events:

$$\alpha ::= \epsilon \mid A(x, v, l) \mid M_{\text{sent}}(v, l, t)$$

where x is the variable name, v is the value and l is the security level of the assignment. $A(x, v, l)$ represents an assignment, and $M_{\text{sent}}(v, l, t)$ represents a message sent. Then, we define the auxiliary semantics at level l_{adv} for each of the commands, as follows:

Assignment

S-Assign-Pub

$$\frac{\langle m, e\rangle \Downarrow v \qquad \Gamma(x) = l \qquad l \sqsubseteq l_{\text{adv}}}{\langle x := e, m, t, s, r, p\rangle \to_{A(x,v,l)} \langle \texttt{stop}, m[x \leftarrow v], t + 1, s, r, p\rangle}$$

S-Assign-Sec

$$\frac{\langle m, e\rangle \Downarrow v \qquad l \not\sqsubseteq l_{\text{adv}}}{\langle x := e, m, t, s, r, p\rangle \to_{\epsilon} \langle \texttt{stop}, m[x \leftarrow v], t + 1, s, r, p\rangle}$$

Sequence

$$\text{S-Seq1-Ev}\quad \frac{\langle c_1, m, t, s, r, p\rangle \to_\alpha \langle \texttt{stop}, m', t', s', r', p'\rangle}{\langle c_1; c_2, m, t, s, r, p\rangle \to_\alpha \langle c_2, m', t', s', r', p'\rangle}$$

$$\text{S-Seq2-Ev}\quad \frac{\langle c_1, m, t, s, r, p\rangle \to_\alpha \langle c_1', m', t', s', r', p'\rangle \qquad c_1' \neq \texttt{stop}}{\langle c_1; c_2, m, t, s, r, p\rangle \to_\alpha \langle c_1'; c_2, m', t', s', r', p'\rangle}$$

Recv

$$\text{S-Recv-Public}\quad \frac{\begin{array}{c} s' = \{(l', v', t') \in s, l' = l\} \qquad r' = \{(l', v', t') \in r, l' = l\} \qquad s_l(s', r') = v \\ n = p(l) \qquad p' = l' \mapsto \min(0, p(l') - n) \qquad \Gamma(x) = l' \qquad l' \sqsubseteq l_{\text{adv}} \end{array}}{\langle x \leftarrow \texttt{recv}(l), m, t, s, r, p\rangle \to_{A(x,v,\Gamma(x))} \langle \texttt{stop}, m[x \leftarrow v], t + n, s, \{(l, v, t + n)\} \cup r, p'\rangle}$$

$$\text{S-Recv-Sec}\quad \frac{\begin{array}{c} s' = \{(l', v', t') \in s, l' = l\} \qquad r' = \{(l', v', t') \in r, l' = l\} \qquad s_l(s', r') = v \\ n = p(l) \qquad p' = l' \mapsto \min(0, p(l') - n) \qquad \Gamma(x) = l' \qquad l' \not\sqsubseteq l_{\text{adv}} \end{array}}{\langle x \leftarrow \texttt{recv}(l), m, t, s, r, p\rangle \to_\epsilon \langle \texttt{stop}, m[x \leftarrow v], t + n, s, \{(l, v, t + n)\} \cup r, p'\rangle}$$

Send

$$\text{S-Send-Pub}\quad \frac{\langle m, e\rangle \Downarrow v \qquad p(l) = t' \qquad l \sqsubseteq l_{\text{adv}} \qquad s' = \{(l, v, t + t')\} \cup s}{\langle \texttt{send}(l, e), m, t, s, r, p\rangle \to_{M_{\text{sent}}(l,t,t+t')} \langle \texttt{stop}, m, t + 1, s', r, p\rangle}$$

$$\text{S-Send-Sec}\quad \frac{\langle m, e\rangle \Downarrow v \qquad p(l) = t' \qquad l \not\sqsubseteq l_{\text{adv}} \qquad s' = \{(l, v, t + t')\} \cup s}{\langle \texttt{send}(l, e), m, t, s, r, p\rangle \to_\epsilon \langle \texttt{stop}, m, t + 1, s', r, p\rangle}$$

Padding

$$\text{S-Pad1-Ev}\quad \frac{\begin{array}{c} \langle c, m, t, s, r, p\rangle \to_\alpha \langle c', m', t', s', r', p'\rangle \\ \langle \texttt{padr}(e, l)\ \texttt{do}\ c, m, t, s, r, p\rangle \to \langle \texttt{padr}(v - \Delta t)\ \texttt{do}\ c', m', t', s', r', p''\rangle \\ c' \neq \texttt{stop} \end{array}}{\langle \texttt{padr}(e, l)\ \texttt{do}\ c, m, t, s', r', p\rangle \to_\alpha \langle \texttt{padr}(v - \Delta t)\ \texttt{do}\ c', m', t', s', r', p''\rangle}$$

$$\text{S-Pad2-Ev}\quad \frac{\begin{array}{c} \langle c, m, t, s, r, p\rangle \to_\alpha \langle \texttt{stop}, m', t', s', r', p'\rangle \\ \langle \texttt{padr}(e, l)\ \texttt{do}\ c, m, t, s, r, p\rangle \to \langle \texttt{stop}, m', t', s', r', p''\rangle \end{array}}{\langle \texttt{padr}(e, l)\ \texttt{do}\ c, m, t, s, r, p\rangle \to_\alpha \langle \texttt{stop}, m', t', s', r', p''\rangle}$$

Lemma 5 (Adequacy of the semantics with events). *Let Γ be an environment, c a program and m a memory. Then $\langle c, m, t, s, r, p\rangle \to \langle c', m', t', s', r', p'\rangle$ if and only if there is an event α such that $\langle c, m, t, q, p\rangle \to_\alpha \langle c', m', t', q', p'\rangle$.*

Proof. By examining each case. □

Lemma 6 (Preservation of typing for auxiliary semantics). *Let Γ be an environment, pc a level, $\langle c, m, t, s, r, p\rangle$ a configuration well-formed w.r.t. Γ and pc. Suppose this configuration takes a step*

$$\langle c, m, t, s, r, p\rangle \rightarrow_\alpha \langle c', m', t', s', r', p'\rangle.$$

Then the resulting configuration $\langle c', m', t', s', r', p'\rangle$ is also well-formed w.r.t. Γ and pc.

Proof. Immediate from previous lemmas. □

B.3 Bridge relation and its properties

The now introduce our main working relation of the proof – the *bridge* relation between relations. Informally, two configurations are related by the bridge relation, when the second configuration is reachable from the first one with a non-empty event, or the second configuration is terminal.

Definition 6 (Bridge relation at level $l_{\mathbf{adv}}$). *Bridge relation $\langle c, m, t, s, r, p\rangle \curvearrowright^n_\alpha \langle c', m', t', s', r', p'\rangle$:*

$$\textsc{Bridge-stop}\quad \frac{\langle c, m, t, s, r, p\rangle \rightarrow_\epsilon \langle \texttt{stop}, m', t', s', r', p'\rangle}{\langle c, m, t, s, r, p\rangle \curvearrowright^0_\epsilon \langle \texttt{stop}, m', t', s', r', p'\rangle}$$

$$\textsc{Bridge-Ev}\quad \frac{\langle c, m, t, s, r, p\rangle \rightarrow_\alpha \langle c', m', t', s', r', p'\rangle \qquad \alpha \neq \epsilon}{\langle c, m, t, s, r, p\rangle \curvearrowright^0_\alpha \langle c', m', t', s', r', p'\rangle}$$

$$\textsc{Bridge-Multi}\quad \frac{c' \neq \texttt{stop} \qquad \langle c, m, t, s, r, p\rangle \rightarrow_\epsilon \langle c', m', t', s', r', p'\rangle \qquad \langle c', m', t', s', r', p'\rangle \curvearrowright^n_\alpha \langle c'', m'', t'', s'', r'', p''\rangle}{\langle c, m, t, s, r, p\rangle \curvearrowright^{n+1}_\alpha \langle c'', m'', t'', s'', r'', p''\rangle}$$

We now prove a number of technical lemmas about the properties of the bridge relation.

Lemma 7 (Bridge of sequential composition). *Let Γ be a typing environment, a sequential composition of two commands c_1 and c_2 such that $\langle c_1; c_2, m, t, s, r, p\rangle \curvearrowright^n_\alpha \langle c', m', t', s', r', p'\rangle$ then one of the following holds:*

1. *$n > 0$ and there are k, m'_1, s'_1, r'_1, p'_1 and t'_1 such that $k < n$,*

$$\langle c_1, m, t, s, r, p\rangle \curvearrowright^k_\epsilon \langle \texttt{stop}, m'_1, t'_1, s'_1, r'_1, p'_1\rangle$$

and

$$\langle c_2, m'_1, t'_1, s'_1, r'_1, p'_1\rangle \curvearrowright^{n-k-1}_\alpha \langle c', m', t', s', r', p'\rangle.$$

2. $\alpha \neq \epsilon$ *and there is* c_1' *such that* $\langle c_1, m, t, s, r, p\rangle \curvearrowright_\alpha^n \langle c_1', m', t', s', r', p'\rangle$ *and*

$$c' = \begin{cases} c_1'; c_2 & \text{if } c_1' \neq \texttt{stop} \\ c_2 & \text{otherwise} \end{cases}$$

Proof. Note that the bridge step can't be derived from Bridge-Stop, because the semantics forbid $c_1; c_2$ to step to `stop`. The proof proceeds by induction on n:

Base case $n = 0$**:**

Bridge-Ev: $\alpha \neq \epsilon$. We have $\langle c_1; c_2, m, t, s, r, p\rangle \to_\alpha \langle c', m', t', s', r', p'\rangle$. This can come from either:

1. S-Seq1-Ev. In this case there is $c_1' = \texttt{stop}$ such that $\langle c_1, m, t, s, r, p\rangle \to_\alpha \langle c_1', m', t', s', r', p'\rangle$.
   Hence $\langle c_1, m, t, s, r, p\rangle \curvearrowright_\alpha^0 \langle c_1', m', t', s', r', p'\rangle$ and $c' = c_2$
2. S-Seq2-Ev: there is $c_1' \neq \texttt{stop}$ such that $\langle c_1, m, t, s, r, p\rangle \to_\alpha \langle c_1', m', t', s', r', p'\rangle$.
   Hence $\langle c_1, m, t, s, r, p\rangle \curvearrowright_\alpha^0 \langle c_1', m', t', s', r'', p'\rangle$ and $c' = c_1'; c_2$.

Bridge-multi: Impossible: require $n > 0$

Inductive step Bridge-Ev: Impossible: require $n = 0$

Bridge-multi: There exists a configuration $\langle c'', m'', t'', s'', r'', p''\rangle$ such that $\langle c1; c2, m, t, s, r, p\rangle \to_\epsilon \langle c'', m'', t'', s'', r'', p''\rangle$ and $\langle c'', m'', t'', s'', r'', p''\rangle \curvearrowright_\alpha^{n-1} \langle c', m', t', s', r', p'\rangle$. There are two cases: either $c'' = c_1'; c_2$ or $c'' = c_2$. In the first case, by applying the induction hypothesis to the second bridge relation, we are in one the those two cases:

- $n-1 > 0$ and there are k and m_1', t_1', s_1', r_1' and p_1' such that $k < n-1$ and
$$\langle c_1', m'', t'', s'', r'', p''\rangle \curvearrowright_\epsilon^k \langle \texttt{stop}, m_1', t_1', s_1', r_1', p_1'\rangle$$
and
$$\langle c_2, m_1', t_1' sq_1', r_1', p_1'\rangle \curvearrowright_\alpha^{n-k-2} \langle c', m', t', s', r', p'\rangle.$$
- $\alpha \neq \epsilon$ and there is c_1'' such that $\langle c_1', m, t, s, r, p\rangle \curvearrowright_\alpha^{n-1} \langle c_1'', m', t', s', r', p'\rangle$ and
$$c' = \begin{cases} c_1'; c_2 & \text{if } c_1' \neq \texttt{stop} \\ c_2 & \text{otherwise} \end{cases}$$

In both case, by applying the rules of the bridge relation we obtain the result.
Otherwise, we are in the first case of the theorem with $k = 0$

□

Lemma 8 (Commands typed in secret are not making public assignment/messages). *Let* Γ *be a typing environment,* l *a security level such that* $l \not\sqsubseteq l_{adv}$ *and* c *a program such that* $\Gamma, l \vdash c$*, and two configurations such that* $\langle c, m, t, s, r, p\rangle \to_\alpha \langle c', m', t', s', r', p'\rangle$*. Then:*

1. $\alpha = \epsilon$
2. $m \sim_{l_{adv}} m'$
3. $s \sim_{l_{adv}} s'$

4. $r \sim_{l_{adv}} r'$

Proof. By induction on c:

Case $c = \texttt{skip}$**:** $\alpha = \epsilon$, and no assignments, messages or modification to paddings are made.

Case $c = x := e$**:** $\Gamma, l \vdash x := e$ therefore $l \sqsubseteq \Gamma(x)$. Therefore $\Gamma(x) \not\sqsubseteq l_{\text{adv}}$: the only applicable rule is S-Assign-Sec. Therefore $\alpha = \epsilon$.
$m' = m[x \leftarrow v]$ where $\langle m, e\rangle \Downarrow v$. Since $\Gamma(x) \not\sqsubseteq l_{\text{adv}}$, $m \sim_{l_{\text{adv}}} m'$. No modification to messages are made.

Case $c = x \leftarrow \texttt{recv}(l')$**:** $\Gamma, l \vdash x \leftarrow \texttt{recv}(l')$ therefore $l \sqsubseteq \Gamma(x)$. Therefore $\Gamma(x) \not\sqsubseteq l_{\text{adv}}$: the only applicable rule is S-Recv-Sec. Therefore $\alpha = \epsilon$, $m \sim_{l_{\text{adv}}} m'$ and $r \sim_{l_{\text{adv}}} r'$.

Case $c = \texttt{send}(e, l')$**:** $l \sqsubseteq l'$ therefore $l' \not\sqsubseteq l_{\text{adv}}$. The only applicable rule is S-Send-Secret, hence $\alpha = \epsilon$. Since $l' \not\sqsubseteq l_{\text{adv}}$, $q \sim_{l_{\text{adv}}} q'$, and no modifications to messages are made.

Other cases: by applying the induction hypothesis

□

Lemma 9 (Bridge of padded commands). *Let* $\texttt{padr}(v, l, p_0, v_0, t_0)\ \texttt{do}\ c$ *be a well-typed command, and* $\langle \texttt{padr}(v, l, p_0, v_0, t_0)\ \texttt{do}\ c, m, t, s, r, p\rangle$ *a configuration. Suppose that*

$$\langle \texttt{padr}(v, l, p_0, v_0, t_0)\ \texttt{do}\ c, m, t, s, r, p\rangle \curvearrowright^n_\alpha \langle d, m', t', s', r', p'\rangle.$$

Then we are in one of these two cases:

1. $d = \texttt{stop}$ *and there exist* p'' *such that* $p' = l' \mapsto \begin{cases} p_0(l') + v_0 - (t' - t_0) & \text{if } l \not\sqsubseteq l' \\ p''(l') & \text{otherwise} \end{cases}$
and
$$\langle c, m, t, s, r, p\rangle \curvearrowright^n_\alpha \langle \texttt{stop}, m', t', s', r', p''\rangle.$$
2. *There exist* v' *and* c' *such that* $d = \texttt{padr}(v', l, p_0, v_0, t_0)\ \texttt{do}\ c'$ *and*
$$\langle c, m, t, s, r, p\rangle \curvearrowright^n_\alpha \langle c', m', t', s', r', p'\rangle.$$

Proof. By induction on the bridge relation.

Bridge-stop: This means $n = 0$, $\alpha = \epsilon$, and $d = \texttt{stop}$.
Then $\langle \texttt{padr}(v, l, p_0, v_0, t_0)\ \texttt{do}\ c, m, t, s, r, p\rangle \to_\epsilon \langle \texttt{stop}, m', t', s', r', p'\rangle$. Therefore, the only applicable rule is S-Pad-Stop. Hence, there is p'' such that $\langle c, m, t, s, r, p\rangle \to_\epsilon \langle \texttt{stop}, m', t', s', r', p''\rangle$ and that verifies the above property. Therefore, by the rule Bridge-Stop,

$$\langle c, m, t, s, r, p\rangle \curvearrowright^0_\epsilon \langle \texttt{stop}, m', t', s', r', p'\rangle.$$

Bridge-ev: This means $n = 0$ and $\alpha \neq \epsilon$.
Then $\langle \texttt{padr}(v, l, p_0, v_0, t_0)\ \texttt{do}\ c, m, t, s, r, p\rangle \to_\alpha \langle d, m', t', s', r', p'\rangle$. Therefore, the only applicable rule are S-Pad-Stop and S-Pad-Cont.

S-Pad-Stop: There is p'' such that $\langle c, m, t, s, r, p\rangle \to_\alpha \langle \texttt{stop}, m', t', s', r', p''\rangle$ and that verifies the above property. This means that $d = \texttt{stop}$. Therefore, via the rule Bridge-Ev,

$$\langle c, m, t, s, r, p\rangle \curvearrowright_\alpha^0 \langle \texttt{stop}, m', t', s', r', p'\rangle$$

S-Pad-Cont Then there is c' such that $\langle c, m, t, s, r, p\rangle \to_\alpha \langle c', m', t', s', r', p'\rangle$. Therefore, $d = \texttt{padr}(v', l, p_0, v_0, t_0)\ \texttt{do}\ c'$ with v' a natural number, and

$$\langle c, m, t, s, r, p\rangle \curvearrowright_\alpha^0 \langle c', m', t', s', r', p'\rangle.$$

Bridge-Multi: Then $n > 0$. There is $\langle \texttt{padr}(e, l, p_0)\ \texttt{do}\ c, m, t, s, r, p\rangle \to_\epsilon \langle c'', m'', t'', s'', r'', p''\rangle$ with $c'' \neq \texttt{stop}$ and such that

$$\langle c'', m'', t'', s'', r'', p''\rangle \curvearrowright_\alpha^{n-1} \langle c', m', t', s', r', p'\rangle.$$

But $c'' = \texttt{padr}(v', l, p_0, v_0, t_0)\ \texttt{do}\ c^{(3)}$ for some $c^{(3)}$ and v'. By applying the induction hypothesis to this relation, we obtain the result.

□

B.4 Noninterference for bridge relation

Using these technical lemmas, we can now formulate our noninterference for bridge relation.

Theorem 2 (Noninterference for bridge). *Let Γ be a typing environment, and $\langle c_1, m_1, t_1, s_1, r_1, p_1\rangle \sim_{l_{adv}} \langle c_2, m_2, t_2, s_2, r_2, p_2\rangle$ two configurations such that $\Gamma, pc \vdash c$.*

Suppose there exist α_1 and α_2 two events, n and n_2 two natural numbers and two configurations $\langle c_1', m_1', t_1', s_1', r_1', p_1'\rangle$ and $\langle c_2', m_2', t_2', s_2', r_2', p_2'\rangle$ such that:

$$\langle c_1, m_1, t, s_1, r_1, p\rangle \curvearrowright_{\alpha_1}^{n} \langle c_1', m_1', t_1', s_1', r_1', p_1'\rangle$$

and

$$\langle c_2, m_2, t, s_2, r_2, p\rangle \curvearrowright_{\alpha_2}^{n_2} \langle c_2', m_2', t_2', s_2', r_1', p_2'\rangle.$$

Then it must be that:

1. $\langle c_1', m_1', t_1', s_1', r_1', p_1'\rangle \sim_{l_{adv}} \langle c_2', m_2', t_2', s_2', r_2', p_2'\rangle$
2. $\alpha_1 \neq \epsilon$ *if and only if* $\alpha_2 \neq epsilon$
3. *If* $\alpha_1 = A(x, v, l')$, *then* $\alpha_2 = A(x, v, l')$
4. *If* $\alpha_1 = M_{sent}(v, l', t)$ *then* $\alpha_2 = M_{sent}(v, l', t)$

Proof. Induction on the number of steps n:

Base case $n = 0$: Induction on c_1.

Case $c_1 = \texttt{skip}$**:** The only possibility is

$$\langle \texttt{skip}, m_1, t_1, s_1, r_1, p_1\rangle \to_\epsilon \langle \texttt{stop}, m_1, t_1 + 1, s_1, r_1, p_1\rangle$$

for the first run and

$$\langle \texttt{skip}, m_2, t_2, s_2, r_2, p_2\rangle \to_\epsilon \langle \texttt{stop}, m_2, t_2 + 1, s_2, r_2, p_2\rangle$$

for the second run.
Therefore, we have $c_1' = c_2' = \texttt{stop}$, $\alpha_1 = \epsilon = \alpha_2$ and due to the hypothesis: $m_1' = m_1 \sim_{l_{\text{adv}}} m_2 = m_2'$, $r_1' = r_1 \sim_{l_{\text{adv}}} r_2 = r_2'$, $s_1' = s_1 \sim_{l_{\text{adv}}} s_2 = s_2'$, and

$$\begin{aligned}\forall l \sqsubseteq l_{\text{adv}},\ p_1'(l) + t_1' &= p_1(l) + t_1 + 1\\ &= p_2(l) + t_2 + 1 \text{ by hypothesis}\\ &= p_2'(l) + t_2'\end{aligned}$$

Case $c_1 = x := e$**:** Then $c_2 = c_1$.
We examine $\langle x := e, m_1, t_1, s_1, r_1, p_1\rangle \curvearrowright^0_{\alpha_1} \langle c_1', m_1', t_1', s_1', r_1', p_1'\rangle$.
Rule Bridge-Multi: not applicable because n = 0
Rule Bridge-Stop: Then $\alpha_1 = \epsilon$, and $\langle x := e, m_1, t_1, s_1, r_1, p_1\rangle \to_\epsilon \langle \texttt{stop}, m_1', t_1', s_1', r_1', p_1'\rangle$. This is only possible by the rule S-Assign-Sec, therefore we have $\Gamma(x) \not\sqsubseteq l_{\text{adv}}$.
Now, we observe that the bridge transition for the second run must also be produced by an assignment to secret variable x (via the same rules). This gives us $c_2' = \texttt{stop}$ and $\alpha_2 = \epsilon$.
Since $\Gamma(x) \not\sqsubseteq l_{\text{adv}}$, the assignment to variable x does not change memory equivalence at level l_{adv}, hence $m_1 \sim_{l_{\text{adv}}} m_1'$. For the same reason, $m_2 \sim_{l_{\text{adv}}} m_2'$. By transitivity, and hypothesis, $m_1' \sim_{l_{\text{adv}}} m_2'$.
We have:

- $c_1' = c_2' = \texttt{stop}$
- $\alpha_1 = \alpha_2 = \epsilon$
- $m_1' \sim_{l_{\text{adv}}} m_2'$
- $s_1' = s_1$ and $s_2' = s_2$ thus $s_1' \sim_{l_{\text{adv}}} s_2'$
- $r_1' = r_1$ and $r_2' = r_2$ thus $r_1' \sim_{l_{\text{adv}}} r_2'$
- $t_1' = t_1 + 1$ and $t_2' = t_2 + 1$. Since $p_1' = p_1$ and $p_2' = p_2$, for all $l \sqsubseteq l_{\text{adv}}$,

$$\begin{aligned}p_1'(l) + t_1' &= p_1(l) + t_1 + 1\\ &= p_2(l) + t_2 + 1 \text{ by hypothesis}\\ &= p_2'(l) + t_2'.\end{aligned}$$

Rule Bridge-Ev: Then $\alpha_1 \neq \epsilon$. This is only possible via the rule S-Assign-Pub. Therefore $\Gamma(x) \sqsubseteq l_{\text{adv}}$.
Now, we observe that the bridge transition for the second run must also be produced by an assignment to public variable x (via the same rules).

The program is well-typed, therefore there exists l such that $\Gamma \vdash e : l$ and $pc \sqcup l \sqsubseteq \Gamma(x)$. Therefore, $l \sqsubseteq l_{\text{adv}}$. By the Lemma 1 of noninterference for expression, $v_1 = v_2$ where $\langle m_1, e\rangle \Downarrow v_1$ and $\langle m_2, e\rangle \Downarrow v_2$.
Hence, since public updates preserve memory equivalence, we have that $m_1' \sim_{l_{\text{adv}}} m_2'$.

- $c_1' = c_2' = \texttt{stop}$
- $\alpha_1 = \alpha_2 = A(x, v_1, \Gamma(x))$
- $m_1' \sim_{l_{\text{adv}}} m_2'$
- $s_1' = s_1$ and $s_2' = s_2$ thus $s_1' \sim l_{\text{adv}} s_2'$
- $r_1' = r_1$ and $r_2' = r_2$ thus $r_1' \sim l_{\text{adv}} r_2'$
- $t_1' = t_1 + 1$ and $t_2' = t_2 + 1$. Since $p_1' = p_1$ and $p_2' = p_2$, for all $l \sqsubseteq l_{\text{adv}}$,

$$\begin{aligned} p_1'(l) + t_1' &= p_1(l) + t_1 + 1 \\ &= p_2(l) + t_2 + 1 \\ &= p_2'(l) + t_2'. \end{aligned}$$

Case $c_1 = d_{1,1}; d_{2,1}$**:** Then $c_2 = d_{1,2}; d_{2,2}$ with $d_{1,1} \sim_{l_{\text{adv}}} d_{1,2}$ and $d_{2,1} \sim_{l_{\text{adv}}} d_{2,2}$
We apply the lemma to the first bridge relation. We can only be in the second case, since $n = 0$. There is d_1' such that $\langle d_{1,1}, m_1, t_1, s_1, r_1, p_1\rangle \curvearrowright_\alpha^0 \langle d_1', m_1', t_1', s_1', r_1', p_1'\rangle$ and $c_1' = \begin{cases} d_1'; d_{2,1} & \text{if } d_1' \neq \texttt{stop} \\ d_{2,1} & \text{otherwise} \end{cases}$.
By applying the same lemma to the second bridge relation we have two possibilities:

1. $n_2 > 0$ and there are k, m_2'', t_2'', s_2'', r_2'' and p_2'' such that $k < n_2$ and

$$\langle d_{2,1}, m_2, t_2, s_2, r_2, p_2\rangle \curvearrowright_\epsilon^k \langle \texttt{stop}, m_2'', t_2'', s_2'', r_2'', p_2''\rangle$$

   and $\langle d_{2,2}, m_2'', t_2'', s_2'', r_2'', p_2''\rangle \curvearrowright_{\alpha_2}^{n_2-k-1} \langle c_2', m_2', t_2', s_2', r_2', p_2'\rangle$.
   Applying the inner induction hypothesis on this two equations for $d_{1,1}$ and $d_{1,2}$ gives us that $\alpha_1 = \epsilon$. This case is impossible.
2. $\alpha_2 \neq \epsilon$ and there is d_1'' such that $\langle d_{1,2}, m_2, t_2, s_2, r_2, p_2\rangle \curvearrowright_{\alpha_2}^{n_2} \langle d_1'', m_2', t_2', s_2', r_2', p_2'\rangle$ and $c_2' = \begin{cases} d_1''; d_{2,2} & \text{if } d_1'' \neq \texttt{stop} \\ d_{2,2} & \text{otherwise} \end{cases}$.
   By applying the inner hypothesis induction to $d_{1,1}$ and $d_{1,2}$ we obtain that $\langle d_1', m_1', t_1', s_1', r_1', p_1'\rangle \sim_{l_{\text{adv}}} \langle d_1'', m_2', t_2', s_2', r_2', p_2'\rangle$. Hence this is also the case for the two configurations resulting from c_1 and c_2.

Case $c_1 = \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2$**:** Not applicable because $n = 0$

Case $c_1 = \texttt{while } e \texttt{ do } c'$**:** Not applicable because $n = 0$

Case $c_1 = \texttt{send}(l, e)$**:** Then $c_1 = c_2 = \texttt{send}(l, e)$.
We examine $\langle \texttt{send}(e, l), m_1, t_1, s_1, r_1, p_1\rangle \curvearrowright_{\alpha_1}^0 \langle c_1', m_1', t_1', s_1', r_1', p_1'\rangle$.
Rule Bridge-Multi: not applicable because n = 0

Rule Bridge-Stop: Then $\alpha_1 = \epsilon$ and $\langle \texttt{send}(e, l), m_1, t_1, s_1, r_1, p_1\rangle \to_\epsilon \langle c_1', m_1', t_1', s_1', r_1', p_1'\rangle$.
This is only possible by applying the rule S-Send-Secret, therefore we have $l \not\sqsubseteq l_{\text{adv}}$. Therefore, the addition of the message to the set does not change message sent equivalence: $s_1 \sim_{l_{\text{adv}}} s_1'$.
Now, we observe that the bridge transition for the second run must also be produced by a message sent at level l. Thus, $s_2 \sim_{l_{\text{adv}}} s_2'$. By transitivity, $s_1' \sim_{l_{\text{adv}}} s_2'$. Both final commands are `stop`, and in both cases the message received queues and the memory are not modified, so their equivalence holds.
Last, $t_1' = t_1 + 1$ and $t_2' = t_2 + 1$. Paddings are not modified therefore $p_1' + t_1' = p_2' + t_2'$.
The final configurations are low-equivalent at security level l_{adv}.

Rule Bridge-Ev: Then $\langle \texttt{send}(e, l), m_1, t_1, s_1, r_1, p_1\rangle \to_{\alpha_1} \langle c_1', m_1', t_1', s_1', r_1', p_1'\rangle$ and $\alpha_1 \neq \epsilon$. The only applicable rule is S-Send-Pub, therefore $l \sqsubseteq l_{\text{adv}}$.
Now we observe that the bridge transition for the second run must also be produced by a message sent to the same public security level.
The program is well-typed, therefore there exists l' such that $\Gamma \vdash e : l'$ and $pc \sqcup l' \sqsubseteq l \sqsubseteq l_{\text{adv}}$, therefore $l' \sqsubseteq l_{\text{adv}}$.
By the lemma of noninterference for expressions, $v_1 = v_2$ where $\langle m_1, e\rangle \Downarrow v_1$ and $\langle m_2, e\rangle \Downarrow v_2$.
Hence, we have $s_1' \sim_{l_{\text{adv}}} s_2'$.
Both final commands are `stop`, the memories and the received messages set are not modified.
Last, exactly as before, $t_1' = t_1 + 1$ and $t_2' = t_2 + 1$.
The final configurations are low-equivalent at security level l_{adv} and the events α_1 and α_2 are equal.

Case $c = x \leftarrow \texttt{recv}(l)$**:**
We examine the rule $\langle x \leftarrow \texttt{recv}(l), m_1, t_1, s_1, r_1, p_1\rangle \curvearrowright^0_{\alpha_1} \langle c_1', m_1', t_1', s_1', r_1', p_1'\rangle$.
Rule Bridge-Multi: not applicable because $n = 0$.
Rule Bridge-Stop:
Then $c_1' = \texttt{stop}$, $\alpha_1 = \epsilon$ and

$$\langle x \leftarrow \texttt{recv}(l), m_1, t_1, s_1, r_1, p_1\rangle \to_\epsilon \langle \texttt{stop}, m_1', t_1', s_1', r_1', p_1'\rangle.$$

The only applicable rule is S-Recv-Sec. Therefore $\Gamma(x) \not\sqsubseteq l_{\text{adv}}$. Hence, the second run must also be produced by the same rules.
Hence, $c_1' = c_2' = \texttt{stop}$ and $\alpha_1 = \alpha_2 = \epsilon$.
Since $\Gamma(x) \not\sqsubseteq l_{\text{adv}}$, $m_1' = m_1[x \leftarrow v_1]$, and $m_2' = m_2[x \leftarrow v_2]$, we have that $m_1' \sim_{l_{\text{adv}}} m_1$ and $m_2' \sim_{l_{\text{adv}}} m_2$. By transitivity of the relation, and by hypothesis, $m_1' \sim_{l_{\text{adv}}} m_2'$.
For the same reason, $r_1' \sim_{l_{\text{adv}}} r_1$ and $r_2' \sim_{l_{\text{adv}}} r_2$, and then $r_1' \sim_{l_{\text{adv}}} r_2'$.
$s_1' = s_1$ and $s_2' = s_2$ thus by hypothesis $s_1' \sim_{l_{\text{adv}}} s_2'$.

Let $l' \sqsubseteq l_{\text{adv}}$. Let us prove that $p_1'(l') + t_1' = p_2'(l') + t_2'$. Let us consider four cases:

- $p_1(l) < 0$ and $p_2(l) < 0$. Then $p'_1(l') + t'_1 = p_1(l') + t_1 + 1$ and $p'_2(l') + t'_2 = p_2(l') + t_2 + 1$. We conclude by hypothesis.
- $p_1(l) < 0$ and $p_2(l) \geq 0$. Then $p'_1(l') + t'_1 = p_1(l') + t_1 + 1$. There are two cases:
  1. $l \sqsubseteq l'$. Then $p'_2(l') = 0$, and $t'_2 = t_2 + p_2(l) + 1$. Therefore,

$$\begin{aligned} p'_1(l') + t'_1 &= p_1(l') + t_1 + 1 \\ &= p_2(l') + t_2 + 1 \text{ by hypothesis} \\ &= p'_2(l') + t'_2 \end{aligned}$$

  2. $l \not\sqsubseteq l'$. Then $p'_2(l') = p_2(l') - p_2(l)$, and $t'_2 = t_2 + p_2(l) + 1$. Therefore,

$$\begin{aligned} p'_1(l') + t'_1 &= p_1(l') + t_1 + 1 \\ &= p_2(l') + t_2 + 1 \text{ by hypothesis} \\ &= p_2(l') - p_2(l) + t_2 + p_2(l) + 1 \\ &= p_2(l') - p_2(l) + t'_2 \end{aligned}$$

- The symmetric case $p_1(l) \geq 0$ and $p_2(l) < 0$ is similar
- $p_1(l) \geq 0$ and $p_2(l) \geq 0$.
  There are two cases:
  - Suppose first that $l \sqsubseteq l'$. Then $p'_1(l') = 0 = p'_2(l')$. Therefore,

$$\begin{aligned} p'_1(l') + t'_1 &= 0 + t_1 + p_1(l) && \text{by defintion of } t'_1 \\ &= 0 + t_2 + p_2(l) && \text{because } l \sqsubseteq l_{\text{adv}} \text{ and by hypothesis} \\ &= p'_2(l') + t'_2 \end{aligned}$$

  - Otherwise, suppose that $l \not\sqsubseteq l'$. Then $p'_1(l') = p_1(l') - p_1(l)$ and $p'_2(l') = p_2(l') - p_2(l)$. Then,

$$\begin{aligned} p'_1(l') + t'_1 &= p_1(l') - p_1(l) + t_1 + p_1(l) && \text{by defintion of } t'_1 \\ &= p_1(l') + t_1 \\ &= p_2(l') + t_2 && \text{by hypothesis} \\ &= p_2(l') - p_2(l) + t_2 + p_2(l) \\ &= p'_2(l') + t'_2 \end{aligned}$$

Rule Bridge-Ev:

Then $\alpha_1 \neq \epsilon$. The only applicable rule is S-Recv-Pub, thus $\Gamma(x) \sqsubseteq l_{\text{adv}}$, $c'_1 = \texttt{stop}$, $\alpha_1 = A(x, v_1, \Gamma(x))$ with

$$v_1 = s_l(\{(v', l', t') \in s_1 \mid l = l'\}, \{(v', l', t') \in r_1 \mid l = l'\})$$

and

$$\langle c_1, m_1, t_1, s_1, r_1, p_1 \rangle \rightarrow_{\alpha_1} \langle \texttt{stop}, m'_1, t'_1, s'_1, r'_1, p'_1 \rangle.$$

Since $\Gamma(x) \sqsubseteq l_{\text{adv}}$, the second run must also be produced by the same rules.

Hence, $c'_1 = c'_2 = \texttt{stop}$ and $\alpha_2 = A(x, v_2, \Gamma(x))$ with

$$v_2 = s_l(\{(v', l', t') \in s_2 \mid l = l'\}, \{(v', l', t') \in r_2 \mid l = l'\}).$$

Since $\Gamma(x) \sqsubseteq l_{\text{adv}}$, and since the program is well-typed, $l \sqsubseteq l_{\text{adv}}$. Therefore, $v_1 = v_2$ and $\alpha_1 = \alpha_2$.
Now, $m'_1 = m_1[x \leftarrow v_1]$ and $m'_2 = m_2[x \leftarrow v_1]$. Since $m_1 \sim_{l_{\text{adv}}} m_2$, $m'_1 \sim_{l_{\text{adv}}} m'_2$.
For the same reason, $r'_1 \sim_{l_{\text{adv}}} r'_2$.
$s'_1 = s_1$ and $s'_2 = s_2$ thus $s'_1 \sim_{l_{\text{adv}}} s'_2$.

Let $l' \sqsubseteq l_{\text{adv}}$. Let us prove that $p'_1(l') + t'_1 = p'_2(l') + t'_2$. Let us consider four cases:

- $p_1(l) < 0$ and $p_2(l) < 0$. Then $p'_1(l') + t'_1 = p_1(l') + t_1 + 1$ and $p'_2(l') + t'_2 = p_2(l') + t_2 + 1$. We conclude by hypothesis.
- $p_1(l) < 0$ and $p_2(l) \geq 0$. Then $p'_1(l') + t'_1 = p_1(l') + t_1 + 1$. There are two cases:
  1. $l \sqsubseteq l'$. Then $p'_2(l') = 0$, and $t'_2 = t_2 + p_2(l) + 1$. Therefore,
  $$\begin{aligned} p'_1(l') + t'_1 &= p_1(l') + t_1 + 1 \\ &= p_2(l') + t_2 + 1 \text{ by hypothesis} \\ &= p'_2(l') + t'_2 \end{aligned}$$
  2. $l \not\sqsubseteq l'$. Then $p'_2(l') = p_2(l') - p_2(l)$, and $t'_2 = t_2 + p_2(l) + 1$. Therefore,
  $$\begin{aligned} p'_1(l') + t'_1 &= p_1(l') + t_1 + 1 \\ &= p_2(l') + t_2 + 1 \text{ by hypothesis} \\ &= p_2(l') - p_2(l) + t_2 + p_2(l) + 1 \\ &= p_2(l') - p_2(l) + t'_2 \end{aligned}$$
- The symmetric case $p_1(l) \geq 0$ and $p_2(l) < 0$ is similar
- $p_1(l) \geq 0$ and $p_2(l) \geq 0$.
  There are two cases:
  - Suppose first that $l \sqsubseteq l'$. Then $p'_1(l') = 0 = p'_2(l')$. Therefore,
  $$\begin{aligned} p'_1(l') + t'_1 &= 0 + t_1 + p_1(l) && \text{by defintion of } t'_1 \\ &= 0 + t_2 + p_2(l) && \text{because } l \sqsubseteq l_{\text{adv}} \text{ and by hypothesis} \\ &= p'_2(l') + t'_2 \end{aligned}$$
  - Otherwise, suppose that $l \not\sqsubseteq l'$. Then $p'_1(l') = p_1(l') - p_1(l)$ and $p'_2(l') = p_2(l') - p_2(l)$. Then,
  $$\begin{aligned} p'_1(l') + t'_1 &= p_1(l') - p_1(l) + t_1 + p_1(l) && \text{by defintion of } t'_1 \\ &= p_1(l') + t_1 \\ &= p_2(l') + t_2 && \text{by hypothesis} \\ &= p_2(l') - p_2(l) + t_2 + p_2(l) \\ &= p'_2(l') + t'_2 \end{aligned}$$

Case $c_1 = \texttt{padr}(v, l, p_{0,1}, v_{0,1}, t_{0,1})\ \texttt{do}\ d_1$**:**
Therefore $c_2 = \texttt{padr}(v, l, p_{0,2}, v_{0,2}, t_{0,2})\ \texttt{do}\ d_2$ with $d_1 \sim_{l_{\text{adv}}} d_2$.
We apply the lemma regarding the inner bridge relation for paddings to both bridge relation:
For the first relation, we obtain that we are in one of two cases:

1. $c'_1 = \texttt{stop}$ and there exist p''_1 such that
$$\langle d_1, m_1, t_1, s_1, r_1, p_1\rangle \curvearrowright^0_{\alpha_1} \langle \texttt{stop}, m'_1, t'_1, s'_1, r'_1, p''_1\rangle.$$
2. There exist e' and $d'_1 \neq \texttt{stop}$ such that $c'_1 = \texttt{padr}(v'_1, l, p_{0,1}, v_{0,1}, t_{0,1})\ \texttt{do}\ d'_1$ and
$$\langle d_1, m_1, t_1, s_1, r_1, p_1\rangle \curvearrowright^0_{\alpha_1} \langle d'_1, m'_1, t'_1, s'_1, r'_1, p'_1\rangle.$$

For the second relation, we obtain that we are in one of two cases:

1. $c'_2 = \texttt{stop}$ and there exist p''_2 such that
$$\langle d_2, m_2, t_2, s_2, r_2, p_2\rangle \curvearrowright^{n_2}_{\alpha_2} \langle \texttt{stop}, m'_2, t'_2, s'_2, r'_2, p''_2\rangle.$$
2. There exist e' and $d'_2 \neq \texttt{stop}$ such that $c'_2 = \texttt{padr}(v'_2, l, p_{0,2}, v_{0,2}, t_{0,2})\ \texttt{do}\ d'_2$ and
$$\langle d_2, m_2, t_2, s_2, r_2, p_2\rangle \curvearrowright^{n_2}_{\alpha_2} \langle d'_2, m'_2, t'_2, s'_2, r'_2, p'_2\rangle.$$

If we apply the inner induction hypothesis to case 1.2. (resp 2.1.) we obtain that $\texttt{stop} = d'_1 \neq \texttt{stop}$ (resp. $\texttt{stop} = d'_2 \neq \texttt{stop}$) which is impossible.
Let us consider the other cases:
Case 1.1: By applying the inner induction hypothesis to d_1 and d_2, we get that for all $l' \sqsubseteq l_{\text{adv}}$, $t'_1 + p''_1(l') = t'_2 + p''_2(l')$, and $\alpha_1 = \alpha_2$.
We also have that

$$p'_1 = l' \mapsto \begin{cases} p_{0,1}(l') + v_{0,1} - (t'_1 - t_{0,1}) & \text{if } l \not\sqsubseteq l' \\ p''_1(l') & \text{otherwise} \end{cases}$$

and

$$p'_2 = l' \mapsto \begin{cases} p_{0,2}(l') + v_{0,2} - (t'_2 - t_{0,2}) & \text{if } l \not\sqsubseteq l' \\ p''_2(l') & \text{otherwise} \end{cases}.$$

Let $l' \sqsubseteq l_{\text{adv}}$. Suppose first that $l \sqsubseteq l'$. Then $p'_1(l') = p''_1(l')$ and $p'_2(l') = p''_2(l')$.
Hence, $t'_1 + p'_1(l') = t'_2 + p'_2(l')$.

Otherwise, suppose that $l \not\sqsubseteq l'$. Then $p'_1(l') = p_{0,1}(l') + v_{0,1} - (t'_1 - t_{0,1})$ therefore

$$\begin{aligned} p'_1(l') + t'_1 &= p_{0,1}(l') + v_{0,1} + t_{0,1} \\ &= p_{0,2}(l') + v_{0,2} + t_{0,2} \text{ by definition of } \sim_{l_{\text{adv}}} \text{ for commands} \\ &= p'_2(l') + t'_2 \end{aligned}$$

Case 2.2: By applying the inner induction hypothesis to d_1 and d_2 we obtain the result immediatly.
We have indeed $c'_1 \sim_{l_{\text{adv}}} c'_2$ by definition of $\sim_{l_{\text{adv}}}$.

Case $c_1 = \texttt{pad}(e, l)\ \texttt{do}\ c$**:** Not applicable because $n = 0$

Inductive case: by induction on the structure of c_1:

Case $c_1 = \texttt{skip}$**:** not applicable because $n > 0$

Case $c_1 = x := e$**:** not applicable because $n > 0$

Case $c_1 = d_{1,1}; d_{2,1}$**:** Then $c_2 = d_{1,2}; d_{2,2}$ with $d_{1,1} \sim_{l_{\text{adv}}} d_{1,2}$ and $d_{2,1} \sim_{l_{\text{adv}}} d_{2,2}$.
By applying the lemma for sequential composition, we obtain the following:

1. For the first run we have two possibilities:
   (a) $n_1 > 0$ and there are k_1, m''_1, t''_1, s''_1, r''_1 and p''_1 such that $k_1 < n$ and

$$\langle d_{1,1}, m_1, t_1, s_1, r_1, p_1\rangle \curvearrowright_{\epsilon}^{k_1} \langle \texttt{stop}, m''_1, t''_1, s''_1, r''_1, p''_1\rangle \tag{1}$$

   and

$$\langle d_{2,1}, m''_1, t''_1, s''_1, r''_1, p''_1\rangle \curvearrowright_{\alpha_1}^{n-k_1-1} \langle c'_1, m'_1, t'_1, s'_1, r'_1, p'_1\rangle \tag{2}$$

   (b) $\alpha_1 \neq \epsilon$ and ther eis d''_1 such that:

$$\langle d_{1,1}, m_1, t_1, s_1, r_1, p_1\rangle \curvearrowright_{\alpha_1}^{n} \langle d''_1, m'_1, t'_1, s'_1, r'_1, p'_1\rangle \tag{3}$$

   and $c'_1 = \begin{cases} d''_1; d_{2,1} & \text{if } d''_2 \neq \texttt{stop} \\ d_{2,1} & \text{otherwise} \end{cases}$.
2. For the second run we have two possibilities:
   (a) $n_2 > 0$ and there are k_2, m''_2, t''_2, s''_2, r''_2 and p''_2 such that $k_2 < n_2$ and

$$\langle d_{1,2}, m_2, t_2, s_2, r_2, p_2\rangle \curvearrowright_{\epsilon}^{k_2} \langle \texttt{stop}, m''_2, t''_2, s''_2, r''_2, p''_2\rangle \tag{4}$$

   and

$$\langle d_{2,2}, m''_2, t''_2, s''_2, r''_2, p''_2\rangle \curvearrowright_{\alpha_2}^{n_2-k_2-1} \langle c'_2, m'_2, t'_2, s'_2, r'_2, p'_2\rangle \tag{5}$$

   (b) $\alpha_2 \neq \epsilon$ and there is d''_2 such that:

$$\langle d_{1,2}, m_2, t_2, s_2, r_2, p_2\rangle \curvearrowright_{\alpha_2}^{n} \langle d''_2, m'_2, t'_2, s'_2, r'_2, p'_2\rangle \tag{6}$$

   and $c'_2 = \begin{cases} d''_1; d_{2,2} & \text{if } d''_2 \neq \texttt{stop} \\ d_{2,2} & \text{otherwise} \end{cases}$.

Let us study all four cases:

Case 1a + 2b and 1b + 2a: by applying the innder induction hypothesis we obtain that $\alpha_2 = \epsilon$ in the first case, and $\alpha_1 = \epsilon$ in the second: these cases are impossible.

Case 1a + 2a: We apply the induction hypothesis to $d_{1,1}$ and $d_{1,2}$, which give us the equivalence of the two intermediary configuration, then to $d_{2,1}$ and $d_{2,2}$ which gives us the equivalence of the two final configuration.

Case 1b + 2b: We are done immediatly by applying the inner induction hypothesis to $d_{1,1}$ and $d_{1,2}$.

Case $c_1 = \texttt{if}\ e\ \texttt{then}\ d_1\ \texttt{else}\ d_1'$**:**

Then $c_2 = \texttt{if}\ e\ \texttt{then}\ d_2\ \texttt{else}\ d_2'$ with $d_1 \sim_{l_{\text{adv}}} d_2$ and $d_1' \sim_{l_{\text{adv}}} d_2'$

By T-If, $\Gamma, pc \vdash d_1$, $\Gamma, pc \vdash d_1'$, $\Gamma, pc \vdash d_2$, $\Gamma, pc \vdash d_2'$ and there is l such that $\Gamma \vdash e : l$ and $l \sqsubseteq pc$. If $l \sqsubseteq l_{\text{adv}}$, then by the lemma of noninterference for expressions, both runs take the same branch, and we are done by the inner induction hypothesis.

Otherwise, if $l \not\sqsubseteq l_{\text{adv}}$, then $pc \not\sqsubseteq l_{\text{adv}}$, and by the lemma regarding the update in secret context, $m_1 \sim_l m_1'$ and $m_2 \sim_l m_2'$. By transitivity of $\sim_l$, $m_1' \sim_l m_2'$ This is also the case for the messages.

Case $c = \texttt{while}\ e\ \texttt{do}\ c'$**:** Immediate with the outer induction hypothesis

Case $c = \texttt{send}(l', e)$**:** not applicable because $n > 0$

Case $c = x \leftarrow \texttt{recv}(l')$**:** not applicable because $n > 0$

Case $c_1 = \texttt{padr}(v, l, p_{0,1}, v_{0,1}, t_{0,1})\ \texttt{do}\ d_1$**:**

Therefore $c_2 = \texttt{padr}(v, l, p_{0,2}, v_{0,2}, t_{0,2})\ \texttt{do}\ d_2$ with $d_1 \sim_{l_{\text{adv}}} d_2$.

We apply the lemma regarding the inner bridge relation for paddings to both bridge relation:

For the first relation, we obtain that we are in one of two cases:

1. $c_1' = \texttt{stop}$ and there exist p_1'' such that
$$\langle d_1, m_1, t_1, s_1, r_1, p_1\rangle \curvearrowright_{\alpha_1}^{n_1} \langle \texttt{stop}, m_1', t_1', s_1', r_1', p_1''\rangle.$$
2. There exist e' and $d_1' \neq \texttt{stop}$ such that $c_1' = \texttt{padr}(v_1', l, p_{0,1}, v_{0,1}, t_{0,1})\ \texttt{do}\ d_1'$ and
$$\langle d_1, m_1, t_1, s_1, r_1, p_1\rangle \curvearrowright_{\alpha_1}^{n_1} \langle d_1', m_1', t_1', s_1', r_1', p_1'\rangle.$$

For the second relation, we obtain that we are in one of two cases:

1. $c_2' = \texttt{stop}$ and there exist p_2'' such that
$$\langle d_2, m_2, t_2, s_2, r_2, p_2\rangle \curvearrowright_{\alpha_2}^{n_2} \langle \texttt{stop}, m_2', t_2', s_2', r_2', p_2''\rangle.$$
2. There exist e' and $d_2' \neq \texttt{stop}$ such that $c_2' = \texttt{padr}(v_2', l, p_{0,2}, v_{0,2}, t_{0,2})\ \texttt{do}\ d_2'$ and
$$\langle d_2, m_2, t_2, s_2, r_2, p_2\rangle \curvearrowright_{\alpha_2}^{n_2} \langle d_2', m_2', t_2', s_2', r_2', p_2'\rangle.$$

If we apply the inner induction hypothesis to case 1.2. (resp 2.1.) we obtain that $\texttt{stop} = d_1' \neq \texttt{stop}$ (resp. $\texttt{stop} = d_2' \neq \texttt{stop}$) which is impossible.

Let us consider the other cases:

Case 1.1: By applying the inner induction hypothesis to d_1 and d_2, we get that for all $l' \sqsubseteq l_{\text{adv}}$, $t_1' + p_1''(l') = t_2' + p_2''(l')$, and $\alpha_1 = \alpha_2$.

We also have that

$$p_1' = l' \mapsto \begin{cases} p_{0,1}(l') + v_{0,1} - (t_1' - t_{0,1}) & \text{if } l \not\sqsubseteq l' \\ p_1''(l') & \text{otherwise} \end{cases}$$

and

$$p_2' = l' \mapsto \begin{cases} p_{0,2}(l') + v_{0,2} - (t_2' - t_{0,2}) & \text{if } l \not\sqsubseteq l' \\ p_2''(l') & \text{otherwise} \end{cases}.$$

Let $l' \sqsubseteq l_{\text{adv}}$. Suppose first that $l \sqsubseteq l'$. Then $p_1'(l') = p_1''(l')$ and $p_2'(l') = p_2''(l')$.
Hence, $t_1' + p_1'(l') = t_2' + p_2'(l')$.

Otherwise, suppose that $l \not\sqsubseteq l'$. Then $p_1'(l') = p_{0,1}(l') + v_{0,1} - (t_1' - t_{0,1})$ therefore

$$\begin{aligned} p_1'(l') + t_1' &= p_{0,1}(l') + v_{0,1} + t_{0,1} \\ &= p_{0,2}(l') + v_{0,2} + t_{0,2} \text{ by definition of } \sim_{l_{\text{adv}}} \text{ for commands} \\ &= p_2'(l') + t_2' \end{aligned}$$

Case 2.2: By applying the inner induction hypothesis to d_1 and d_2 we obtain the result immediatly.
We have indeed $c_1' \sim_{l_{\text{adv}}} c_2'$ by definition of $\sim_{l_{\text{adv}}}$.

Case $c = \texttt{pad}(e, l)$ `do` d**:** The only applicable rule is Bridge-Multi.
Since the program is well-typed, there exists l' such that $\Gamma \vdash e : l'$. If $l' \sqsubseteq l_{\text{adv}}$, then by noninterference for the expression e, and then by the outer induction hypothesis, we obtain the result.
Otherwise, $l' \not\sqsubseteq l_{\text{adv}}$. Then, $pc \sqcup l \not\sqsubseteq l_{\text{adv}}$, and since $\Gamma, pc \sqcup l \vdash d$, and since commands typed in secret context do not make public events, we obtain the result.

□

B.5 Proof of Theorem 1

A final technical piece that we need to put in place before proving Theorem 1 is that our bridge relation is adequate w.r.t. the original semantics.

Lemma 10 (Bridge adequacy). *Given a program c such that $\Gamma, pc \vdash c$ and m, s, r, t and p such that $\langle c, m, t, s, r, p\rangle \to^n \langle \texttt{stop}, m'', t'', s'', r'', p''\rangle$.*

Then, there are c', m', t', s', r', p', α, k and n' such that

$$\langle c, m, t, s, r, p\rangle \curvearrowright_{\alpha}^{k} \langle c', m', t', s', r', p'\rangle$$

and $\langle c', m', t', s', r', p'\rangle \to^{n'} \langle \texttt{stop}, m'', t'', s'', r'', p''$ where $k + n' + 1 = n$.

Proof. By induction on n. The base case $n = 1$ is trivial. For the inductive case, assume that the lemma holds for $n - 1$ steps, and consider the case of n steps.

We have that $\langle c, m, t, s, r, p\rangle \to \langle c', m', t', s', r', p'\rangle$ and $\langle c', m', t', s', r', p'\rangle \to^{n-1} \langle \texttt{stop}, m'', t'', s'', r'', p''\rangle$. There is some α such that $\langle c, m, t, s, r, p\rangle \to_\alpha \langle c', m', t', s', r', p'\rangle$.

We consider two possibilities:

Case $\alpha \neq \epsilon$**:** We are done by Bridge-Public, with $k = 0$ and $n' = n - 1$

Case $\alpha = \epsilon$**:** We have two cases:

1. n-1 = 0. Then by bridge-stop we are done, and $k = 0$ and $n' = n - 1$.
2. n-1 > 0. Then $c' \neq \texttt{stop}$. By the induction hypothesis, and bridge-multi we have the result.

□

We now revisit our main theorem.

Restatement of 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:*

$$\langle c, m_1, t_1, s_1, r_1, p_1\rangle \to^{n_1} \langle \texttt{stop}, m_1', t_1', s_1', r_1', p_1'\rangle$$

and

$$\langle c, m_2, t_2, s_2, r_2, p_2\rangle \to^{n_2} \langle \texttt{stop}, m_2', t_2', s_2', r_2', p_2'\rangle$$

it holds that:

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

Proof. By strong induction on n_1:

Base case $n_1 = 0$**:** Immediate.

Inductive case: By bridge adequacy, bridge noninterference theorem, and the induction hypothesis.

□