Towards Language-Based Mitigation of Traffic Analysis Attacks

Jeppe Fredsgaard Blaabjerg
Aarhus University
jfblaa@cs.au.dk

Aslan Askarov
Aarhus University
aslan@cs.au.dk

***Abstract*—Traffic analysis attacks pose a major risk for online security. Distinctive patterns in communication act as fingerprints, enabling adversaries to de-anonymise communicating parties or to infer sensitive information. Despite the attacks being known for decades, practical solution are scarce. Network layer countermeasures have relied on black box padding schemes that require significant overheads in latency and bandwidth to mitigate the attacks, without fundamentally preventing them, and the problem has received little attention in the language-based information flow literature. Language-based methods provide a strong foundation for fundamentally addressing security issues, but previous work has overwhelmingly assumed that interactive programs communicate over secure channels, where messages are undetectable by unprivileged adversaries. This assumption is too strong for online communication where packets can be trivially observed by eavesdropping. In this paper we introduce SELENE, a small language for principled, provably secure communication over channels where packets are publicly observable, and we demonstrate how our program level defence can reduce the latency and bandwidth overheads induced compared with program-agnostic defence mechanisms. We believe that our results constitute a step towards practical, secure online communication.**

***Index Terms*—Traffic analysis, noninterference, language-based security**

I. Introduction

Work on traffic analysis attacks has shown that many systems and services are vulnerable to de-anonymisation and loss of secrecy by producing distinctive patterns in their network traffic. Traffic analysis has particularly been studied in the context of anonymous communication and website fingerprinting [12, 16, 22, 23, 28, 29, 32]. Defence strategies against website fingerprinting are commonly done at the network level [12], and rely on constant rate padding, where source traffic is morphed to fit a predefined target pattern [19]. Constant rate padding can be applied in a black box fashion, making it an intuitively appealing technique against website fingerprinting. However, it often falls short in practice as achieving a high degree of security introduces intolerable bandwidth and latency overheads for many applications [17], such as anonymous, low-latency browsing and communication [18] and privacy preserving IoT devices [1]. Cherubin et al. argue that the application layer defences against traffic analysis are more natural as they act directly on the objects that are fingerprinted at the network level, while defences at lower layers must model legitimate traffic in order to generate convincing traffic padding [12].

Website fingerprinting is not the only attack made possible by traffic analysis. Online services that process sensitive information are now ubiquitous and previous work has shown that many such services are vulnerable to attack, as their communication behaviour reveals system secrets. Analysis by Chen et al. suggests that the scope of the issue is industry-wide [11]. Their study finds that design features used for creating reactive sites generate characteristic traffic patterns that allow an adversary to infer highly detailed, sensitive user information. They demonstrate this vulnerability across a number of high-profile websites, e.g. they are able to infer which illness a user selects on an online health site, and argue that traffic analysis attacks pose an unprecedented threat to the confidentiality of user information processed by online systems, and that this information is often far more sensitive than identifying which website a user visits as studied in anonymity research.

Language-based information flow methods provide principled ways of enforcing that the observable behaviour of a program does not depend on secrets. The language-based approach is appealing as the security condition of noninterference [20] can be provably enforced using a type system. O'Neill et al. formulate a noninterference condition for interactive programs [27], where programs communicate over in- and output channels. Their condition requires that input on secret channels does not influence output on public channels. This condition has been used in a breadth of other work [3, 8, 10, 13, 14, 21, 31]. Unfortunately, the models used in these works assume that messages on secret channels are invisible to adversaries. Other work models Internet communication using expressly public channels [25]. This makes the security results inapplicable for reasoning about online services and distributed programs where secret information is shared between remote, trusted entities. The only other work we are aware of, that allows an adversary to observe the communication behaviour of a program on non-public channels, is by Sabelfeld and Mantel [30]. They consider encrypted channels that protect message contents, but do not hide message presence, and give a timing-sensitive security condition. Their security condition does not consider the size of messages, which can be exploited in traffic analysis attacks, and their semantics lets the blocking behaviour of receives on encrypted channels be public by letting number of available messages be public.

Even simple interactions are not secure when messages can be eavesdropped. We demonstrate this using four example programs that each highlight a different source of leaking. In the following examples we consider a simple two point lattice with elements {L, H} and ordering L ⊑ L, L ⊑ H, and H ⊑ H, and adopt the convention that low variables start with `l` and high variables start with `h`. For simplicity, we assume that `Public` is a channel at level L and all other channels, e.g. `Alice`, `Bob`, are at level H. We first consider a program, where a number of messages are sent depending on the value of a secret variable:

```
1    /* Program 1 - Message count */
2  h_count = 0;
3  while (h_count < h_secret)
4  do {
5    out(Alice,1);
6    h_count = h_count + 1;
7  }
```

Program 1 satisfies the common security conditions of previous works, as the value of secret variable `h_secret` only influences secret output (line 5). However, if traffic can be eavesdropped, the program is trivially insecure. By counting the number of messages sent, an adversary can easily infer the exact value of the secret, as the number of messages depends on the secret.

We assume that each message is tagged with recipient information and that the adversary can observe both the size of each message and the time at which it was sent, i.e., the adversary is timing-sensitive. These assumptions lead us to naturally identify three other sources of leaks exemplified by the following programs, where respectively the recipient, the size, and the timing of messages leak secrets. These programs would commonly be considered secure in previous work.

```
  /* Program 2 - Recipient of message */
if (h) then {
  out(Alice, 42);
} else {
  out(Bob, 42);
}
```

```
  /* Program 3 - Size of message */
if (h) then {
  out(Alice, "Hello");
} else {
  out(Alice, "");
}
```

```
  /* Program 4 - Time of message */
if (h) then {
  out(Alice, 42);
} else {
  sleep(100);
  out(Alice, 42);
}
```

As the above examples suggest, many convenient patterns in writing interactive programs are no longer secure when messages can be eavesdropped.

In this paper, we show that program level padding can be used for provably secure confidentiality against attackers observing the network trace. We do this by introducing SELENE, a Statically Enforced Language for Equivalence of Network Events. SELENE is a simple imperative programming language, that allows programmatic control over traffic shaping. We show that well-typed programs in SELENE satisfy timing-sensitive, progress-sensitive non-interference. We use a knowledge-based definition of non-interference [4] and show that an adversary learns no secrets by observing runs of well-typed programs. We assume that communication channels are partly observable. Namely, we assume that the presence of messages and the associated meta-information is publicly visible, while the contents of messages is only visible to trusted parties.

Our strategy for preventing traffic analysis attacks is to provide programmatic control over traffic shaping. We do this by splitting message sending into two distinct concepts: message allocation and message population. To this end, SELENE uses two novel language primitives, `schedule` and `queue`, that respectively allocate a number of packets to be sent on a channel and add to a buffered output queue for a channel. This simple strategy allows for utilising program level information to keep latency and bandwidth overheads low when compared with black-box padding. This property is particularly beneficial for resource constrained systems. However, the strategy also comes with a downside, namely a restriction to when new traffic may be scheduled.

We present the formal semantics of the language in Section II, but here present a few program examples possible in the language.

Consider a scenario where a doctor has asked a patient to take a home-test for an illness and to return the result. Depending on the result, the doctor may make a referral to a specialist clinic. Any message sent from the doctor to the clinic is publicly observable and plainly sending a referral will naturally leak that the patient returned a positive test result. However, if the doctor commits ahead of time to sending *some* message to the clinic, regardless of the results of the test, the confidentiality of the patient's information can be protected.

```
1     /* Program 5 - Referral */
2   // Size of int
3   l_size = sizeof(0);
4
5   // Send to specialist in 300 time units
6   schedule(Clinic,l_size,300);
7
8   // Get id and test result from patient
9   h_id = in(Patient);
10  h_is_positive = in(Patient);
11
12  if (h_is_positive) then {
13    queue(Clinic,h_id);
14  } else {
15    skip;
16  }
```

On line 6, the doctor schedules a send to the clinic in 300 time units. They await messages from the patient containing id number (line 9) and the test result (line 10), and if positive, the doctor queues a referral to the clinic (line 13). This strategy is somewhat optimistic as it may take more than 300 time units for the patient to send the result to the doctor. In this case, or in case the test result is negative, nothing will be queued to the clinic before the send occurs. If the queue is empty at the time of a scheduled send, dummy packets are sent instead. When, to whom, and how much the doctor sends is thereby made public, while *what* the doctor sends is kept secret.

As a second example, we consider the password checker in Program 6.

```
  /* Program 6 - Password checker */
string h_password;
int h_token;
l_size_ok = sizeof(h_token);
l_size_bad = sizeof("LOGIN FAILED");

schedule(Alice,max(l_size_ok,l_size_bad),100);
h_guess = in(Alice);
if (h_guess == h_password) then {
  queue(Alice,h_token);
} else {
  queue(Alice,"LOGIN FAILED");
}
```

The password checker stores a secret password and returns a token to be used as proof of authority upon receiving a successful guess. The password checker schedules bandwidth for sending either the token or a login failure message by using the maximum of the two sizes. By scheduling the response before the guess is received, the program does not leak whether a valid guess was received, let alone whether the guess was correct.

As a final example, we consider a small popularity poll. Alice wishes to know whether her opinion that dogs are better than cats is shared by a majority of people. She sets up a simple online voting service running Program 7 below:

```
  /* Program 7 - Popinion */
/* Alice asks: Are cats or dogs better?
  Vote: Cats = 1, Dogs = -1 */
int h_my_vote;
l_tally = 0;
l_count = 0;

while (l_count < 10)
do {
  l_vote = in(Public);
  if (l_vote == -1 || l_vote == 1) then {
    l_tally = l_tally + l_vote;
  } else {
    skip;
  }
  l_count = l_count + 1;
}

// Size of the longest message
l_size = sizeof("Most disagree");
schedule(Alice, l_size, 100);

if (h_my_vote * l_tally > 0) then {
  queue(Alice, "Most agree");
} else if (h_my_vote * l_tally < 0) then {
  queue(Alice, "Most disagree");
} else {
  queue(Alice, "Tie");
}
```

The voting service stores Alice's secret choice in variable `h_my_vote`, tallies ten votes from a public channel, and schedules sending to Alice using the size of the longest message and time based on an estimate of what is needed for the branching. Inferring upper bounds on the time needed for queuing is orthogonal to the work in this paper and we opt for using simple estimates. Finally, the service computes whether a majority agrees or disagrees with Alice and sends a corresponding message. We observe that no bandwidth is needed until ten votes have been received by the service. Since it cannot be determined statically when this occurs our approach reduces the traffic overhead induced compared with constant rate padding schemes as it allows scheduling of traffic on an as-needed basis, as long as the program context is public.

The main contributions of this paper are:

- We spotlight the gap in the assumptions made in the language-based information flow literature for interactive programs and the channels available for real-world, online communication.
- We introduce SELENE, a language for using channels with observable traffic information in a principled and provably secure way, thereby recovering the strong security guarantees of language-based techniques.
- We introduce a novel model that combines program and runtime behaviour in a single small-step semantics, and give a knowledge-based security condition for timing-sensitive, progress-sensitive noninterference.
- We provide a progress-sensitive type system using both values of a fixed size type and values of a variable size type.
- We prove soundness of our type system, thereby obtaining a static guarantee that well-typed programs in SELENE do not leak via traffic patterns.

The remainder of this paper is structured as follows. In Section II we specify the threat model and provide the syntax and semantics of SELENE. We define attacker knowledge and give a strong security condition against traffic analysis attacks in Section III. We present the security type system for SELENE and prove it sound in Section IV. Finally, we discuss our work in Section V and give related work in Section VI, before we conclude in Section VII.

II. Security Model and Language

This section presents our model and the syntax and semantics of our language.

A. *Security model*

SELENE is an interactive, imperative language for single threaded, interactive programs with blocking receives. The language is largely standard, apart from our message sending primitives and command `sizeof` for computing the size of a value. We assume a standard security lattice $\mathcal{L}$ of security levels ℓ, with distinguished top and bottom elements, $\top$ and $\bot$, lattice ordering $\sqsubseteq$, and least upper bound operation $\sqcup$. Each variable has a fixed security level that does not change during execution.

As is standard in prior work on information flow control, we focus on confidentiality at the local node. Remote nodes trusted at some level ℓ are also trusted to appropriately protect information sent to them up to level ℓ. We further assume that remote nodes are also running SELENE programs. We model incoming traffic using lists. This modelling choice was shown equivalent to functional strategies for modelling deterministic, interactive programs by Clark and Hunt [13]. To this end, we consider an input environment I mapping each channel to a (possibly empty) list of input packets and let program values be obtainable from a sequence of packets corresponding to the value. For simplicity, we identify channels by their security level.

We observe that traffic analysis attacks exploit patterns in traffic to make inferences about the secret state of a system without requiring that the adversary can read the contents of packets. We therefore make the simplifying assumption that the contents of packets are sufficiently protected against adversaries, e.g. by using encryption, but allow the adversary to observe the presence, recipient, and time of packets. We assume that packets are of fixed size, thereby transforming the question of packet size into a question of packet count.

B. Threat model

We consider interactive, distributed programs that communicate with remote network nodes. We consider an active adversary who is trusted at a security level ℓ_{adv}, who knows the program being run on the local node, and who knows initial secrets up to level ℓ_{adv}. Additionally, the adversary eavesdrops on incoming and outgoing encrypted communication of the local node, observing packet presence, timing, and the remote communication party. Communication on a channel is encrypted corresponding to the security level of the channel, and the adversary can decrypt and read packets on channels up to security level ℓ_{adv}. The objective of the adversary is to refine their knowledge on initial secrets.

C. The language and program semantics

$$
\begin{aligned}
e &::= n \mid s \mid x \mid e \oplus e \\
c &::= x = e \mid c; c \mid \texttt{skip} \mid \texttt{sleep}(e) \mid x = \texttt{sizeof}(e) \\
&\quad \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \texttt{do}\ c \\
&\quad \mid x = \texttt{in}(\ell) \mid \texttt{schedule}(\ell, e, e) \mid \texttt{queue}(\ell, e)
\end{aligned}
$$

Figure 1. Syntax of the language

Figure 1 presents the syntax of our language. We explain the formal semantics and explain the nonstandard features.

We use a big-step semantics for evaluating expressions and assume these take unit time. The rules are standard and are given in Fig. 2. We let $\oplus$ range over total operations on arithmetic expressions. The values of our language are integers n and strings s. We let *Int* denote the set of integers and *String* denote the set of strings and let $Val = Int \uplus String$.

$$
\frac{v \in Val}{\langle v, m \rangle \Downarrow v} \qquad \frac{m(x) = v}{\langle x, m \rangle \Downarrow v}
$$

$$
\frac{\langle e_1, m \rangle \Downarrow v_1 \qquad \langle e_2, m \rangle \Downarrow v_2 \qquad v = v_1 \oplus v_2}{\langle e_1 \oplus e_2, m \rangle \Downarrow v}
$$

Figure 2. Semantics for evaluating expressions

Programs are typed using fixed typing environment Γ. We write $\Gamma(x) = \sigma@\ell$ to denote that variable x has type σ and security level ℓ. The types of our language are **int** and $\mathbf{string}_\ell$, where ℓ is the security level of the size of the string. Input packets either contain (part of) an input value, or are dummy. We write $\boxed{v}_N^j$ to denote the j'th of N packets encoding value v and let $\boxed{\bullet}$ denote dummy packets.

For evaluating program commands c we use a small-step semantics transition $\langle c, m, I \rangle \xrightarrow{ts}_\alpha \langle c', m', I' \rangle$, where m is a memory, I is an input environment, and α is a program event generated by the step. Program steps take place at a time *ts*, however they do not increment time. We instead define a global semantics on top of the program semantics and let global steps increment time. We discuss the global semantics shortly. Program events can be empty, denoted by ϵ, or an assignment, enqueue, scheduling, or input event as given by the following grammar:

$$
\alpha ::= \epsilon \mid \mathsf{a}(x, v) \mid \mathsf{q}(\ell, v) \mid \mathsf{s}(\ell, n, n) \mid \mathsf{i}(\ell, x, v)
$$

Fig. 3 presents the stepping rules of our program operational semantics.

SizeOf: Command `sizeof` evaluates an expression to obtain a value and returns the number of packets needed to store that value. This is useful as the language requires the programmer to explicitly schedule the number of packets they wish to send. We assume a fixed packet size η, and assume that all integers are of fixed size, and that the size of a string is dependent on the length of the string.

In: The transitions of our small-step semantics for program commands are parametric in timestamps *ts*, allowing us to model blocking by conditioning transitions on *ts*. We model network input using primitive $x = \texttt{in}(\ell)$. To preserve the type of variable x, the input primitive determines whether $m(x)$ is an integer or a string value, captured by set A in rule IN, and uses this as argument for auxiliary function choose (Fig. 4), along with the packet sequence for channel ℓ and timestamp *ts*. Function choose is a partial function, modelling the potential for blocking. The choose function adds packets of the appropriate type to an accumulator used for decoding an input value, and steps over packets of other type, discarding any dummy packets. It returns a decoded value and the remaining packet sequence for the channel if successful.

$$\textsc{Assign}\quad \frac{\langle e, m\rangle \Downarrow v}{\langle x = e, m, I\rangle \xrightarrow{ts}_{\mathsf{a}(x,v)} \langle \mathtt{stop}, m[x \mapsto v], I\rangle}$$

$$\textsc{SizeOf}\quad \frac{\langle e, m\rangle \Downarrow v \qquad n = \left\lceil \dfrac{size(v)}{\eta} \right\rceil}{\langle x = \mathtt{sizeof}(e), m, I\rangle \xrightarrow{ts}_{\mathsf{a}(x,n)} \langle \mathtt{stop}, m[x \mapsto n], I\rangle}$$

$$\textsc{Skip}\quad \frac{}{\langle \mathtt{skip}, m, I\rangle \xrightarrow{ts}_{\epsilon} \langle \mathtt{stop}, m, I\rangle}$$

$$\textsc{Seq-1}\quad \frac{\langle c_1, m, I\rangle \xrightarrow{ts}_{\alpha} \langle c_1', m', I'\rangle \qquad c_1' \neq \mathtt{stop}}{\langle c_1; c_2, m, I\rangle \xrightarrow{ts}_{\alpha} \langle c_1'; c_2, m', I'\rangle}$$

$$\textsc{Seq-2}\quad \frac{\langle c_1, m, I\rangle \xrightarrow{ts}_{\alpha} \langle \mathtt{stop}, m', I'\rangle}{\langle c_1; c_2, m, I\rangle \xrightarrow{ts}_{\alpha} \langle c_2, m', I'\rangle}$$

$$\textsc{Sleep}\quad \frac{\langle e, m\rangle \Downarrow w \qquad w \geq 0 \qquad r = ts + w}{\langle \mathtt{sleep}(e), m, I\rangle \xrightarrow{ts}_{\epsilon} \langle \mathtt{await}(r), m, I\rangle}$$

$$\textsc{Await}\quad \frac{ts \geq r}{\langle \mathtt{await}(r), m, I\rangle \xrightarrow{ts}_{\epsilon} \langle \mathtt{stop}, m, I\rangle}$$

$$\textsc{If-T}\quad \frac{\langle e, m\rangle \Downarrow v \qquad v \neq 0}{\langle \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, m, I\rangle \xrightarrow{ts}_{\epsilon} \langle c_1, m, I\rangle}$$

$$\textsc{If-E}\quad \frac{\langle e, m\rangle \Downarrow 0}{\langle \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, m, I\rangle \xrightarrow{ts}_{\epsilon} \langle c_2, m, I\rangle}$$

$$\textsc{While}\quad \frac{}{\langle \mathtt{while}\ e\ \mathtt{do}\ c, m, I\rangle \xrightarrow{ts}_{\epsilon} \langle \mathtt{if}\ e\ \mathtt{then}\ c; \mathtt{while}\ e\ \mathtt{do}\ c\ \mathtt{else}\ \mathtt{skip}, m, I\rangle}$$

$$\textsc{In}\quad \frac{A \in \{Int, String\} \qquad m(x) \in A \qquad I(\ell) = \vec{p} \qquad (v, \vec{q}) = \mathsf{choose}(\vec{p}, A, ts, [])}{\langle x = \mathtt{in}(\ell), m, I\rangle \xrightarrow{ts}_{\mathsf{i}(\ell,x,v)} \langle \mathtt{stop}, m[x \mapsto v], I[\ell \mapsto \vec{q}]\rangle}$$

$$\textsc{Schedule}\quad \frac{\langle e_1, m\rangle \Downarrow n \qquad \langle e_2, m\rangle \Downarrow w \qquad w \geq 0 \qquad t = ts + w}{\langle \mathtt{schedule}(\ell, e_1, e_2), m, I\rangle \xrightarrow{ts}_{\mathsf{s}(\ell,n,t)} \langle \mathtt{stop}, m, I\rangle}$$

$$\textsc{Queue}\quad \frac{\langle e, m\rangle \Downarrow v}{\langle \mathtt{queue}(\ell, e), m, I\rangle \xrightarrow{ts}_{\mathsf{q}(\ell,v)} \langle \mathtt{stop}, m, I\rangle}$$

Figure 3. Local operational semantics

$$\mathsf{choose}(\vec{p}, A, t, acc) \triangleq \begin{cases} (v, \vec{p}) & \text{if } acc = \boxed{v}^1_N :: \ldots :: \boxed{v}^N_N \\ (v_r, \vec{r}) & \text{if } \vec{p} = (t', \boxed{v}^j_N) \cdot \vec{q} \text{ s.t. } v \in A \text{ and } t' \leq t \text{ and } \mathsf{choose}(\vec{q}, A, t, acc :: \boxed{v}^j_N) = (v_r, \vec{r}) \\ (v_r, (t', \boxed{v}^j_N) \cdot \vec{r}) & \text{if } \vec{p} = (t', \boxed{v}^j_N) \cdot \vec{q} \text{ s.t. } v \notin A \text{ and } t' \leq t \text{ and } \mathsf{choose}(\vec{q}, A, t, acc) = (v_r, \vec{r}) \\ (v_r, \vec{r}) & \text{if } \vec{p} = (t', \boxed{\bullet}) \cdot \vec{q} \text{ s.t. } t' \leq t \text{ and } \mathsf{choose}(\vec{q}, A, t, acc) = (v_r, \vec{r}) \end{cases}$$

Figure 4. Choose function

We let the packets be annotated with timestamps and require that all packets corresponding to a value have been received before the value can be obtained. That is, the timestamp of the final packet must be at or before the timestamp in the transition of the `in` command. If no value can be retrieved, the program blocks and cannot step.

We model internal input (e.g. reading files) only abstractly, by considering them bound in program variables.

Schedule and queue: The `schedule` command takes three arguments; a channel, a number of packets to be sent, and a delay before sending the packets. This issues a request to the runtime system. We describe the runtime system shortly. Command `queue` takes a channel and an expression as arguments and evaluates the expression to obtain a value. It then instructs the runtime system to add the value to a buffered output queue associated with the channel.

Internal commands: Commands `await` and `stop` are only used internally and are therefore not part of the language syntax. Command `await` is reached from command `sleep` and blocks for a specified duration of time. Command `stop` denotes a final program configuration that cannot step any further.

D. The runtime and the global semantics

A key feature of our model is the global configuration modelling the language runtime. The runtime maintains the output queues in output environment O and processes the packet schedule π. We present a small-step semantics for the global transitions in Fig. 5. We explain the interaction between the program configuration and global configuration in more detail. We let the global configuration contain a program configuration and require that the program steps whenever possible. For the sake of brevity, we let P denote a program configuration $\langle c, m, I\rangle$. Program steps are done by rule G-STEP and emit a possibly empty event α. We let the program communicate updates to the runtime through schedule and queue events. To this end, we apply update function upd (Fig. 6) to the schedule and output environment using event α. If the program step emits a queue event $\mathsf{q}(\ell, v)$, value v is split into a number of packets based on the size of the value, and the packets are added to the buffered output queue for channel ℓ. If the program step emits a schedule event $\mathsf{s}(\ell, n, t)$, we use function rsv to reserve time in the schedule for a number of packets on a channel by recursively adding to $\mathsf{dom}(\pi)$. We assume that the schedule is never full, i.e. the function will terminate having scheduled all n packets. We make the simplifying assumption that at most one packet can be sent in any single step and formally model the schedule as a partial function from timestamps to channels.

To model packets being sent, we extend the grammar for events with runtime events β. The runtime emits an empty event if the schedule is undefined for the current timestamp, otherwise we use function send defined below to obtain an event corresponding to the first packet in the scheduled channel's output queue, and an updated output environment. If no packets are queued on the channel, an empty dummy packet is generated and sent.

$$\mathsf{send}(O, \ell) \triangleq \begin{cases} (\mathsf{o}(\ell, p), O[\ell \mapsto \vec{q}]) & \text{if } O(\ell) = p \cdot \vec{q} \\ (\mathsf{o}(\ell, \boxdot), O) & \text{if } O(\ell) = [] \end{cases}$$

To combine program generated events α with runtime generated events β we let global events γ be a triple $(ts : \alpha, \beta)$, where ts is the timestamp of the event. The observations an attacker makes on a run of a program are given by a trace of global events, each containing the timestamp of the step, leading to a timing-sensitive model. While a network attacker does not observe program events α, maintaining them in global events is nevertheless useful, as it allows us to more easily reason about the exact state of a run. In Section III, we define our security condition in terms of an attacker that does not observe program events, i.e., that observes all program events as the empty event ϵ.

We extend the grammar as follows:

$$\beta ::= \epsilon \mid \mathsf{o}(\ell, p)$$
$$\gamma ::= (ts : \alpha, \beta)$$

The global configuration maintains clock ts that is incremented for each step. If a program configuration is blocking, that is, if it cannot take a step at the current timestamp, but has not stopped with command `stop`, the global configuration steps by G-BLOCK, processing the runtime and incrementing the clock. Finally, G-STOP allows the global configuration to continue processing the runtime after the program configuration has reached command `stop`, provided there are scheduled packets left to process.

III. SECURITY CONDITION

In this section we present the security condition for timing-sensitive, progress-sensitive noninterference.

We define our security condition using the knowledge-based approach [2]. The insight of this approach is to consider what an attacker observes during the execution of a program and define knowledge as the set of initial states that are consistent with seeing the execution up to this point. The security condition is then defined as a bound on how much the knowledge is allowed to change for each step of the execution. In this paper we do not consider declassification and we therefore require that attacker knowledge does not change with new observations.

A. Auxiliary definitions

We define attacker knowledge and timing-sensitive, progress-sensitive noninterference in terms of an equivalence relation on program configurations and the attacker observable trace emitted from a run. We give these auxiliary definitions before proceeding to define the security condition.

To denote that two memories are equivalent up to ℓ_{adv} we write $m \approx_{\ell_{adv}} m'$ (Definition 1).

Definition 1 (Memory equivalence up to level)**.** Two memories m and m' are equivalent up to level ℓ_{adv}, written $m \approx_{\ell_{adv}} m'$, if for all $x \in \mathsf{dom}(\Gamma)$ both the following hold:

1) $\Gamma(x) = \sigma @ \ell \wedge \ell \sqsubseteq \ell_{adv} \implies m(x) = m'(x)$
2) $\Gamma(x) = \mathbf{string}_{\ell'} @ \ell \wedge \ell' \sqsubseteq \ell_{adv} \implies size(m(x)) = size(m'(x))$

This definition captures that the values of attacker observable variables must have the same value, and that the size of the value of variables must be the same if the size is attacker observable.

We overload the notation and write $I \approx_{\ell_{adv}} I'$ to denote that two input environments are equivalent up to ℓ_{adv} (Definition 2). The definition requires equality of incoming packets on attacker observable channels, and uses relation $\approx^{net}_{\ell_{adv}}$ for high channels, requiring that these receive packets at the same timestamps. This captures an attacker that can observe the presence of incoming packets on all channels and when they arrive, but who cannot read the contents of packets on high

$$\textsc{G-Step}\quad \frac{P \xrightarrow[\alpha]{ts} P' \qquad (O', \pi') = \mathsf{upd}(O, \pi, \alpha) \qquad (\beta, O'') = \begin{cases} (\epsilon, O') & \text{if } ts \notin \mathsf{dom}(\pi') \\ \mathsf{send}(O', \ell) & \text{if } \pi'(ts) = \ell \end{cases}}{\langle\!| P, O, \pi, ts |\!\rangle \twoheadrightarrow_{(ts:\alpha,\beta)} \langle\!| P', O'', \pi', ts+1 |\!\rangle}$$

$$\textsc{G-Block}\quad \frac{\nexists P' : P \xrightarrow[\alpha]{ts} P' \qquad P = \langle c, m, I\rangle \qquad c \neq \texttt{stop} \qquad (\beta, O') = \begin{cases} (\epsilon, O) & \text{if } ts \notin \mathsf{dom}(\pi) \\ \mathsf{send}(O, \ell) & \text{if } \pi(ts) = \ell \end{cases}}{\langle\!| P, O, \pi, ts |\!\rangle \twoheadrightarrow_{(ts:\epsilon,\beta)} \langle\!| P, O', \pi, ts+1 |\!\rangle}$$

$$\textsc{G-Stop}\quad \frac{P = \langle \texttt{stop}, m, I\rangle \qquad \exists ts' \in \mathsf{dom}(\pi) : ts' \geq ts \qquad (\beta, O') = \begin{cases} (\epsilon, O) & \text{if } ts \notin \mathsf{dom}(\pi) \\ \mathsf{send}(O, \ell) & \text{if } \pi(ts) = \ell \end{cases}}{\langle\!| P, O, \pi, ts |\!\rangle \twoheadrightarrow_{(ts:\epsilon,\beta)} \langle\!| P, O', \pi, ts+1 |\!\rangle}$$

Figure 5. Global operational semantics

$$\mathsf{split}(v) \triangleq \boxed{v}^1_N \cdot \ldots \cdot \boxed{v}^N_N \qquad \text{where } N = \left\lceil \frac{size(v)}{\eta} \right\rceil$$

$$\mathsf{rsv}(\pi, \ell, n, t) \triangleq \begin{cases} \pi_r & \text{if } n > 0 \text{ and } t \notin \mathsf{dom}(\pi) \text{ and} \\ & \mathsf{rsv}(\pi[t \mapsto \ell], \ell, n-1, t+1) = \pi_r \\ \pi_r & \text{if } n > 0 \text{ and } t \in \mathsf{dom}(\pi) \text{ and} \\ & \mathsf{rsv}(\pi, \ell, n, t+1) = \pi_r \\ \pi & \text{if } n \leq 0 \end{cases}$$

$$\mathsf{upd}(O, \pi, \alpha) \triangleq \begin{cases} (O', \pi) & \text{if } \alpha = \mathsf{q}(\ell, v),\ \mathsf{split}(v) = \vec{p}, \\ & O(\ell) = \vec{q}, \text{and } O[\ell \mapsto (\vec{q} \cdot \vec{p})] = O' \\ (O, \pi') & \text{if } \alpha = \mathsf{s}(\ell, n, t) \text{ and} \\ & \mathsf{rsv}(\pi, \ell, n, t) = \pi' \\ (O, \pi) & \text{otherwise} \end{cases}$$

Figure 6. Runtime function

channels. We assume that incoming packets are sent by other SELENE programs and hence are of fixed size.

Definition 2 (Input environment equivalence up to level)**.** Two input environments I and I' are equivalent up to level ℓ_{adv}, written $I \approx_{\ell_{adv}} I'$, if

$$\frac{\ell \sqsubseteq \ell_{adv} \implies I_1(\ell) = I_2(\ell) \qquad \ell \not\sqsubseteq \ell_{adv} \implies I_1(\ell) \approx^{net}_{\ell_{adv}} I_2(\ell)}{I_1 \approx_{\ell_{adv}} I_2}$$

where $\approx^{net}_{\ell_{adv}}$ is defined by

$$\frac{}{[] \approx^{net}_{\ell_{adv}} []} \qquad \frac{\vec{p} \approx^{net}_{\ell_{adv}} \vec{q}}{(t, p_1) \cdot \vec{p} \approx^{net}_{\ell_{adv}} (t, p_2) \cdot \vec{q}}$$

We lift equivalences to program configurations in a straight forward way in Definition 3.

Definition 3 (Program configuration equivalence up to level)**.** Two program configurations $\langle c_1, m_1, I_1\rangle$ and $\langle c_2, m_2, I_2\rangle$ are equivalent up to level ℓ_{adv}, written

$$\langle c_1, m_1, I_1\rangle \approx_{\ell_{adv}} \langle c_2, m_2, I_2\rangle$$

if it holds that $c_1 = c_2$, $m_1 \approx_{\ell_{adv}} m_2$, and $I_1 \approx_{\ell_{adv}} I_2$.

We overload the notation even further and write $O \approx_{\ell_{adv}} O'$ to denote that two output environments are equivalent up to ℓ_{adv} (Definition 4).

Definition 4 (Output environment equivalence up to level)**.** Two output environments O_1 and O_2 are equivalent up to level ℓ_{adv}, written $O_1 \approx_{\ell_{adv}} O_2$, if

$$\frac{\ell \sqsubseteq \ell_{adv} \implies O_1(\ell) = O_2(\ell)}{O_1 \approx_{\ell_{adv}} O_2}$$

Next, we define runtime event projections. Runtime event projection captures the observable parts of output events emitted by the runtime. We write $\lfloor \beta \rfloor_{\ell_{adv}}$ to denote the projection of runtime event β to level ℓ_{adv} (Definition 5). We introduce a new event capturing the sending of packets with unobservable content. We extend the grammar for runtime events as follows:

$$\beta ::= \ldots \mid \mathsf{o}(\ell, -)$$

Runtime event $\mathsf{o}(\ell, p)$ projects to $\mathsf{o}(\ell, -)$ if the level of the channel does not flow to the level being projected to. This captures the assumption that the contents of packets can be securely hidden by cryptography, while the presence of packets and their recipient remain visible.

Definition 5 (Runtime event projection)**.** The projection of runtime event β to level ℓ_{adv}, written $\lfloor \beta \rfloor_{\ell_{adv}}$, is defined as

$$\lfloor \epsilon \rfloor_{\ell_{adv}} = \epsilon$$

$$\lfloor \mathsf{o}(\ell, p) \rfloor_{\ell_{adv}} = \begin{cases} \mathsf{o}(\ell, p) & \text{if } \ell \sqsubseteq \ell_{adv} \\ \mathsf{o}(\ell, -) & \text{if } \ell \not\sqsubseteq \ell_{adv} \end{cases}$$

We write $\tau \upharpoonright \ell_{adv}$ to denote the filtering of trace τ to what is visible at level ℓ_{adv} (Definition 6). We use this to restrict attacker knowledge to the steps where observable events are emitted. This allows us to consider secure programs such as `if h then sleep(10) else skip`, as no output occurs after branching on the secret. We let trace filtering fix the empty event ϵ as the program event component, thereby removing all program events emitted. This captures a network attacker, that only obtains new information by observing packets being sent.

Definition 6 (Trace filtering)**.** The filtering of a trace τ to level ℓ_{adv}, written $\tau \upharpoonright \ell_{adv}$, is defined as

$$\epsilon \upharpoonright \ell_{adv} = \epsilon$$

$$(\tau' \cdot (ts : \alpha, \beta)) \upharpoonright \ell_{adv} =$$

$$\begin{cases} \tau' \upharpoonright \ell_{adv} \cdot (ts : \epsilon, \lfloor \beta \rfloor_{\ell_{adv}}) & \text{if } \lfloor \beta \rfloor_{\ell_{adv}} \neq \epsilon \\ \tau' \upharpoonright \ell_{adv} & \text{otherwise} \end{cases}$$

As two final building blocks, we let O_{init} denote the initially empty output environment and let π_{init} denote the initially empty schedule. That is,

$$\forall \ell \in \mathcal{L} : O_{init}(\ell) = []$$
$$\mathsf{dom}(\pi_{init}) = \emptyset$$

B. Knowledge and noninterference

Using the above we define the knowledge of an attacker at level ℓ_{adv} after observing trace τ. This definition follows the style of other knowledge-based security conditions [5], and intuitively states that an attacker may not refine their knowledge by observing new events.

Definition 7 (Attacker knowledge)**.** Given a program configuration P, such that $\langle\!| P, O_{init}, \pi_{init}, 0 |\!\rangle \twoheadrightarrow^*_\tau \langle\!| P', O', \pi', ts' |\!\rangle$, the attacker knowledge at level ℓ_{adv} is the set of program configurations P_2, that are consistent with observations at that level:

$$\begin{aligned} & k(P, \tau, \ell_{adv}) \triangleq \\ & \quad \{P_2 \mid P \approx_{\ell_{adv}} P_2 \wedge \\ & \quad\quad \langle\!| P_2, O_{init}, \pi_{init}, 0 |\!\rangle \twoheadrightarrow^*_{\tau_2} \langle\!| P_2', O_2', \pi_2', ts_2' |\!\rangle \wedge \\ & \quad\quad (\tau \upharpoonright \ell_{adv}) = (\tau_2 \upharpoonright \ell_{adv})\} \end{aligned}$$

Using the definition of attacker knowledge we define timing-sensitive, progress-sensitive noninterference.

Definition 8 (Timing-sensitive, progress-sensitive noninterference)**.** Given program configuration P such that

$$\langle\!| P, O_{init}, \pi_{init}, 0 |\!\rangle \twoheadrightarrow^*_{\tau \cdot \gamma} \langle\!| P', O', \pi', ts' |\!\rangle$$

the run satisfies timing-sensitive, progress-sensitive noninterference if for all ℓ_{adv} it holds that

$$k(P, \tau \cdot \gamma, \ell_{adv}) \supseteq k(P, \tau, \ell_{adv})$$

This definition states that memories and input environments considered possible before observing global event γ are also considered possible after observing γ, capturing that the adversary learns nothing by observing the event. To demonstrate the security condition, we rewrite Program 3 from Section I in the syntax of SELENE. We consider one run where secret variable `h` is set to 1 and another where it is set to 0, and assume that $n+1$ packets are needed to send a string of length n.

```
  /* Program 3b */
if (h) then {
  queue(Alice, "Hello");
  size = sizeof("Hello");
  schedule(Alice,size,0);
} else {
  queue(Alice, "");
  size = sizeof("");
  schedule(Alice,size,0);
}
```

In the first run, 6 packets are scheduled and sent on channel `Alice` in order to send the string. In the second run, only a single packet is scheduled and sent. As the presence of every packet is observable to the attacker, they can distinguish between the two runs when a second packet is sent, hence violating Definition 8.

IV. Enforcement

In this section we present the security type system for SELENE and prove that all runs of well-typed programs satisfy timing-sensitive, progress-sensitive noninterference. Despite considering an attacker that observes only network events, timing-sensitivity makes the attacker quite strong. We settle for a rather restrictive type system that is secure against a stronger, internal attacker and we use this to show security for a network attacker. Previous work on noninterference for interactive programs by O'Neill et al. [27] achieves a more permissive type system by assuming a timing-insensitive attacker, disallowing high-loops, and assuming that new input is always available, thereby ruling out high-divergence of programs. Unfortunately, timing-insensitive attacker models are insufficient for external attackers, such as the network attacker we consider, as we cannot restrict an attacker's access to timing channels.

As our security condition is timing-sensitive and progress-sensitive, our typing judgements are progress-sensitive. They are of the form

$$\Gamma, pc \vdash c : pc'$$

where pc is the program counter before typing the command and pc' is the program counter after. As we do not consider pc-declassification in this paper, the program counter never goes down. This leads to so-called *pc*-creep, which significantly restricts the programs that can be written in the language. We leave it to future work to explore language primitives for mitigating this and to investigate the security impact of *pc*-declassification on traffic analysis attacks.

A. Type system

The definitions of memory equivalence and event projection in Section III implicitly require a well-formedness condition on security types of variables in Γ. We now formally state this condition and assume for the rest of the paper that all variables in Γ have well-formed security types.

$$\frac{}{\vdash_{\mathsf{wf}} \mathbf{int}@\ell} \qquad \frac{\ell' \sqsubseteq \ell}{\vdash_{\mathsf{wf}} \mathbf{string}_{\ell'}@\ell}$$

The intuition behind the well-formedness condition is that knowing a value implies knowing its size, but not the other way around. Next, we define subtyping relation $<:$. The relation is straight forward, using lattice ordering $\sqsubseteq$ on size levels as a condition on strings.

$$\frac{}{\mathbf{int} <: \mathbf{int}} \qquad \frac{\ell_1 \sqsubseteq \ell_2}{\mathbf{string}_{\ell_1} <: \mathbf{string}_{\ell_2}}$$

We write $\sigma \nearrow \ell$ to denote raising type σ to at least level ℓ. This is used to account for *pc*-taint when assigning strings, to prevent string size from leaking secrets. This allows us to concisely write the conditions for the typing rules.

$$\sigma \nearrow \ell \triangleq \begin{cases} \mathbf{int} & \text{if } \sigma = \mathbf{int} \\ \mathbf{string}_{(\ell \sqcup \ell')} & \text{if } \sigma = \mathbf{string}_{\ell'} \end{cases}$$

The type system for expressions is presented in Fig. 7. The rules are standard, except for the rule for string expressions. This rule follows from the well-formedness condition on security types, and intuitively as the string appears in the program text, hence the size of the string is public.

$$\frac{n \in \mathit{Int}}{\Gamma \vdash n : \mathbf{int}@\bot} \qquad \frac{s \in \mathit{String}}{\Gamma \vdash s : \mathbf{string}_\bot@\bot} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{int}@\ell_1 \qquad \Gamma \vdash e_2 : \mathbf{int}@\ell_2}{\Gamma \vdash e_1 \oplus e_2 : \mathbf{int}@\ell_1 \sqcup \ell_2}$$

Figure 7. Type system for expressions

We present our type system for commands in Fig. 8 and explain the nonstandard rules.

SizeOf: Rule T-SIZEOF expresses that the size of an integer value may be assigned to a variable conditioned only by *pc*. This is intuitively safe as integers have fixed size. The size of a string value may be assigned to a variable if the variable is at least as secret as the least upper bound of *pc* and the size level of the string.

In: Rule T-IN is similar to input rules in previous work. We require that the level of *pc* flows to the level of the channel ℓ. This is to preserve low equivalence of the input environment during steps under high *pc*. As a non-standard condition, the rule uses type raising and the subtyping relation to require $\sigma_x \nearrow \ell <: \sigma_x$. For $\sigma_x = \mathbf{int}$, this condition is trivially satisfied by the definitions. For $\sigma_x = \mathbf{string}_{\ell'}$, this condition corresponds to the condition $\ell \sqsubseteq \ell'$. Intuitively, we consider both the size and the value of a received strings to be as secret as the level of the channel.

Schedule and queue: Rule T-SCHEDULE restricts schedule commands to public *pc* and restricts the integer arguments to also be public. These conditions are natural, as we assume that traffic is publicly observable. As a consequence of progress-sensitive typing, a schedule command cannot occur after the *pc* has been tainted. The queuing of messages is by rule T-QUEUE less restrictive, and is akin to rules for sending in previous information flow literature.

We note in particular that Programs 5, 6, and 7 from Section I are typeable by the typing rules, while the rewritten Program 3b from Section III is not as it performs scheduling after branching on a high variable.

B. Program configuration

We show soundness of our security type system in a number of steps. We show that the type system of SELENE is secure against a stronger, internal attacker and show that this implies security against an external attacker. This is intuitively safe as the external attacker has weaker observational power.

We begin by defining well-formedness conditions on memories (Definition 9) and program configurations (Definition 10). These conditions are standard. We define memory m to be well-formed with respect to typing environment Γ in the straight forward way.

Definition 9 (Well-formedness of memory w.r.t. a typing environment)**.** Given a memory m and a typing environment Γ, we say that m is well-formed w.r.t. Γ if for all $x \in \mathsf{dom}(\Gamma)$ we have

(1) $m(x) \in \mathit{Int} \implies \Gamma(x) = \mathbf{int}@\ell$
(2) $m(x) \in \mathit{String} \implies \Gamma(x) = \mathbf{string}_{\ell'}@\ell$

We define program configuration $\langle c, m, I\rangle$ to be well formed with respect to a typing environment Γ and program counters pc, pc' if c is `stop` of if c is typable, and if m is well-formed with respect to Γ.

Definition 10 (Well-formedness of program configurations)**.** We say that program configuration $\langle c, m, I\rangle$ is well-formed w.r.t. a typing environment Γ and levels pc, pc' when both the following hold:

(1) either c is `stop` or the program is well-typed, i.e., $\Gamma, pc \vdash c : pc'$
(2) m is well-formed w.r.t. Γ

Steps of the program preserve well-formedness by Lemma 1. The proof can be found in the accompanying technical report.

$$\textsc{T-Assign}\ \frac{\Gamma \vdash e : \sigma_e@\ell_e \quad \sigma_e \nearrow pc <: \sigma_x \quad \Gamma(x) = \sigma_x@\ell_x \quad \ell_e \sqcup pc \sqsubseteq \ell_x}{\Gamma, pc \vdash x = e : pc}$$

$$\textsc{T-Skip}\ \frac{}{\Gamma, pc \vdash \texttt{skip} : pc}$$

$$\textsc{T-Sleep}\ \frac{\Gamma \vdash e : \mathbf{int}@\ell}{\Gamma, pc \vdash \texttt{sleep}(e) : pc \sqcup \ell}$$

$$\textsc{T-SizeOf}\ \frac{\Gamma \vdash x : \mathbf{int}@\ell_x \quad \Gamma \vdash e : \sigma_e@\ell_e \quad pc \sqsubseteq \ell_x \quad \sigma_e = \mathbf{string}_{\ell'} \implies \ell' \sqsubseteq \ell_x}{\Gamma, pc \vdash x = \texttt{sizeof}(e) : pc}$$

$$\textsc{T-Await}\ \frac{}{\Gamma, pc \vdash \texttt{await}(r) : pc}$$

$$\textsc{T-If}\ \frac{\Gamma \vdash e : \mathbf{int}@\ell \quad \Gamma, pc \sqcup \ell \vdash c_1 : pc' \quad \Gamma, pc \sqcup \ell \vdash c_2 : pc''}{\Gamma, pc \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : pc' \sqcup pc''}$$

$$\textsc{T-Seq}\ \frac{\Gamma, pc \vdash c_1 : pc' \quad \Gamma, pc' \vdash c_2 : pc''}{\Gamma, pc \vdash c_1; c_2 : pc''}$$

$$\textsc{T-While}\ \frac{\Gamma \vdash e : \mathbf{int}@\ell \quad \Gamma, pc \sqcup \ell \vdash c : pc'}{\Gamma, pc \vdash \texttt{while}\ e\ \texttt{do}\ c : pc'}$$

$$\textsc{T-In}\ \frac{\Gamma \vdash x : \sigma_x@\ell_x \quad pc \sqsubseteq \ell \quad \sigma_x \nearrow \ell <: \sigma_x \quad \ell \sqsubseteq \ell_x}{\Gamma, pc \vdash x = \texttt{in}(\ell) : \ell}$$

$$\textsc{T-Schedule}\ \frac{pc = \bot \quad \Gamma \vdash e_1 : \mathbf{int}@\bot \quad \Gamma \vdash e_2 : \mathbf{int}@\bot}{\Gamma, pc \vdash \texttt{schedule}(\ell, e_1, e_2) : pc}$$

$$\textsc{T-Queue}\ \frac{\Gamma \vdash e : \sigma_e@\ell_e \quad \ell_e \sqcup pc \sqsubseteq \ell}{\Gamma, pc \vdash \texttt{queue}(\ell, e) : pc}$$

Figure 8. Type system for commands

Lemma 1 (Preservation of well-formedness)**.** *Let Γ be a typing environment, pc, pc' be two levels, and $\langle c, m, I\rangle$ be a program configuration, such that the $\langle c, m, I\rangle$ is well-formed w.r.t. Γ, pc, and pc'. Suppose this configuration takes a step*

$$\langle c, m, I\rangle \xrightarrow{ts}_{\alpha} \langle c', m', I'\rangle$$

Then there exists pc'' such that $pc \sqsubseteq pc'' \sqsubseteq pc'$ and such that the resulting program configuration $\langle c', m', I'\rangle$ is well-formed w.r.t. Γ, pc'', and pc'.

To reason about what an internal attacker learns from observing a run, we define projection of program events α to level ℓ_{adv} (Definition 11) capturing the attacker observable changes to the internal state of the system. As our model distinguishes between the secrecy levels of the size of a string and its value, we extend the grammar for program events with an event capturing that a string of size s was assigned to variable x.

$$\alpha ::= \ \dots \mid |\mathsf{a}|(x, s)$$

As program event projection is similar to runtime event projection, we use similar notation, but annotate with a bullet to signify that these relate to internal state.

Definition 11 (Program event projection)**.** The projection of program event α to level ℓ_{adv}, written $\lfloor \alpha \rfloor^{\bullet}_{\ell_{adv}}$, is defined as

$$\lfloor \epsilon \rfloor^{\bullet}_{\ell_{adv}} = \epsilon$$

$$\lfloor \mathsf{s}(\ell, n, w) \rfloor^{\bullet}_{\ell_{adv}} = \mathsf{s}(\ell, n, w)$$

$$\lfloor \mathsf{a}(x, v) \rfloor^{\bullet}_{\ell_{adv}} = \begin{cases} \mathsf{a}(x, v) & \text{if } \Gamma(x) = \sigma@\ell \text{ s.t. } \ell \sqsubseteq \ell_{adv} \\ |\mathsf{a}|(x, s) & \text{if } \Gamma(x) = \mathbf{string}_{\ell'}@\ell \\ & \text{s.t. } \ell \not\sqsubseteq \ell_{adv} \wedge \ell' \sqsubseteq \ell_{adv} \\ & \wedge\ size(v) = s \\ \epsilon & \text{otherwise} \end{cases}$$

$$\lfloor \mathsf{q}(\ell, v) \rfloor^{\bullet}_{\ell_{adv}} = \begin{cases} \mathsf{q}(\ell, v) & \text{if } \ell \sqsubseteq \ell_{adv} \\ \epsilon & \text{if } \ell \not\sqsubseteq \ell_{adv} \end{cases}$$

$$\lfloor \mathsf{i}(\ell, x, v) \rfloor^{\bullet}_{\ell_{adv}} = \begin{cases} \mathsf{i}(\ell, x, v) & \text{if } \ell \sqsubseteq \ell_{adv} \\ \epsilon & \text{if } \ell \not\sqsubseteq \ell_{adv} \end{cases}$$

We define internal trace filtering using program event projection in the straight forward way. We again filter out global events where no observable program or runtime events are emitted to prevent the attacker from observing termination.

Definition 12 (Internal trace filtering)**.** The internal filtering of a trace τ at level ℓ_{adv}, written $\tau \upharpoonright^{\bullet} \ell_{adv}$, is defined as

$$\begin{aligned}
\epsilon \upharpoonright^{\bullet} \ell_{adv} &= \epsilon \\
(\tau' \cdot (ts : \alpha, \beta)) \upharpoonright^{\bullet} \ell_{adv} &= \\
&\begin{cases} \tau' \upharpoonright^{\bullet} \ell_{adv} \cdot (ts : \lfloor\alpha\rfloor^{\bullet}_{\ell_{adv}}, \lfloor\beta\rfloor_{\ell_{adv}}) & \text{if } \lfloor\alpha\rfloor^{\bullet}_{\ell_{adv}} \neq \epsilon \\ & \text{or } \lfloor\beta\rfloor_{\ell_{adv}} \neq \epsilon \\ \tau' \upharpoonright^{\bullet} \ell_{adv} & \text{otherwise} \end{cases}
\end{aligned}$$

Using internal trace filtering, we define internal knowledge. This definition mirrors attacker knowledge, except for using internal trace filtering, thereby giving additional power to the attacker.

Definition 13 (Internal knowledge)**.** Given a program configuration P, such that $\langle\!| P, O_{init}, \pi_{init}, 0 |\!\rangle \twoheadrightarrow^*_{\tau} \langle\!| P', O', \pi', ts' |\!\rangle$, internal knowledge at level ℓ_{adv} is the set of program configurations P_2, that are consistent with observations at that level:

$$\begin{aligned}
&k^{\bullet}(P, \tau, \ell_{adv}) \triangleq \\
&\quad \{P_2 \mid P \approx_{\ell_{adv}} P_2 \wedge \\
&\qquad \langle\!| P_2, O_{init}, \pi_{init}, 0 |\!\rangle \twoheadrightarrow^*_{\tau_2} \langle\!| P_2', O_2', \pi_2', ts_2' |\!\rangle \wedge \\
&\qquad (\tau \upharpoonright^{\bullet} \ell_{adv}) = (\tau_2 \upharpoonright^{\bullet} \ell_{adv})\}
\end{aligned}$$

Lemma 2 captures that an internal attacker is indeed stronger than an external attacker by giving internal knowledge as a lower bound on attacker knowledge. This lemma allows us to relate the result we obtain for an internal attacker to the external attacker we consider in our threat model, thereby enabling us to show the type system of SELENE sound with respect to Definition 8. We refer to the accompanying technical report for the proof.

Lemma 2 (Internal knowledge refines external knowledge)**.** *For any program configuration P, trace τ, and level ℓ_{adv}, the knowledge of an external attacker is less precise than the knowledge of an internal attacker. That is,*

$$k(P, \tau, \ell_{adv}) \supseteq k^{\bullet}(P, \tau, \ell_{adv})$$

In defining the knowledge of a network attacker in Section III, we defined equivalence of input environments (Definition 2) by relating only environments whose high channels receive packets at the same timestamps. However, while the network attacker observes incoming packets, they do not observe if or when the packets are consumed by the program internally. For this reason, program steps do not need to preserve equivalence by Definition 2. To relate input environments internally we define internal equivalence (Definition 14) as equality of packet sequences on attacker observable channels.

Definition 14 (Internal input environment equivalence up to level)**.** Two input environments I and I' are internally equivalent up to level ℓ_{adv}, written $I \approx^{\bullet}_{\ell_{adv}} I'$, if

$$\frac{\ell \sqsubseteq \ell_{adv} \implies I_1(\ell) = I_2(\ell)}{I_1 \approx^{\bullet}_{\ell_{adv}} I_2}$$

Definition 14 is strictly weaker than Definition 2, which we state as Lemma 3.

Lemma 3 (Network input equivalence implies internal input equivalence)**.** *For any input environments I_1, I_2, equivalence by Definition 2 implies equivalence by Definition 14. That is,*

$$I_1 \approx_{\ell_{adv}} I_2 \implies I_1 \approx^{\bullet}_{\ell_{adv}} I_2$$

Proof: Immediate from the definitions. ■

We now present our noninterference lemma for program configurations (Lemma 4). It says that given a level ℓ_{adv} and a program configuration that takes a step emitting some event α, then all configurations equivalent at ℓ_{adv} either take a step, emitting an equivalent event α', and are again equivalent at ℓ_{adv}; or the configuration can be typed with a high *pc* and has no observable effects at level ℓ_{adv}.

Lemma 4 (Program step noninterference)**.** *Given a level ℓ_{adv} and a command c and Γ, pc, pc' such that c is well-formed w.r.t Γ, pc, pc', if*

$$\langle c, m_1, I_1 \rangle \xrightarrow{ts}_{\alpha_1} \langle c', m_1', I_1' \rangle$$

then for any memory m_2 such that $m_1 \approx_{\ell_{adv}} m_2$ and input environment I_2 such that $I_1 \approx^{\bullet}_{\ell_{adv}} I_2$, we have that one of the following holds

1) *either $\langle c, m_2, I_2 \rangle \xrightarrow{ts}_{\alpha_2} \langle c', m_2', I_2' \rangle$ such that $\lfloor\alpha_1\rfloor^{\bullet}_{\ell_{adv}} = \lfloor\alpha_2\rfloor^{\bullet}_{\ell_{adv}}$ and $m_1' \approx_{\ell_{adv}} m_2'$ and $I_1' \approx^{\bullet}_{\ell_{adv}} I_2'$.*
2) *or $\lfloor\alpha_1\rfloor^{\bullet}_{\ell_{adv}} = \epsilon$ and there is pc'' such that $pc'' \not\sqsubseteq \ell_{adv}$ and $\Gamma, pc'' \vdash c : pc'$ and $m_1 \approx_{\ell_{adv}} m_1'$ and $I_1 \approx^{\bullet}_{\ell_{adv}} I_1'$*

We refer the interested reader to the accompanying technical report for the proof and supporting lemmas.

C. Global configuration

We take the next step towards showing soundness of the type system by using Lemma 4 to show single-step noninterference of global configurations. Lemma 5 tells us that for any level ℓ_{adv} and any two equivalent runs, if one run takes a step producing global event γ, then the other run also takes an equivalent step, or the two runs diverge with high *pc*. We do not require that all events emitted after divergence are unobservable to the adversary. Instead, we allow runtime events to still be observable. This is intuitively safe as the runtime behaviour is fully determined by the program events, and no observable program events may be emitted after divergence. To simplify reasoning for divergent runs, we use a configuration with command `stop` in the second condition. This configuration serves as an anchor, allowing us to reason about the observable behaviour of any two equivalent configurations by showing they behave the same as the `stop`-configuration. To satisfy this condition, the configuration must produce no observable program event or changes to memory or input environment, and produces output if and only if the `stop`-configuration produces equivalent output.

Lemma 5 (Global step noninterference)**.** *Given a level ℓ_{adv} and a command c and Γ, pc, pc' such that c is well-formed w.r.t Γ, pc, pc', if*

$$\langle\!\langle \langle c, m_1, I_1 \rangle, O_1, \pi, ts \rangle\!\rangle \twoheadrightarrow_{(ts:\alpha_1,\beta_1)} \langle\!\langle \langle c', m_1', I_1' \rangle, O_1', \pi', ts' \rangle\!\rangle$$

then for any memory m_2 such that $m_1 \approx_{\ell_{adv}} m_2$; input environment I_2 such that $I_1 \approx^{\bullet}_{\ell_{adv}} I_2$; and output environment O_2 such that $O_1 \approx_{\ell_{adv}} O_2$, we have that one of the following holds

1) either

$$\langle\!\langle \langle c, m_2, I_2 \rangle, O_2, \pi, ts \rangle\!\rangle \twoheadrightarrow_{(ts:\alpha_2,\beta_2)} \langle\!\langle \langle c', m_2', I_2' \rangle, O_2', \pi', ts' \rangle\!\rangle$$

such that each of the following hold

a) $\lfloor \alpha_1 \rfloor^{\bullet}_{\ell_{adv}} = \lfloor \alpha_2 \rfloor^{\bullet}_{\ell_{adv}}$
b) $\lfloor \beta_1 \rfloor_{\ell_{adv}} = \lfloor \beta_2 \rfloor_{\ell_{adv}}$
c) $m_1' \approx_{\ell_{adv}} m_2'$
d) $I_1' \approx^{\bullet}_{\ell_{adv}} I_2'$
e) $O_1' \approx_{\ell_{adv}} O_2'$

2) or there is pc'' such that $pc'' \not\sqsubseteq \ell_{adv}$ and c is well-formed w.r.t Γ, pc'', pc', and such that each of the following hold

a) $\lfloor \alpha_1 \rfloor^{\bullet}_{\ell_{adv}} = \epsilon$
b) $m_1 \approx_{\ell_{adv}} m_1'$
c) $I_1 \approx^{\bullet}_{\ell_{adv}} I_1'$
d) $\pi = \pi'$
e) $\beta_1 \neq \epsilon$ *if and only if there are* β_2 *and* O_2' *such that* $\lfloor \beta_1 \rfloor_{\ell_{adv}} = \lfloor \beta_2 \rfloor_{\ell_{adv}}$, $O_1' \approx_{\ell_{adv}} O_2'$, *and*

$$\langle\!\langle \langle \texttt{stop}, m_2, I_2 \rangle, O_2, \pi, ts \rangle\!\rangle \twoheadrightarrow_{(ts:\epsilon,\beta_2)} \langle\!\langle \langle \texttt{stop}, m_2, I_2 \rangle, O_2', \pi, ts' \rangle\!\rangle$$

We again refer to the accompanying technical report for the proof.

D. Soundness of security type system

Using the above results, we are now ready to state our soundness theorem (Theorem 1). It says that any well-typed program satisfies timing-sensitive, progress-sensitive noninterference (Definition 8).

Theorem 1 (Soundness)**.** *Given a typing environment Γ, two levels pc, pc', and a program configuration P that is well-formed w.r.t Γ, pc, pc', the run $\langle\!\langle P, O_{init}, \pi_{init}, 0 \rangle\!\rangle \twoheadrightarrow^{*}_{\tau} \langle\!\langle P', O', \pi', ts' \rangle\!\rangle$ satisfies Definition 8.*

The proof is by induction in the number of steps and uses Lemma 5 to infer that if the run emits an event observable by an internal attacker, then all runs of equivalent program configurations must as well, hence the internal attacker does not learn anything. Using Lemma 2 we conclude that the network attacker equally does not learn anything. The full proof of Theorem 1 is omitted here and can be found in the accompanying technical report.

As noted in Section IV, Programs 5, 6, and 7 from Section I are typeable by the typing rules and hence by Theorem 1 we obtain a proof that they satisfy Definition 8 and do not leak by their output behaviour.

V. Discussion

A. Publicly observable traffic

By our chosen strategy and the assumption that network activity can be eavesdropped, we arrive at a number of restrictions on how output channels can be used. Our strategy does not permit the scheduling of new messages after the program counter has been raised. Nevertheless, Program 7 in Section I demonstrates how dynamic scheduling is possible after receiving input, provided the input is received on a non-secret channel. This allows lower bandwidth overheads compared with constant rate padding schemes, by only scheduling traffic when needed. While it is unsurprising that it is safe to only schedule as needed while in a low context, this intuitive fact is difficult to prove correct without a principled, formal model like we present in this paper.

B. Limitations and future work

In this paper we have opted for simple and explicit packet scheduling via programmer written commands, allowing us to use the IFC system to prevent leaks from both message contents and message presence. Our model and the primitives presented are not intended as a full solution for preventing traffic analysis attacks, but rather aim to bring attention to an as yet unsolved problem, and serve as a step towards providing practical and provably safe usage of channels susceptible to eavesdropping. A significant limitation of our model is that the progress-sensitive nature of the type system makes composition of programs difficult. A *pc*-declassification mechanism would alleviate this issue, but the security impact of allowing such mechanism must be fully understood. To focus our model, we have deliberately not included a *pc*-declassification primitive in the language.

Other strategies for setting up packet schedules may be viable. In particular, we note that patterns in publicly observable input traffic may be used when deciding the shape of output traffic. However, we note that for receives with public blocking behaviour, the employed strategy should not incur significant overheads or hinder the ability to reply. As such, static preprocessing of a target program to determine a packet schedule is not viable.

VI. Related work

Sabelfeld and Mantel [30] also consider the problem of sending secret messages over publicly observable channels as part of a distributed program. They consider concurrent programs that communicate over low channels that are fully observable; encrypted channels where the number of messages is observable, but size and contents is not; and high channels where both message presence and contents are secret. They define a timing-sensitive security definition using strong low-bisimulation, requiring that the number of encrypted messages sent is the same between any two related runs in lockstep. The channels they consider model communication with specific endpoints at nodes – rather than with nodes themselves – and they do not contain dummy messages. Consequently, the blocking behaviour of receives on encrypted channels is public, and encrypted channels in their work do not correspond with non-public channels of our work whose blocking behaviour is non-public. The authors discuss the practical implications of different communication primitives. They argue that receives on channels that exhibit secret blocking behaviour is not secure and non-blocking receives should be used instead, while receives on channels that exhibit public blocking behaviour should use blocking receives to prevent busy waiting.

Zhang et al. [36] propose a general language-based mechanism for controlling timing channels based on the idea of predictive migitation [6]. While both their approach and ours rely on the idea of scheduling observable events, they are orthogonal. A distinguishing property of the predictive mitigation is that because of its generality the only allowed modification to program semantics is delaying of the messages. In contrast, our approach – where we focus on the network attacker – allows us to use dummy messages, preventing the delays caused by mispredictions.

As noted in Section V, a *pc*-declassification mechanism could be used to alleviate some of the restrictions imposed by SELENE's progress-sensitive type system. Bay and Askarov [9] give a formal condition on how much attacker knowledge is allowed to change as a result of *pc*-declassification by bounding it using the so-called progress-knowledge. We leave adapting their approach to SELENE as future work. Vassena et al. [33] propose a dynamic language-level IFC system that supports deterministic parallel thread execution. Such a system could retain a public context thread, potentially mitigating the need for explicit *pc*-declassification.

Oblivious programming languages such as ObliVM [24] and Obliv-C [35] allow programmers to write protocols for secure computations, where multiple parties can perform computation collaboratively without revealing their input via produced trace, e.g. instructions, memory accesses, and values of public variables. To achieve security, such languages commonly simulate the execution of non-chosen branches in conditional statements and publicly bound and pad the number of loop iterations and the number of bits needed to represent secret values. While the goal of oblivious programming languages overlaps with ours at a high level, care is needed for adapting the techniques to our model. Generally speaking, these languages do not allow loop guards or blocking behaviour to be non-public. Our model allows the size of network messages to be kept secret by sending them as a series of (potentially dummy) packets. Consequently, the blocking behaviour of the receive primitive for non-public channels in SELENE is inherently non-public. A solution used in the oblivious approach is to tag values with a public, conservative upper bound on their size. This gives weaker confidentiality, but if acceptable appears a viable solution for the problem we discuss in this paper and we leave application of oblivious programming techniques as future work.

Previous work has examined the possibility of traffic analysis attacks revealing sensitive user information and actions across various settings.

Browsers: Chen et al. [11] and Miller et al. [26] both consider the traffic patterns generated by user interactions on webpages. Miller et al. present an attack against the HTTPS deployments of industry-leading websites spanning multiple sectors. Their attack was able to identify pages within a site with high accuracy, exposing personal details including medical conditions and financial affairs. They propose a defence mechanism that pads contiguous bursts of traffic up to per-website, predefined thresholds. Their analysis shows that the proposed defence mechanism outperforms site-agnostic approaches that pad the sizes of all packets to global, nearest threshold values. Chen et al. find that the potential for traffic analysis attacks is exacerbated by design features for dynamic, reactive websites such as AJAX GUI widgets, which often generate distinctive traffic in response to user interactions.

Cherubin et al. [12] consider website fingerprinting defences at the application layer and introduce ALPaCA, a server side defence for use with Tor. ALPaCA works by transforming site content to conform to average site content, as analysed across multiple Tor sites. Their analysis shows that ALPaCA reduces website fingerprinting accuracy from 69.6% to 10%.

Phones and apps: Conti et al. [15] and Wang et al. [34] both consider attacks on users of Android smartphones. Conti et al. present a machine learning assisted traffic analysis attack that infers user actions in apps with high precision and high recall, e.g., opening a profile page on Facebook or posting a message on Twitter. Wang et al. present a packet level attack on encrypted Android traffic. By collecting and analysing a small amount of wireless traffic, they are able to determine which apps smartphone users are using. Their analysis shows that apps are more susceptible to traffic analysis attacks than online services accessed over browsers, as apps tend to generate more distinct patterns of traffic.

Bahramali et al. [7] show that also instant messaging clients are susceptible to traffic analysis attacks despite using state-of-the-art encryption. They demonstrate an attack capable of identifying members and administrators of IM channels with high accuracy, using only low-cost traffic analysis techniques. They attribute this to the fact that major IM operators do not use mechanisms for obfuscating genuine traffic, arguing their reluctance is due to the performance and usability impact of deploying such techniques.

In the home: Zhang et al. [37] present an attack for inferring user activities by eavesdropping on WLAN traffic. They consider online activities such as web browsing, chatting, gaming, and watching videos. They use a hierarchical classification system based on machine learning algorithms and show that their system can distinguish different online applications with roughly 80% accuracy when given 5 seconds of traffic, and roughly 90% accuracy when given 1 minute of traffic.

Apthorpe et al. [1] consider home IoT devices and attacks inferring when a device is used, thereby revealing sensitive user information such as sleep patterns and when the user is home. They introduce stochastic traffic padding, which decreases attacker confidence by uniformly shaping upload and download traffic during user activities, and injecting equivalent traffic patterns at random times to hide when the device is in use.

VII. Conclusion

In this paper we consider language-based mitigation of traffic analysis attacks. We observe four traits on messages sent that may leak secret information, namely presence, recipient, size, and time. This observation informed the design of SELENE, a small imperative language for interactive programs. The type system of SELENE enforces principled, provably secure communication over channels where packets are publicly observable. The key insight of the language is a novel primitive that provides programmatic control over traffic shaping thereby allowing for reduced overheads in latency and bandwidth compared with black box techniques. We give a formal, timing-sensitive, progress-sensitive security condition based on the knowledge-based approach and prove our type system sound. We believe that our model faithfully captures online communication constraints, and that our results constitute a step towards practical, secure online communication. We believe the security risks of traffic analysis attacks against confidentiality are significant and that work on language-based information flow for interactive programs must be mindful of the assumptions being made about the security of communications channels. We welcome and encourage future work to explore language-based techniques for providing strong security guarantees against traffic analysis attacks.

VIII. Acknowledgements

We thank Alix Trieu for his comments and feedback and thank the anonymous reviewers for their valuable suggestions for improving the presentation of this paper. This work was funded by the Danish Council Independent Research for the Natural Sciences (DFF/FNU, project 6108-00363).

References

[1] N. J. Apthorpe, D. Y. Huang, D. Reisman, A. Narayanan, and N. Feamster, "Keeping the smart home private with smart(er) iot traffic shaping," *CoRR*, vol. abs/1812.00955, 2018. [Online]. Available: http://arxiv.org/abs/1812.00955

[2] A. Askarov and S. Chong, "Learning is change in knowledge: Knowledge-based security for dynamic policies," *2012 IEEE 25th Computer Security Foundations Symposium*, pp. 308–322, 2012.

[3] A. Askarov and A. Sabelfeld, "Tight enforcement of information-release policies for dynamic languages," in *2009 22nd IEEE Computer Security Foundations Symposium*, 2009, pp. 43–59.

[4] A. Askarov and A. C. Myers, "Attacker control and impact for confidentiality and integrity," *Logical Methods in Computer Science*, vol. 7, no. 3, 2011. [Online]. Available: https://doi.org/10.2168/LMCS-7(3:17)2011

[5] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *2007 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 2007, pp. 207–221.

[6] 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*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 297–307. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866341

[7] A. Bahramali, A. Houmansadr, R. Soltani, D. Goeckel, and D. Towsley, "Practical traffic analysis attacks on secure messaging applications," *Proceedings 2020 Network and Distributed System Security Symposium*, 2020. [Online]. Available: http://dx.doi.org/10.14722/ndss.2020.24347

[8] I. Bastys, M. Balliu, T. Rezk, and A. Sabelfeld, "Clockwork: Tracking remote timing attacks," in *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, 2020, pp. 350–365.

[9] J. Bay and A. Askarov, "Reconciling progress-insensitive noninterference and declassification," in *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*. IEEE, 2020, pp. 95–106.

[10] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, "Reactive noninterference," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 79–90. [Online]. Available: https://doi.org/10.1145/1653662.1653673

[11] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. USA: IEEE Computer Society, 2010, p. 191–206. [Online]. Available: https://doi.org/10.1109/SP.2010.20

[12] G. Cherubin, J. Hayes, and M. Juárez, "Website fingerprinting defenses at the application layer," *PoPETs*, vol. 2017, no. 2, pp. 186–203, 2017. [Online]. Available: https://doi.org/10.1515/popets-2017-0023

[13] D. Clark and S. Hunt, "Non-interference for deterministic interactive programs," in *Formal Aspects in Security and Trust: 5th International Workshop, FAST 2008 Malaga, Spain, October 9-10, 2008 Revised Selected Papers*, 04 2009, pp. 50–66.

[14] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, pp. 1157–1210, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1891823.1891830

[15] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde, "Analyzing android encrypted network traffic to identify user actions," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 1, pp. 114–125, 2016.

[16] D. Das, S. Meiser, E. Mohammadi, and A. Kate, "Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency - choose two," *IACR Cryptology ePrint Archive*, vol. 2017, p. 954, 2017.

[17] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton, "Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. USA: IEEE Computer Society, 2012, p. 332–346. [Online]. Available: https://doi.org/10.1109/SP.2012.28

[18] J. Feigenbaum, A. Johnson, and P. Syverson, "Preventing active timing attacks in low-latency anonymous communication," in *Privacy Enhancing Technologies*. Berlin, Heidelberg: Springer Berlin Heidelberg, 07 2010, pp. 166–183.

[19] X. Fu, B. Graham, R. Bettati, W. Zhao, and D. Xuan, "Analytical and empirical analysis of countermeasures to traffic analysis attacks," in *2003 International Conference on Parallel Processing, 2003. Proceedings.*, 11 2003, pp. 483 – 492.

[20] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy*, 1982, pp. 11–11.

[21] D. Hedin and A. Sabelfeld, "A perspective on information-flow control," in *Software Safety and Security*, 2012.

[22] M. Juárez, M. Imani, M. Perry, C. Díaz, and M. Wright, "WTF-PAD: toward an efficient website fingerprinting defense for tor," *CoRR*, vol. abs/1512.00524, 2015. [Online]. Available: http://arxiv.org/abs/1512.00524

[23] A. Kwon, M. AlSabah, D. Lazar, M. Dacier, and S. Devadas, "Circuit fingerprinting attacks: Passive deanonymization of tor hidden services," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 287–302. [Online]. Available: http://dl.acm.org/citation.cfm?id=2831143.2831162

[24] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 359–376.

[25] H. Mantel and A. Sabelfeld, "A unifying approach to the security of distributed and multi-threaded programs," *J. Comput. Secur.*, vol. 11, no. 4, p. 615–676, Jul. 2003.

[26] B. Miller, L. Huang, A. D. Joseph, and J. D. Tygar, "I know why you went to the clinic: Risks and realization of https traffic analysis," in *Privacy Enhancing Technologies*, E. De Cristofaro and S. J. Murdoch, Eds. Cham: Springer International Publishing, 2014, pp. 143–163.

[27] K. R. O'Neill, M. R. Clarkson, and S. Chong, "Information-flow security for interactive programs," in *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, ser. CSFW '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 190–201. [Online]. Available: https://doi.org/10.1109/CSFW.2006.16

[28] R. Overdorf, M. Juárez, G. Acar, R. Greenstadt, and C. Díaz, "How unique is your .onion? an analysis of the fingerprintability of tor onion services," *CoRR*, vol. abs/1708.08475, 2017. [Online]. Available: http://arxiv.org/abs/1708.08475

[29] A. Panchenko, F. Lanze, A. Zinnen, M. Henze, J. Pennekamp, K. Wehrle, and T. Engel, "Fingerprinting at internet scale," in *Proceedings of the 23rd Internet Society (ISOC) Network and Distributed System Security Symposium (NDSS 2016)*, 2015.

[30] A. Sabelfeld and H. Mantel, "Static confidentiality enforcement for distributed programs," in *Static Analysis*, M. V. Hermenegildo and G. Puebla, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 376–394.

[31] D. Schoepe and A. Sabelfeld, "Understanding and enforcing opacity," in *Proceedings of the 2015 IEEE 28th Computer Security Foundations Symposium*, ser. CSF '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 539–553. [Online]. Available: https://doi.org/10.1109/CSF.2015.41

[32] S. Siby, M. Juárez, N. Vallina-Rodriguez, and C. Troncoso, "Dns privacy not so private: the traffic analysis perspective," 2018.

[33] M. Vassena, G. Soeller, P. Amidon, M. Chan, J. Renner, and D. Stefan, "Foundations for parallel information flow control runtime systems," in *Principles of Security and Trust*, F. Nielson and D. Sands, Eds. Cham: Springer International Publishing, 2019, pp. 1–28.

[34] Q. Wang, A. Yahyavi, B. Kemme, and W. He, "I know what you did on your smartphone: Inferring app usage over encrypted data traffic," in *2015 IEEE Conference on Communications and Network Security (CNS)*, 2015, pp. 433–441.

[35] S. Zahur and D. Evans, "Obliv-c: A language for extensible data-oblivious computation," Cryptology ePrint Archive, Report 2015/1153, 2015, https://eprint.iacr.org/2015/1153.

[36] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, 2012, pp. 99–110. [Online]. Available: https://doi.org/10.1145/2254064.2254078

[37] F. Zhang, W. He, X. Liu, and P. G. Bridges, "Inferring users' online activities through traffic analysis," in *Proceedings of the Fourth ACM Conference on Wireless Network Security*, ser. WiSec '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 59–70. [Online]. Available: https://doi.org/10.1145/1998412.1998425