http://www.diva-portal.org

Postprint

This is the accepted version of a paper presented at *IEEE Computer Security Foundations Symposium (CSF 2020)*.

Citation for the original published paper:

Bastys, I., Balliu, M., Rezk, T., Sabelfeld, A. (2020)
Clockwork: Tracking Remote Timing Attacks
In:

N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-273162

Clockwork: Tracking Remote Timing Attacks

Iulia Bastys* Musard Balliu† Tamara Rezk‡ Andrei Sabelfeld*
*Chalmers University of Technology †KTH Royal Institute of Technology ‡INRIA Sophia-Antipolis

Abstract—**Timing leaks have been a major concern for the security community. A common approach is to prevent secrets from affecting the execution time, thus achieving security with respect to a strong, *local* attacker who can measure the timing of program runs. However, this approach becomes restrictive as soon as programs branch on a secret.**

This paper focuses on timing leaks under *remote* execution. A key difference is that the remote attacker does not have a reference point of when a program run has started or finished, which significantly restricts attacker capabilities. We propose an extensional security characterization that captures the essence of remote timing attacks. We identify patterns of combining clock access, secret branching, and output in a way that leads to timing leaks. Based on these patterns, we design Clockwork, a monitor that rules out remote timing leaks. We implement the approach for JavaScript, leveraging JSFlow, a state-of-the-art information flow tracker. We demonstrate the feasibility of the approach on case studies with IFTTT, a popular IoT app platform, and VJSC, an advanced JavaScript library for e-voting.

1. Introduction

The security community has extensively studied timing leaks, from investigating their foundations [1], [4], [7], [14], [33], [44], [48]–[50], [62] to analyzing them in practice [19], [29], [33], [52]. Timing attacks that exploit speculative execution [43] have recently received particular attention.

Restrictions to deal with timing attacks A common approach is to prevent secrets from affecting the execution time, thus achieving security with respect to a strong, *local* attacker who can measure the timing of program runs. At the very least, the local attacker observes time at the start and end of computation, while some local attacker models observe time before and after each operation as well as the full program-counter trace [1], [4], [51]. This approach is popular in cryptography, where timing leaks are often closed by *constant-time execution* (e.g., [3], [4], [18], [36]). There are several constant-time execution implementations of cryptographic algorithms, including AES, DES, RC4, SHA256, and RSA. Another approach is to allow branching on secrets but prohibit any subsequent attacker-visible side effects of the program [20], [55]. This approach is effective with respect to so-called *internal timing* leaks [48], where the timing behavior of threads affects the interleaving of attacker-visible events via the scheduler.

While these approaches tackle strong attackers, they are restrictive as soon as programs branch on a secret. Indeed, "adhering to constant-time programming is hard" and "doing so requires the use of low-level programming languages or compiler knowledge, and forces developers to deviate from conventional programming practices" [4].

The problem is challenging because there are many ways to set up timing leaks in a program. For example, after branching on a secret the program might take different time in the branches because of: (i) more time-consuming operations in one of the branches [1], [51], (ii) cache effects, when in one of the branches data or instructions are cached but not in the other branch [4], [32], (iii) garbage collection (GC), when in one of the branches GC is triggered but not in the other branch [46], and (iv) just-in-time (JIT) compilation, when in one of the branches a JIT-compiled function is called but not in the other branch [21]. Researchers have been painstakingly addressing these types of leaks, often by creating mechanisms that are specific to some of these types [1], [4], [21], [32], [46], [51]. Because of the intricacies of each type, addressing their combination without ending up with a severely restrictive mechanism poses a major challenge (see Section 7).

This motivates a general mechanism to tackle timing leaks independently of their type. However, rather than combining mechanisms for the different types of timing leaks for strong local attackers, is there a setting where the capabilities of attackers are perhaps not as strong, enabling us to design a general yet less restrictive mechanism?

Remote timing attacks This paper focuses on timing leaks under *remote* execution. A key difference is that the remote attacker does not have a reference point of when a program run has started or finished. This significantly restricts attacker capabilities.

We illustrate remote timing attacks by two settings: a server-side setting of IoT apps where apps that manipulate private information run on a server, and a client-side setting where e-voting code runs in a browser.

IFTTT [42] (If This Then That), Zapier, and Microsoft Power Automate are popular IoT platforms driven by enduser programming. App makers publish their apps on these platforms. Upon installation apps manipulate user sensitive information, connecting cyberphysical "things" (e.g., smart homes, cars, and fitness armbands) to online services (e.g., Google and Dropbox) and social networks (e.g., Facebook and Twitter). An important security goal is to prevent a

malicious app from leaking user private information to the attacker.

Recent research [10], [16], [17], [25], [26], [34], [58] identifies ways to leak private information by malicious IoT apps and suggests information flow tracking as countermeasure. The suggested mechanisms perform data-flow (*explicit* [30]) and control-flow (*implicit* [30]) tracking. Unfortunately, they fall short of addressing timing leaks. Thus, a malicious app can still exfiltrate private information, even if the app is free of explicit and implicit flows.

The Verificatum JavaScript Cryptographic library (VJSC) [31] is an advanced client-side cryptographic library for e-voting. This library motivates the question of remote timing leaks with respect to attackers who can observe the presence of encrypted messages on the network.

This leads us to the following general research questions: (i) What is the right model for remote timing attacks? (ii) How do we rule out remote timing leaks without rejecting useful secure programs? (iii) How do we generalize our findings to programs that manipulate information at multiple levels of sensitivity beyond just private and public? (iv) How do we harden existing information flow tools to track remote timing leaks? (v) Are there case studies to give evidence for the feasibility of the approach?

Contributions To help answering these questions, we propose an extensional knowledge-based security characterization that captures the essence of remote timing attacks. In contrast to the local attacker that counts execution steps/time since the beginning of the execution, our model of the remote attacker is only allowed to observe communication events on attacker-visible channels, along with their timestamps. At the same time, the attacker is in charge of the potentially malicious code with capabilities to access the clock, in line with assumptions about remote execution on IoT app platforms and e-voting clients.

A timing leak is typically enabled by branching on a secret and taking different time in the branches. The branches might run different sequences of commands and/or exhibit different cache behavior. As discussed earlier, it is desirable to avoid such restrictive alternatives as forcing constant-time execution, prohibiting attacker-visible output any time after the branching, or prohibiting branching on a secret in the first place.

Our key observation is that for a remote attacker to successfully exploit a timing leak in an explicit and implicit flow-free program, the program behavior must follow the following pattern: (i) branching on a secret takes place in a program run, and either (ii-a) the branching is followed by more than one attacker-visible I/O event, or (ii-b) the branching is followed by one attacker-visible I/O event and prior to the branching there is either an attacker-visible I/O event, or a clock read.

Based on this pattern, we design Clockwork, a monitor that rules out timing leaks and pushes for permissiveness. Among runs that are free of explicit and implicit flows, runs that do not access the clock and only have one attacker-visible I/O event are accepted. Runs that do not perform attacker-visible I/O after branching on a secret are also accepted. As we will see, these kinds of runs are frequently encountered in both secure IoT and e-voting apps.

We implement our monitor for JavaScript, leveraging JSFlow [39]–[41], a state-of-the-art information flow tracker. We demonstrate the feasibility of the approach on a case study with IFTTT, showing how to prevent malicious app makers from exfiltrating users' private information via timing, and a case study with VJSC, showing how to track remote timing attacks with respect to network attackers. Our case studies demonstrate both the security and permissiveness of the approach. While apps with timing leaks are rejected, benign apps that use clock and I/O operations in a non-trivial fashion are accepted.

In summary, the paper offers the following contributions with respect to the above research questions:

(i) We present a general framework to reason about remote timing leaks and provide a knowledge-based security characterization. This characterization incorporates such novel aspects as existentially quantifying over time points when the computation has started and reasoning about timeouts (Section 2).

(ii) We design a flexible and sound security enforcement mechanism to rule out timing leaks in a simple imperative language by tracking clock access, secret branching, and public output. The mechanism is parametric in a variety of cache models (Section 3).

(iii) We generalize the approach to multiple levels of sensitivity beyond private and public (Section 4).

(iv) We implement our enforcement on top of JSFlow, a state-of-the-art information flow tracker for JavaScript (Section 5).

(v) We present case studies with IFTTT, a popular IoT app platform, and VJSC, an advanced cryptographic library for e-voting, preventing timing leaks without being overly restrictive (Section 6).

2. Security characterization

This section presents the attacker model, the syntax and semantics of the underlying language, and the knowledge-based security characterization.

2.1. Attacker model

We assume a remote attacker able to write programs and publish them on a cloud service, for example an IoT app maker who creates an app and publishes it on the IoT app platform. After installation, the (malicious) app will execute whenever triggered (such as upon taking a photo or parking a car). Note that the attacker does not have a reference point of when the program run has started or finished. However, by observing the outputs sent on attacker-visible channels and by analyzing the timestamps of these outputs, the attacker may aim to infer some information about the sensitive data (e.g., attempting to leak the secret photo URL or the GPS coordinates of the car).

2.2. Language

We consider a simple imperative language extended with instructions for clock reading and for sending output on

$$
\begin{array}{lll}
v & ::= & n \mid s \\
e & ::= & v \mid x \mid f(e_1, \ldots, e_n) \\
c & ::= & \texttt{stop} \mid \texttt{end} \mid x = e \mid c;c \mid \\
 & & \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \texttt{do}\ c \mid \\
 & & x\ \texttt{getsTime} \mid \texttt{out}_\ell(e)
\end{array}
$$

Figure 1: Syntax

different channels.

Syntax Values v consist of strings s and integers n. Expressions e consist of values v, variables x, and n-ary operations $f(e_1, \ldots, e_n)$. Most commands c are standard. Non-standard ones are end for marking the termination of a control flow statement, x getsTime for clock reading and timestamp writing to variable x, and $\texttt{out}_\ell(e)$ for outputting the value of expression e on channel ℓ.

Semantics We assume the memory m to be a mapping from program variables to values. We write $m[x \mapsto v]$ to denote the memory that maps program variable x to value v, while all other mappings are the same as in memory m. We write $\langle e, m\rangle \Downarrow v$ to denote that expression e evaluates to value v in memory m.

A configuration is a tuple $\langle c, m, hst\rangle$ consisting of command c, memory m, and history hst. History hst records the commands previously executed up to the point of inspection. The history may contain events $\texttt{asn}(x, e)$ for assigning expression e to variable x, $\texttt{br}(e)$ for conditional branching on expression e, join for reaching a join point, and $\texttt{o}(e, \ell)$ for outputting expression e on channel ℓ.

Figure 2 defines semantic rules $\langle c, m, hst\rangle \xrightarrow{o} \langle c', m', hst'\rangle$ for producing output o while taking a step from program c in memory m and current history hst to a new configuration $\langle c', m', hst'\rangle$. With the exception of rule SEQ-2, semantic rules may add events to the history sequence: rules ASSIGN and TIME event $\texttt{asn}(x, e)$, rules IF and WHILE event $\texttt{br}(e)$, rule END event join, and rule OUTPUT event $\texttt{o}(e, \ell)$. We briefly discuss rules IF, TIME and OUTPUT, as the other ones are mostly standard.

Rule IF describes the execution of conditional statement if e then c_1 else c_2. After performing a step, the conditional ends up in the sequential execution of branch c_1 or c_2 and end, where end marks that the control flow region has ended. Having an explicit indication of reaching a join point is useful for building a security monitor on top of our semantics.

Our language allows for clock invocations via command x getsTime in rule TIME. Timestamp t is computed by applying function $stmp()$ (for "timestamp") to the current history hst, explained below.

To express the time of observing messages, we model outputs as pairs consisting of the actual value v of expression e to be output and the time t when the output took place. Additionally, we label each output with the label ℓ of the channel on which it is sent (rule OUTPUT).

Generic time model for cache Our goal is to address the effect of cache behavior on the execution time for programs in our language and to do so for a variety of cache models.

Big-step semantics for expressions:

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

$$
\frac{\langle e_i, m\rangle \Downarrow v_i \quad i = 1, \ldots, n \qquad \langle f(v_1, \ldots, v_n), m\rangle \Downarrow v}{\langle f(e_1, \ldots, e_n), m\rangle \Downarrow v}
$$

Small-step semantics for commands with history:

ASSIGN
$$
\frac{\langle e, m, hst\rangle \Downarrow v}{\langle x = e, m, hst\rangle \to \langle \texttt{stop}, m[x \mapsto v], hst :: \texttt{asn}(x, e)\rangle}
$$

SEQ-1
$$
\frac{\langle c_1, m, hst\rangle \xrightarrow{o} \langle c_1', m', hst'\rangle}{\langle c_1; c_2, m, hst\rangle \xrightarrow{o} \langle c_1'; c_2, m', hst'\rangle}
$$

SEQ-2
$$
\frac{}{\langle \texttt{stop}; c, m, hst\rangle \to \langle c, m, hst\rangle}
$$

END
$$
\frac{}{\langle \texttt{end}, m, hst\rangle \to \langle \texttt{stop}, m, hst :: \texttt{join}\rangle}
$$

IF
$$
\frac{\langle e, m\rangle \Downarrow v \qquad v \neq 0 \Rightarrow i = 1 \qquad v = 0 \Rightarrow i = 2}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m, hst\rangle \to \langle c_i; \texttt{end}, m, hst :: \texttt{br}(e)\rangle}
$$

WHILE
$$
\frac{\langle e, m\rangle \Downarrow v \qquad v \neq 0 \Rightarrow c' = c; \texttt{while}\ e\ \texttt{do}\ c \qquad v = 0 \Rightarrow c' = \texttt{stop}}{\langle \texttt{while}\ e\ \texttt{do}\ c, m, hst\rangle \to \langle c'; \texttt{end}, m, hst :: \texttt{br}(e)\rangle}
$$

TIME
$$
\frac{t = stmp(hst)}{\langle x\ \texttt{getsTime}, m, hst\rangle \to \langle \texttt{stop}, m[x \mapsto t], hst :: \texttt{asn}(x, t)\rangle}
$$

OUTPUT
$$
\frac{\langle e, m\rangle \Downarrow v \qquad t = stmp(hst :: \texttt{o}(e, \ell))}{\langle \texttt{out}_\ell(e), m, hst\rangle \xrightarrow{(v,t)_\ell} \langle \texttt{stop}, m, hst :: \texttt{o}(e, \ell)\rangle}
$$

Figure 2: Semantics

Yet instead of defining a generic model of cache itself (which would include a detailed memory representation of how addresses are accessed with respect to data and instruction cache) we focus directly on the possible time effects of data and instruction cache.

The advantage of this approach is that we can offer a time model representative for a wide class of cache implementations [4]. Our only assumption is that cache (and thus execution time) may depend on the computation history. The history is expressed by sequences of command events recording which commands are run. Variables (whose memory addresses are fixed as we do not have a heap in

NO-TO

$$\frac{\langle c, m_0, t_0\rangle \overset{O}{\rightarrow}^* \langle c', m, hst\rangle \qquad stmp(hst) \leq timeout}{\langle c, m_0, t_0\rangle \overset{O}{\Rightarrow}^* \langle c', m, hst\rangle}$$

TO

$$\frac{\begin{array}{c}\langle c, m_0, t_0\rangle \overset{O}{\Rightarrow}^* \langle c', m, hst\rangle \\ (\langle c', m, hst\rangle \rightarrow \langle _, _, hst'\rangle \wedge stmp(hst') > timeout \\ \vee \\ c' \neq \texttt{stop} \wedge \langle c', m, hst\rangle \not\rightarrow \langle _, _, _\rangle)\end{array}}{\langle c, m_0, t_0\rangle \Uparrow O}$$

Figure 3: Top level rules

our language) accessed on both reads and writes are also recorded in the history. This can be seen from rule ASSIGN in Figure 2 which upon an assignment $x = e$ records the command in the history through event $\texttt{asn}(x, e)$. This allows modeling cache-related time differences as we will see in the examples in Figure 5.

Thus, function $stmp()$ operates on the sequence of events recorded since the start of the program. The program executes from an initial configuration which contains in place of the history a timestamp t_0 denoting the time when the program has started. The initial timestamp is used for computing further timestamps by applying $stmp()$ to the current history.

Note that our semantics is parametric in function $stmp()$. Our only assumption on $stmp()$ is that it is a *strictly increasing* function mapping histories to a numeric domain representing real time, so that for all histories hst and history events ev we have $stmp(hst :: ev) > stmp(hst)$. As we will see in Section 3.2, this allows us to demonstrate that our enforcement is compatible with a variety of cache models.

Note that we could have also recorded in the history the values of variables read and written. However, the actual values are less important as caching depends on the instructions run and memory locations accessed. Further, granting the $stmp()$ function access to memory secrets would be problematic from a security point of view, as this would allow functions to directly leak the value of secret variables into time, a covert channel requiring a malicious system designer to exploit it.

Timeout Programs in our setting execute with a timeout. The top level rules in Figure 3 distinguish the possibility of producing a sequence of events O within a timeout (defined by $\overset{O}{\Rightarrow}^*$ in rule NO-TO) or timing out after producing O (defined by $\Uparrow O$ in rule TO). The top level rules are thus parameterized in t_0 and $timeout$.

Given an initial configuration for program c with initial memory m_0 and timestamp t_0, this configuration executes and produces outputs as long as the performed steps take no more than what the timeout allows, i.e., $stmp(hst) \leq timeout$. Rule NO-TO captures this: starting from the initial configuration, program c produces list of outputs O and ends up in a new configuration $\langle c', m, hst\rangle$ where $stmp(hst) \leq timeout$. Rule TO captures the situation when the execution times out after producing a list of outputs O. We use wildcard _ when a certain component of the semantic rule is not relevant. One reason for timing out is reaching a time limit. Another reason is getting blocked in the evaluation. Although the latter is impossible in the above semantics, it becomes possible when raising security exceptions in the extended semantics with security monitoring in the next section. In either case, the final configuration is irrelevant and thus omitted.

2.3. Security definition

Projection to ℓ We assume a typing environment Γ mapping variables to security labels ℓ. Labels ℓ are drawn from a lattice of security labels $\mathcal{L} = (\{\texttt{L}, \texttt{H}\}, \sqsubseteq)$ with join ($\sqcup$) and meet ($\sqcap$) operations, where $\texttt{L} \sqsubseteq \texttt{H}$. Label L denotes public, attacker-visible data, while H denotes private user data.

We further define *memory projection to ℓ* to obtain the subset of memory locations whose security label in Γ is ℓ:

$$m|_\ell = \{\{x \mapsto v\} \in m \mid \Gamma(x) = \ell\}$$

Hence, $m = m|_\texttt{L} \uplus m|_\texttt{H}$, where $\uplus$ denotes the disjoint union. We will often refer to $m|_\texttt{L}$ as the *low part* of the memory, and to $m|_\texttt{H}$ as the *high part* of the memory.

We abuse the notation and apply the projection operator to traces of outputs as well. Thus, given trace of outputs O, we define the order-preserving *O projection to ℓ* to obtain the list of outputs in the trace sent on channel ℓ:

$$O|_\ell = \begin{cases} \epsilon & \text{if } O = \epsilon \\ (v, t)_{\ell'} :: O'|_\ell & \text{if } O = (v, t)_{\ell'} :: O' \wedge \ell' \sqsubseteq \ell \\ O'|_\ell & \text{if } O = (v, t)_{\ell'} :: O' \wedge \ell' \not\sqsubseteq \ell \end{cases}$$

where ϵ denotes the empty list.

Attacker knowledge In order to support direct reasoning about what is leaked through the observation of timestamped output, we settle for a *knowledge-based* [6], [24] attacker model. As we previously mentioned, we assume the attacker knows the program c. We also assume the attacker has full knowledge of the $stmp()$ function. In addition, the attacker also knows the low part $m_0^\texttt{L}$ of the initial memory m_0 and observes the trace $O_\texttt{L}$ of low outputs produced by c executing in m_0 and starting at some initial time t_0. Recall that the attacker does not know this time t_0.

Knowledge-based security relates what the attacker knows about secrets before and after observing output. More specifically, the attacker's knowledge about secrets is represented by the set of all initial high memories $m_0^\texttt{H}$ that together with the initial low memory $m_0^\texttt{L}$ could have produced the low output trace $O_\texttt{L}$. Note that the attacker's knowledge is parameterized in $stmp()$ because it operates on the semantics that is parameterized in $stmp()$. Formally:

Definition 1 (Attacker's knowledge)**.**

$$k(c, m_0^\texttt{L}, O_\texttt{L}) = \{m_0^\texttt{H} \mid \exists t_0.\ \langle c, m_0, t_0\rangle \overset{O}{\rightarrow}^* \wedge\ O_\texttt{L} = O|_\texttt{L}\},$$

where $m_0 = m_0^{\mathtt{L}} \uplus m_0^{\mathtt{H}}$.

We write $\langle c, m, hst\rangle \overset{O}{\rightarrow^*}$ to denote that program c starting in memory m and having history hst produces in one or more steps a trace of outputs O.

Note the existential quantification over t_0. It enables us to express that the attacker does not know when the computation started, reflecting the desired setting of remote execution. Consider Program 10 ($p10$) from Figure 5:

```
if h then h1 = h2;
outL(1)
```

For simplicity, here and in some of the later examples we drop the `else` clause (assuming a no-op command in the `else` branch). Depending on the initial value of h, a possible output trace of this program might look like, e.g., $(1, 10)_{\mathtt{L}}$ (when h is 0) or $(1, 20)_{\mathtt{L}}$ (when h is not 0). Yet, due to the existential quantification over the initial timestamp, the attacker's knowledge is the full set in both cases $k(p10, m_0^{\mathtt{L}}, (1, 10)_{\mathtt{L}}) = k(p10, m_0^{\mathtt{L}}, (1, 20)_{\mathtt{L}}) = \{m_0^{\mathtt{H}}\}$, meaning the attacker learns nothing about h.

Although we do not model nondeterministic timing effects, we believe it is possible to lift our framework to nondeterministic $stmp()$ functions. By focusing on secret inputs that may (nondeterministically) lead to a given attacker observation, knowledge-based settings naturally model nondeterministic systems [6], [9], [24].

Another novelty presented by our approach when compared to standard knowledge-based definitions is dealing with timeouts. The rationale for timeout-insensitive security is similar to *progress-insensitive security (PINI)* [5], which is typically enforced by information-flow monitors. Consider Program 12 from Figure 5:

```
while h do h = h;
outL(1)
```

Classical information flow monitors in a setting without timeouts run into a problem: if $\mathtt{out}_{\mathtt{L}}(1)$ is performed then the fact that h was non-zero is leaked. On the other hand, prohibiting loops with high guards would be a drastic restriction. Instead, PINI is often adopted which accepts the program as secure with the idea that is only allowed to leak via "progress" in computation: the attacker should not learn anything beyond what the attacker learns from the fact that $\mathtt{out}_{\mathtt{L}}(1)$ has been reached. Askarov et al. [5] show that the only attacks possible on PINI are brute-force attacks enumerating the space of secret values and diverging upon encountering a match.

By adopting this rationale in a setting with timeouts, we settle for a timeout-insensitive condition. This condition does not prohibit branching on secrets. It permits leaking, but only as much as can be observed from whether the computation timed out. Note that mounting brute-force attacks is harder with timeouts, as the number of guesses is restricted by the time allocated for a run.

Timeout knowledge Hence, we define *timeout knowledge* as how much the attacker can learn from the fact that a program times out.

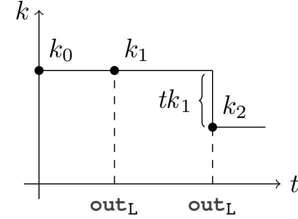

Figure 4: The x axis is time, and the y axis is the attacker's knowledge. The plot tracks the attacker's knowledge for a secure program. k_0 is the attacker's knowledge before observing any outputs; k_1 is the attacker's knowledge after observing the first low output; tk_1 is the attacker's timeout knowledge after observing the first low output; and k_2 is the attacker's knowledge after observing the second low output.

For program c and initial memory m_0 that produces low output list O_{L}, the attacker's timeout knowledge is represented by the set of all initial high memories $m_0^{\mathtt{H}}$ that together with the initial low memory $m_0^{\mathtt{L}}$ could have produced O_{L} and then timed out. Note that the timeout knowledge is parameterized in both $timeout$ and $stmp()$. Formally:

Definition 2 (Timeout knowledge)**.**

$$tk(c, m_0^{\mathtt{L}}, O_{\mathrm{L}}) = \{m_0^{\mathtt{H}} \mid \exists t_0.\ \langle c, m_0, t_0\rangle \Uparrow O \wedge\ O_{\mathrm{L}} = O|_{\mathrm{L}}\},$$

where $m_0 = m_0^{\mathtt{L}} \uplus m_0^{\mathtt{H}}$.

For example, Program 12 ($p12$) times out when $m_0(h) \neq 0$. Provided $timeout$ is large enough, the execution terminates when $m_0(h) = 0$. We thus have $tk(p12, _, \epsilon) = \{m_0^{\mathtt{H}} \mid m_0^{\mathtt{H}}(h) \neq 0\}$.

Security definition Observe that the smaller the attacker's knowledge set, the more the attacker knows about the secret data. To express timeout-insensitivity we demand that for every new attacker-visible output the knowledge set should not decrease by more than the previous timeout knowledge. Figure 4 illustrates this.

Hence, a program is (remotely) secure if the knowledge of the attacker observing a new event o after having observed low output trace O_{L} is no more precise than the knowledge the attacker previously gained from observing low output trace O_{L} minus the timeout knowledge for the same low output trace. Note that remote security is also parameterized in both $stmp()$ and $timeout$. Formally:

Definition 3 (Remote Security)**.** Given $timeout$ and $stmp()$, program c complies with remote security (RS) if for all low memories m_{L} and low output traces $O_{\mathrm{L}} :: o$ such that $\langle c, m_0, t_0\rangle \overset{O::o}{\Rightarrow^*}$ for some t_0 and $O|_{\mathrm{L}} = O_{\mathrm{L}} :: o$ we have $k(c, m_{\mathrm{L}}, O_{\mathrm{L}} :: o) \supseteq k(c, m_{\mathrm{L}}, O_{\mathrm{L}}) \setminus tk(c, m_{\mathrm{L}}, O_{\mathrm{L}})$.

Examples Our definitions and statements are parameterized over arbitrary $timeout$ and strictly increasing $stmp()$. For the purposes of the examples throughout the paper, assume a simple model with $timeout = 1000$ and $stmp()$ to be the number of events in the history. Assume h, h_1, and

h_2 are secret variables, and the rest are public. Let us further illustrate RS on examples (see Figure 5) and compare it to standard local attacker-based definitions in the style of *timing-sensitive noninterference (TimSNI)* [1], [4], [51]. Upon varying the secret part of an initial memory, these definitions either require a constant number of instructions in the program runs [1], [51] or place syntactic restrictions on taking the same control flow path [4], in both cases significantly restricting possibilities for branching on secrets.

Both RS and TimSNI agree on the baseline of rejecting explicit and implicit flows, which are dangerous even without considering time. Program 1 displays an explicit flow passing a secret directly to a public output. The attacker's knowledge is refined from the full set to a singleton for h, hence RS rejects the program. Program 2 leaks information about h implicitly via the control flow of the branching. The attacker's knowledge is refined from the full set to either the set of memories with non-zero integers for h (in case the public output is 1) or to the set of memories where h is 0 (in case the public output is 0). Therefore, RS rejects the program.

RS and TimSNI also agree on remotely-exploitable timing attacks. Program 3 records the time before and after a computation whose duration depends on a secret. The former is stored in variable x, while the latter is retrieved from the timestamp at the moment of the output. TimSNI rejects the program because of the conditional that breaks constant-time. RS rejects the program because the publicly-observable time difference allows the attacker to infer whether h was 0. Program 4 is similarly insecure, as the two clock reads before and after branching on secret are available via the time of the public output.

Program 5 demonstrates a leak when there is no branching on a secret between time reads. Although the branching takes place before the first output, the effect of the branching is reflected in the assignment between the outputs. If h were non-zero, the time difference between outputting 1 and 2 would likely be smaller due to cache.

Program 6 exploits data/instruction cache to set up a timing leak. The assignment to h_2 computing the factorial for 30 will execute faster in case h is non-zero due to data/instruction cache effects. Our definition captures this through function $stmp()$ that depends on the command history. As the time difference is recorded and sent to the attacker, RS deems the program insecure. TimSNI is in agreement, rejecting the program because of the conditional.

Program 7 illustrates a leak by a carefully crafted delay. In contrast to Program 3, the time read is not passed to the public output directly. Instead, the current clock value stored in variable x is used, and based on the parity of secret h, the program produces the final output either after pausing until an "even time segment" (between seconds 0 and 1, or 2 and 3, ...) for even values of h, or after pausing until an "odd time segment" (between second 1 and 2, or 3 and 4, ...) for odd values of h. Program 8 achieves a similar effect, but without using time reads in high context.

The beauty of RS is that it captures these subtle leaks by design. Recall the attacker has full knowledge of the $stmp()$

	Program	TimSNI TypeS	RS Clockwork
(1)	/* explicit */ **out**$_L$(h)	×	×
(2)	/* implicit */ **if** h **then** $l = 1$ **else** $l = 0$; **out**$_L$(l)	×	×
(3)	/* time, branch, I/O */ x getsTime; **if** h **then** $h_1 = h_2$; **out**$_L$(x)	×	×
(4)	/* I/O, branch, I/O */ **out**$_L$(1); **if** h **then** $h_1 = h_2$; **out**$_L$(2)	×	×
(5)	/* cache */ **if** h **then** $h_1 = h_2$; **out**$_L$(1); $h_1 = h_2$; **out**$_L$(2)	×	×
(6)	/* cache/JIT */ **if** h **then** $h_1 = fact(30)$; t_0 getsTime; $h_2 = fact(30)$; t_1 getsTime; **out**$_L$($t_1 - t_0$)	×	×
(7)	/* high delay */ x getsTime; **if** h % 2 = $seconds(x)$ % 2 **then** $h = h$ **else** $h = h; \ldots; h = h$; **out**$_L$(1)	×	×
(8)	/* low delay */ x getsTime; **if** $seconds(x)$ % 2 **then** $x = x$ **else** $x = x; \ldots; x = x$; **if** h % 2 **then** $h = h$ **else** $h = h; \ldots; h = h$; **out**$_L$(1)	×	×
(9)	/* no I/O */ **if** h **then** $h_1 = h_2$	×	✓
(10)	/* I/O last */ **if** h **then** $h_1 = h_2$; **out**$_L$(1)	×	✓
(11)	/* I/O first */ **out**$_L$(1); **out**$_L$(2); **if** h **then** $h_1 = h_2$	×	✓
(12)	/* timeout */ **while** h **do** $h = h$; **out**$_L$(1)	×	✓

Figure 5: Security definitions: TimSNI vs. RS. Enforcements: TypeS vs. Clockwork.

function. Thus, observing an output in an even time segment indicates that the secret was even, and vice versa. TimSNI happens to reject both programs for a more conservative reason, as there is branching on secret that breaks constant-time even before the public output is reached.

We will now demonstrate the difference between the two definitions by discussing the programs that are intuitively se-

cure and rightfully accepted by RS, yet rejected by TimSNI.

Program 9 executes an assignment depending on the value of secret variable h. TimSNI deems this program insecure because it always considers execution time to be observable by the attacker, and the program takes different time depending on the secret. In contrast, as the program does not exhibit any public outputs, RS deems it secure.

Program 10 illustrates the difference between a remote attacker that cannot observe when the program started executing and a local attacker that can. As in the previous example, this program is deemed insecure by TimSNI. In contrast, it is compliant with RS because even if the observation of 1 on channel L has a timestamp, the attacker cannot infer anything about the secret values because the attacker does not know when the program started executing. Indeed, the attacker initial and final knowledge sets are equal. Similarly, Program 11 is compliant with RS, but does not satisfy TimSNI.

Finally, Program 12 ($p12$) illustrates insensitivity to leaks via timeouts. Recall that the program times out on initial memories m_0 with $m_0(h) \neq 0$, giving $tk(p12, _, \epsilon) = \{m_0^{\mathtt{H}} \mid m_0^{\mathtt{H}}(h) \neq 0\}$. As for any initial low memory $m_{\mathtt{L}}$, $\{m_0^{\mathtt{H}} \mid m_0^{\mathtt{H}}(h) = 0\} = k(p12, m_{\mathtt{L}}, \epsilon :: (1,t)_L) \supseteq k(p12, m_{\mathtt{L}}, \epsilon) \setminus tk(c, m_{\mathtt{L}}, O|_{\mathtt{L}}) = \{m_0^{\mathtt{H}} \mid m_0^{\mathtt{H}}(h) = 0\}$, the program is accepted by RS. It is rejected by TimSNI for similar reasons as above. This indicates that RS is more permissive than TimSNI as it considers a weaker attacker model, in line with attackers models on IoT platforms such as IFTTT.

3. Enforcement

This section presents Clockwork, our security monitor. Recall that "adhering to constant-time programming is hard" [4]. We target pushing the boundaries of what can be done without resorting to constant-time programming. At the same time we target avoiding other conservative measures, such as labeling all clock readings as sensitive [56], wrapping all conditionals that branch on secrets into atomic statements [60], disallowing public outputs after branching on secrets [57], or disallowing looping on secrets [1], [51], [60].

3.1. Security monitor

The examples in Figure 5 identify patterns of secure and insecure programs, which we use for the design of the monitor. Note that all these examples, secure and insecure, are rejected by traditional constant-time type-based enforcements (TypeS) in the style of Volpano and Smith [60] and Agat [1], [51]. Our goal is to improve permissiveness.

Clockwork consists of two components: untimed and timed. The untimed component leverages standard dynamic information flow tracking [11], [16], [38], [41] extended with bookkeeping of histories. This component is sufficient to reject explicit and implicit flows as in Programs 1 and 2. The timed component is unique to our setting. We focus on this component in the presentation of the monitor.

The timed component enforces the following discipline. It allows a single low output after a high-guarded control flow statement only if no clock readings were performed before and it disallows any low output after a high-guarded control flow statement if at least one low output was performed before, irrespective of when the clock readings were made.

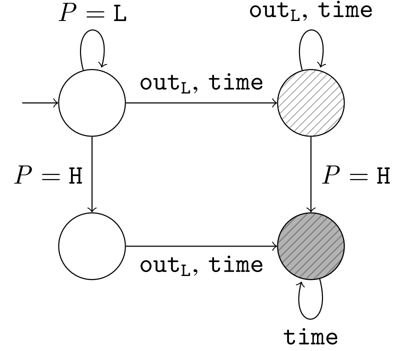

Figure 6: Security monitor as state automaton. Transitions consists of (i) $P = \mathtt{H}$ denoting context upgrade from L to H for the first time, (ii) $\mathtt{out}_{\mathtt{L}}$ denoting an output on the low channel $\mathtt{out}_{\mathtt{L}}(e)$, for some e, and (iii) `time` denoting a time read x `getsTime`, for some x. All states are terminal. The hatched states reflect automaton states where time reads are allowed. The state in the dark node does not allow any further low outputs.

Security state We extend the configuration from the previous section with a security state $st = (S, P, \Gamma, T, Q)$. We use a program counter (pc) mechanism to record the security level of the guard in the current control-flow context. The security state tuple contains a stack of program counters S (initially empty), a security label P denoting the highest program counter ever pushed onto the stack (initially L), a typing environment Γ, a boolean T denoting whether any time-reading clock invocations have been made (initially `false`), and a boolean Q denoting whether any low outputs have been produced (initially `false`). Γ is defined as previously, a mapping from program variables x to security levels ℓ drawn from a lattice of security levels $\mathcal{L} = (\{\mathtt{L}, \mathtt{H}\}, \sqsubseteq)$. Of these, S and Γ are standard [11], [16], [38], [41] while P, T, and Q are novel to our mechanism.

Figure 6 illustrates the use of security states by depicting the timed component of our security monitor as a state automaton. In case the program enters high context before having read time or produced low output (moving down from the start node of the automaton), we allow a single low output or time read. In the other case (moving right from the start node) we allow low outputs and time reads until we enter high context. After that time reads are still allowed but not low outputs. All states are terminal. The hatched states reflect states where time reads are allowed. The state in the dark node does not allow any further low outputs.

Security monitor semantics The semantics of our security monitor Clockwork is defined by judgment $\langle c, m, hst, st\rangle \rightarrow \langle c', m', hst', st'\rangle$ which reads as program c in memory m,

with program execution history hst and security state st after a single step of computation reaches configuration $\langle c', m', hst', st' \rangle$. Figure 7 illustrates the semantic rules of Clockwork. When evaluating an expression, the security monitor returns both a value and a security label. The gray highlighting spotlights the timed features of the monitor.

The state st involves several additional security checks that need to be satisfied before the monitor allows for a new step in the computation. In the following, we discuss some of the most important rules of Clockwork.

Rule SEC-ASSIGN checks for explicit flows. In combination with rules SEC-IF, SEC-WHILE, and SEC-END, it also tracks implicit flows. Rules SEC-IF and SEC-END keep track of the stack S of program counters, pushing and popping the current pc respectively (similarly in SEC-WHILE and SEC-END). Function $lev(S)$ on stack $S = \ell_1 :: \ldots :: \ell_k$ is defined to return the join $\sqcup_{i=1}^{k} \ell_i$ of the levels on the stack. $lev(S)$ is thus $\mathtt{H}$ if and only if $\mathtt{H} \in S$.

Rules SEC-IF and SEC-WHILE record the first branching on $\mathtt{H}$ by updating P. These rules also implement standard *no-sensitive upgrade (NSU)* checks [8], [61] which do not allow relabeling variables whose security level is below the context level. Thus, assignment of expression e to variable x is allowed only if the security level of x is not below the security level of the security context ($lev(S) \sqsubseteq \Gamma(x)$). This rightfully rejects the implicit flow in Program 2.

Rule SEC-TIME updates the security state by setting T to `true`, recording that a time read has been made. Otherwise, it is similar to rule SEC-ASSIGN.

Rules SEC-OUTPUT-$*$ illustrate the cases when a (low) output is allowed. A first requirement is that outputting e on channel ℓ is permitted if the security level ℓ_e of e is not above the label ℓ of the channel ($\ell_e \sqsubseteq \ell$). This rightfully rejects the explicit flow in Program 1.

Rule SEC-OUTPUT-1 captures the case when the highest pc ever pushed onto the stack is $\mathtt{L}$, i.e., $P = \mathtt{L}$, irrespective of whether any low outputs have been previously produced. The two upper automaton states in Figure 6 are captured by this rule. This allows us to rightfully accept Program 11. The case when the highest pc on the stack is $\mathtt{H}$ is considered by rule SEC-OUTPUT-2, matching the two lower states of the automaton in Figure 6. The rule allows for only a single low output, under the condition that no prior time reads were performed. This rightfully rejects Programs 3 to 8.

The delay leaks in Programs 7 and 8 show that an enforcement attempting to be liberal with time reads by, e.g., not restricting the time reads but instead tainting time as soon as the computation entered the first high context would be unsound. Indeed, allowing output on the low channel as in these programs is insecure. The insecurity is captured by our monitor because the attempted output is preceded by a time read and high branching.

At the same time, these restrictions do not prevent us from rightfully accepting Programs 9, 10, and 12.

Alternative enforcement Another enforcement pattern is to restrict the data affected by the time reads, and not the time reads themselves. The enforcement can be achieved by introducing time taints into the security labels and treating the information that depends on time reads as time-tainted. Then `time` in Figure 6 can be interpreted as events that branch on time-dependent data rather than time reads. Under this discipline $\mathtt{out}_{\mathtt{L}}(e)$ would be allowed under $P = \mathtt{H}$ as long as e is not time-tainted. This means that programs like

```
x getsTime;
c(h,l); // does not use x
y getsTime;
... // use x and y to compute time statistics
outL(1)
```

where $\mathtt{c}(h, l)$ is explicit and implicit flow-free can be accepted. However, while this alternative gains permissiveness in one way, it loses permissiveness in another, due to the NSU restrictions when tracking time-taintedness. This means the execution of secure programs such as

```
x getsTime;
if seconds(x) % 2 then y = 1
```

is blocked when taking the **then** branch because redefining untainted variables in a tainted context is illegal. While the program is problematic for this alternative enforcement, Clockwork rightfully accepts it.

3.2. Soundness

We begin to present the formal guarantees of our system by introducing a semantics preservation result: any program accepted by the security monitor preserves the original semantics of that program. Formal definitions of relations stated here only informally are reported in the appendix, while proofs of the statements below are presented in the full version of the paper [15].

Lemma 1 (Semantics preservation)**.** *Given $stmp()$, for any program c, memory m, and history hst, if $\langle c, m, hst, _ \rangle \overset{O}{\rightarrow}^{*} \langle c', m', hst', _ \rangle$ then $\langle c, m, hst \rangle \overset{O}{\rightarrow}^{*} \langle c', m', hst' \rangle$.*

Recall that the semantics of the monitor is parametric in function $stmp()$. We assume our selection of function $stmp()$ ignores the computational timing costs involved by the additional security checks performed by Clockwork. Recall also that $stmp()$ is a function on histories. Thus, for two equal history sequences $stmp()$ returns the same timestamp, irrespective of the memories that led to those histories.

By observing the security automaton in Figure 6, we can see that a flexible $stmp()$ function leads to an enforcement compatible with a variety of cache models.

Consider the trace of a program run which has not entered a high context. This corresponds to the two upper states of the automaton where P is $\mathtt{L}$. In these states the monitor forces the control path to be independent of the secrets. This means that any other trace originating in a low-equal initial memory must run in *strong lockstep* with the original trace. Strong lockstep is a strong notion: the security monitor configurations must be identical, with the only component that may differ being the high parts of the memories. Because the $stmp()$ function only depends on the history and not on the memory, strong lockstep implies

$$\langle v,m,\Gamma\rangle \Downarrow v:\mathtt{L} \qquad \frac{\Gamma(x)=\ell \qquad m(x)=v}{\langle x,m,\Gamma\rangle \Downarrow v:\ell} \qquad \frac{\langle e_i,m,\Gamma\rangle \Downarrow v_i:\ell_i\ (i=1\dots n) \qquad v=f(v_1,\dots,v_n) \qquad \ell=\bigsqcup_{i=1}^{n}\ell_i}{\langle f(e_1,\dots,e_n),m,\Gamma\rangle \Downarrow v:\ell}$$

SEC-ASSIGN

$$\frac{\langle e,m,\Gamma\rangle \Downarrow v:\ell \qquad st=(S,P,\Gamma,T,Q) \qquad lev(S)\sqsubseteq\Gamma(x) \qquad st'=(S,P,\Gamma[x\mapsto\ell\sqcup lev(S)],T,Q)}{\langle x=e,m,hst,st\rangle \rightarrow \langle \mathtt{stop},m[x\mapsto v],hst::\mathtt{asn}(x,e),st'\rangle}$$

SEC-SEQ-1

$$\frac{\langle c_1,m,hst,st\rangle \xrightarrow{o} \langle c_1',m',hst',st'\rangle}{\langle c_1;c_2,m,hst,st\rangle \xrightarrow{o} \langle c_1';c_2,m',hst',st'\rangle}$$

SEC-SEQ-2

$$\frac{}{\langle \mathtt{stop};c,m,hst,st\rangle \rightarrow \langle c,m,hst,st\rangle}$$

SEC-END

$$\frac{st=(S::pc,P,\Gamma,T,Q) \qquad st'=(S,P,\Gamma,T,Q)}{\langle \mathtt{end},m,hst,st\rangle \rightarrow \langle \mathtt{stop},m,hst::\mathtt{join},st'\rangle}$$

SEC-IF

$$\frac{\langle e,m,\Gamma\rangle \Downarrow v:\ell \qquad v\neq 0\Rightarrow i=1 \qquad v=0\Rightarrow i=2 \qquad st=(S,P,\Gamma,T,Q) \qquad st'=(S::\ell,\ P\sqcup\ell\ ,\Gamma,T,Q)}{\langle \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2,m,hst,st\rangle \rightarrow \langle c_i;\mathtt{end},m,hst::\mathtt{br}(e),st'\rangle}$$

SEC-WHILE

$$\frac{\begin{array}{c}\langle e,m,\Gamma\rangle \Downarrow v:\ell\\ v\neq 0\Rightarrow c'=c;\mathtt{end};\mathtt{while}\ e\ \mathtt{do}\ c \qquad v=0\Rightarrow c'=\mathtt{end} \qquad st=(S,P,\Gamma,T,Q) \qquad st'=(S::\ell,\ P\sqcup\ell\ ,\Gamma,T,Q)\end{array}}{\langle \mathtt{while}\ e\ \mathtt{do}\ c,m,hst,st\rangle \rightarrow \langle c',m',hst::\mathtt{br}(e),st'\rangle}$$

SEC-TIME

$$\frac{t=stmp(hst) \qquad st=(S,P,\Gamma,T,Q) \qquad lev(S)\sqsubseteq\Gamma(x) \qquad st'=(S,P,\Gamma[x\mapsto P],\mathtt{true},Q)}{\langle x\ \mathtt{getsTime},m,hst,st\rangle \rightarrow \langle \mathtt{stop},m[x\mapsto t],hst::\mathtt{asn}(x,t),st'\rangle}$$

SEC-OUTPUT-1

$$\frac{\begin{array}{c}\langle e,m,\Gamma\rangle \Downarrow v:\ell_e\\ st=(S,\mathtt{L},\Gamma,T,Q) \qquad t=stmp(hst::\mathtt{o}(e,\ell)) \qquad \ell_e\sqsubseteq\ell \qquad \ell\neq\mathtt{L}\Rightarrow st'=st \qquad \ell=\mathtt{L}\Rightarrow st'=(S,\mathtt{L},\Gamma,T,\ \mathtt{true}\ )\end{array}}{\langle \mathtt{out}_\ell(e),m,hst,st\rangle \xrightarrow{(v,t)_\ell} \langle \mathtt{stop},m,hst::\mathtt{o}(e,\ell),st'\rangle}$$

SEC-OUTPUT-2

$$\frac{\begin{array}{c}\langle e,m,\Gamma\rangle \Downarrow v:\ell_e \qquad st=(S,\mathtt{H},\Gamma,T,Q) \qquad t=stmp(hst::\mathtt{o}(e,\ell))\\ lev(S)\sqcup\ell_e\sqsubseteq\ell \qquad \ell\neq\mathtt{L}\Rightarrow st'=st \qquad \ell=\mathtt{L}\Rightarrow\ (T\vee Q=\mathtt{false})\wedge st'=(S,\mathtt{H},\Gamma,T,\mathtt{true})\end{array}}{\langle \mathtt{out}_\ell(e),m,hst,st\rangle \xrightarrow{(v,t)_\ell} \langle \mathtt{stop},m,hst::\mathtt{o}(e,\ell),st'\rangle}$$

Figure 7: Semantics of security monitor Clockwork. The timed features are highlighted in gray.

that the timestamps computed for the respective events by $stmp()$ will also be identical in both traces. The timestamps of the respective events will always be the same because the sequences of executed instructions are identical.

Next, consider the trace of a program run which has entered high context. This corresponds to the two lower states of the automaton where P is $\mathtt{H}$. The $stmp()$ function is not important in these states. In the lower left state, we allow at most one low output or time read, but the attacker will not be able to learn anything from the output's timestamp because the attacker does not know when the computation started and there are no other low outputs or time reads to relate to. In the lower right state, no further low output is allowed, which obviously implies that the attacker will not be able to learn anything further.

This brings us to the main result of this section: Clockwork enforces remote security.

Theorem 1 (Soundness)**.** *Given timeout and $stmp()$, for any program c, initial memory m_0, and timestamp t_0, if $\langle c,m_0,t_0,st_0\rangle \overset{O}{\Rightarrow}{}^* \langle c',m,hst,st\rangle$ and $O|_{\mathtt{L}}=O_{\mathtt{L}}::o$, then $k(c,m_0^{\mathtt{L}},O_{\mathtt{L}}::o)\supseteq k(c,m_0^{\mathtt{L}},O_{\mathtt{L}})\setminus tk(c,m_0^{\mathtt{L}},O_{\mathtt{L}})$.*

Proof. By contradiction. Assuming the inverse of $k(c,m_0^{\mathtt{L}},O_{\mathtt{L}}::o)\supseteq k(c,m_0^{\mathtt{L}},O_{\mathtt{L}})\setminus tk(c,m_0^{\mathtt{L}},O_{\mathtt{L}})$, there exists $m_2=m_0^{\mathtt{L}}\uplus m_2^{\mathtt{H}}$ such that $m_2^{\mathtt{H}}\in k(c,m_0^{\mathtt{L}},O_{\mathtt{L}})\setminus tk(c,m_0^{\mathtt{L}},O_{\mathtt{L}})$,

but $m_2^{\mathtt{H}} \notin k(c, m_0^{\mathtt{L}}, O_{\mathtt{L}} :: o)$. At the same time, because $\langle c, m_0, t_0, st_0\rangle \overset{O}{\Rightarrow}^* \langle c', m, hst, st\rangle$, there exists $m_1 = m_0^{\mathtt{L}} \uplus m_1^{\mathtt{H}}$ so that $m_1^{\mathtt{H}} \in k(c, m_0^{\mathtt{L}}, O_{\mathtt{L}} :: o)$ and $m_1 \notin tk(c, m_0^{\mathtt{L}}, O_{\mathtt{L}})$, implying $m_1^{\mathtt{H}} \in k(c, m_0^{\mathtt{L}}, O_{\mathtt{L}}) \setminus tk(c, m_0^{\mathtt{L}}, O_{\mathtt{L}})$.

To establish contradiction, we prove the sequence of low outputs $O_{\mathtt{L}} :: o$ of the monitored execution in m_1 is mirrored by equivalent configurations in the monitored execution that originates from m_2:

$$
\begin{array}{ccccccccccccc}
cfg_1 & \overset{O_{\mathtt{L}}}{\rightarrow}^* & cfg_1' & \overset{\epsilon}{\nearrow} & cfg_1'' & \rightarrow^* & cfg_1^{iv} & \searrow & cfg_1^{v} & \rightarrow^* & cfg_1^{n} & \overset{o}{\rightarrow} & cfg_1''' \\
 & & \sim_{(1)} & & & & & & \sim_{(2)} & & \sim_{(3)} & & \\
cfg_2 & \overset{O_{\mathtt{L}}}{\rightarrow}^* & cfg_2' & \overset{\epsilon}{\nearrow} & cfg_2'' & \rightarrow^* & cfg_2^{iv} & \searrow & cfg_2^{v} & \rightarrow^* & cfg_2^{n} & \overset{o'}{\rightarrow} & cfg_2'''
\end{array}
$$

The first row indicates the execution originating from m_1, while the second row the execution originating from m_2. $\nearrow$ indicates a (first) change in pc from L to H, $\searrow$ indicates a change in pc from H to L. ϵ indicates no output is produced in transition $\nearrow$.

We make use of confinement and lockstep reasoning to prove equivalences 1-3, depending on when the first branching on a secret (which flips P from L to H) is encountered. Strong equivalence is preserved when $P = \mathtt{L}$ giving (1), confinement gives equivalence (2), while weak equivalence is preserved when $P = \mathtt{H}$ giving (3). The latter is sufficient for mirroring the runs because no low outputs are allowed under high P. The full proof is reported in the appendix, together with the formal definitions of the equivalences and the auxiliary lemmas. □

4. Generalization to arbitrary lattices

In order to generalize the monitor to arbitrary lattices of security levels, we keep track of not only the previous low outputs, but the previous outputs at all levels. Thus, we overload boolean Q from the security state in Section 3 to represent a function from security levels to booleans: if Q maps security level ℓ to `true` (resp. `false`) then an output at level ℓ has occurred (resp. has not occurred). The goal of the monitor is to only allow flows from lower to higher security levels. With the exception of the output rules, the generalized monitor rules remain as in Figure 7, but Q is considered now to be a function, as described above, and not a boolean. The output rules change as follows:

GEN-SEC-OUTPUT-1

$$
\frac{\begin{array}{c} st = (S, P, \Gamma, T, Q) \qquad P \sqsubseteq \ell \qquad \langle e, m, st\rangle \Downarrow v : \ell_e \\ t = stmp(hst :: \mathsf{o}(e, \ell)) \qquad \ell_e \sqsubseteq \ell \\ st' = (S, P, \Gamma, T, Q[\ell \mapsto \mathtt{true}]) \end{array}}{\langle \mathtt{out}_\ell(e), m, hst, st\rangle \overset{(v,t)_\ell}{\rightarrow} \langle \mathtt{stop}, m, hst :: \mathsf{o}(e, \ell), st'\rangle}
$$

GEN-SEC-OUTPUT-2

$$
\frac{\begin{array}{c} st = (S, P, \Gamma, T, Q) \qquad P \not\sqsubseteq \ell \qquad \langle e, m, st\rangle \Downarrow v : \ell_e \\ t = stmp(hst :: \mathsf{o}(e, \ell)) \qquad lev(S) \sqcup \ell_e \sqsubseteq \ell \\ \forall \ell'.\ \ell' \sqsubseteq \ell . Q(\ell') = \mathtt{false} \qquad T = \mathtt{false} \\ st' = (S, P, \Gamma, T, Q[\ell \mapsto \mathtt{true}]) \end{array}}{\langle \mathtt{out}_\ell(e), m, hst, st\rangle \overset{(v,t)_\ell}{\rightarrow} \langle \mathtt{stop}, m, hst :: \mathsf{o}(e, \ell), st'\rangle}
$$

Rule GEN-SEC-OUTPUT-1 applies if $P \sqsubseteq \ell$, that is if the output channel level (ℓ) is equal to or higher than the highest pc encountered so far (P). If expression e is allowed on channel ℓ ($\ell_e \sqsubseteq \ell$), the security state is updated to record that there was an output at level ℓ ($Q[\ell \mapsto \mathtt{true}]$).

Rule GEN-SEC-OUTPUT-2 applies if $P \not\sqsubseteq \ell$. Intuitively, this means the output may leak information about previous higher branches. In order to avoid leaks due to clock readings (as in Program 3), we require $T = \mathtt{false}$. In order to prevent the attacker from learning information about the time of the current output by inspecting the timestamps of the previous outputs, we require no previous outputs at level lower than or equal to ℓ to have occurred (constraint $\forall \ell'.\ \ell' \sqsubseteq \ell . Q(\ell') = \mathtt{false}$). A leaking example in this case, when considering a lattice with $\mathtt{L} \sqsubseteq \mathtt{M} \sqsubseteq \mathtt{H}$, is a program similar to Program 4 where the second output is sent on channel M (instead of L).

In order to claim the security guarantees of the generalized monitor, we generalize the knowledge of the attacker observing at level ℓ or below to an arbitrary lattice. Similarly, the generalized attacker's knowledge is parameterized in $stmp()$.

Definition 4 (Generalized attacker's knowledge)**.**

$$k_\ell(c, m_0^\ell, O_\ell) = \{m_0^{\mathtt{H}} \mid \exists t_0.\ \langle c, m_0, t_0\rangle \overset{O}{\rightarrow}^* \wedge O_\ell = O|_\ell\},$$

where m_0^ℓ maps all variables with level $\sqsubseteq \ell$ to their values, $m_0^{\mathtt{H}}$ maps all variables with level $\not\sqsubseteq \ell$ to their values, and $m_0 = m_0^\ell \uplus m_0^{\mathtt{H}}$.

We generalize the timeout knowledge in a similar way, also parameterized in $stmp()$.

Definition 5 (Generalized timeout knowledge)**.**

$$tk(c, m_0^\ell, O_\ell) = \{m_0^{\mathtt{H}} \mid \exists t_0.\ \langle c, m_0, t_0\rangle \Uparrow O \wedge\ O_\ell = O|_\ell\},$$

where m_0^ℓ and m_0 are defined as in Definition 4.

The soundness theorem for the generalized monitor is stronger than the one in Section 3 since we can enforce security for attackers that can observe at any security level (not only L).

Theorem 2 (Soundness for generalized monitor)**.** *Given timeout and stmp(), for any level ℓ, program c, initial memory m_0, and timestamp t_0, if $\langle c, m_0, t_0, st_0\rangle \overset{O}{\Rightarrow}^* \langle c', m, hst, st\rangle$ and $O|_\ell = O_\ell :: o$, then $k_\ell(c, m_0^\ell, O_\ell :: o) \supseteq k_\ell(c, m_0^\ell, O_\ell) \setminus tk_\ell(c, m_0^\ell, O_\ell)$.*

The proof of the theorem is presented in the full version of the paper [15].

5. Implementation

We implement the security monitor in Figure 7 as an extension to JSFlow [40], a state-of-the-art information flow tracker for JavaScript. We then evaluate it on a set of both secure and insecure programs to assess its soundness and demonstrate its permissiveness. The implementation of the monitor and the benchmarks are available publicly [15].

Extension to JSFlow The implementation of our monitor closely follows the semantics of Clockwork. We extend the context of the JSFlow monitor with two boolean variables, tracking whether any time-reading clock invocations have been made (initially `false`), and whether any low outputs have been produced (initially `false`), as well as a security label tracking the highest program counter ever pushed onto the stack (initially `L`). The other components of the security state are already defined by JSFlow.

By default, JSFlow treats any clock readings via the `Date` construct as high. Hence, we modify the monitor such that the label assigned will be instead the highest label of the program context *pc*. Also, whenever a `Date` constructor is invoked, we record it by setting the corresponding boolean variable to `true`. Similarly, whenever a conditional or loop statement branches on a high guard, or a low output is produced for the first time, we update the corresponding variables accordingly.

Evaluation We evaluate our monitor on a set of secure and insecure benchmark programs. The benchmark suite includes the 12 programs from Figure 5. In addition, we utilize the publicly available benchmarks by Bastys et al. [16] to extract programs that model popular third party IFTTT apps (13 programs with and 13 programs without timing leaks) as well as code gathered from online forums (7 programs with and 7 programs without timing leaks). The experiments demonstrate that our monitor rejects all insecure programs, while accepting all secure programs.

6. Case studies: IFTTT and VJSC

In this section we demonstrate the feasibility of our monitor on a case study with IFTTT, a popular IoT app platform, and a case study with VJSC, a state-of-the-art cryptographic library for implementing e-voting clients.

At the core of IFTTT are so-called *applets*, reactive apps that include *triggers*, *actions*, and *filter* code. When the event specified by the trigger (e.g., "If I'm approaching my home") is fired, the event specified by the action (e.g., "Switch on the smart home lights") is executed. App makers can use filter code to customize the action event (e.g., "to red"). If present, the filter code is invoked after the trigger has been fired and before the action is dispatched. Previous related work gives an overview of trigger-action IoT platforms [10], [35].

We will further focus on the filter code, as it is not visible to the user installing an applet. Therefore, a malicious app maker can write filter code that exfiltrates the user private information upon installation and execution of an applet [16]. Filter code consists of JavaScript code snippets with APIs pertaining to the services the applet uses. The code is run in a sandbox and it cannot block or perform any I/O operations other than by using APIs to configure the output actions of the applet. Moreover, the filter code is executed in batch mode and it is forced to terminate upon a timeout. If the timeout is not exceeded, the output actions take place after the filter code has terminated.

IFTTT applets are an excellent use case for illustrating our remote attacker model and for validating the soundness and permissiveness of our monitor: the filter code manipulates sensitive information from trigger APIs (e.g., user location, voice-controlled assistants, email or calendar events). Further, the filter code may perform secret-dependent branching and clock readings via `Date` APIs and IFTTT-specific timing APIs, such as `Meta.currentUserTime` and `Meta.triggerTime`, and may only lead to at most one output action. In this model, a malicious app maker has no direct knowledge of the execution time of a trigger, unless it performs clock readings via timing APIs. We remark that IFTTT-specific timing APIs such as `Meta.currentUserTime` and `Meta.triggerTime` always yield the same clock readings within a single run and therefore are only exploitable in combination with `Date` APIs.

The Open Verificatum project [31] provides implementations of cryptographic primitives and protocols that can be used to implement a wide range of electronic voting systems. The software has been used in real elections to tally more than 3,000,000 votes, including elections in Norway, Spain, and Estonia. We focus on the client-side Verificatum JavaScript Crypto (VJSC) library, which provides encryption primitives needed in e-voting. Specifically, VJSC allows generating public and private key pairs based on (variations of) the El Gamal cryptosystem, and uses them to encrypt votes and send them to a central server leveraging a mix-net infrastructure. In this case, we assume a network attacker that can observe the presence of an encrypted vote on the network whenever it is sent by the client.

6.1. Remote timing attack on IFTTT

We show that even if explicit and implicit flows are ruled out (e.g., assume a monitor is in place to block these flows in IFTTT [16]), it is still possible for malicious apps to exfiltrate the user private information. We have implemented and tested the following malicious pattern:

```
x = secretAPI();
xBin = convertToBinary(x);
hacked = "";
for (i=0 to maxConstant){
   startTime = getTime();
   if (xBin(i) == 0){long_computation();}
   else {short_computation();}
   endTime = getTime();
   if (endTime - startTime > 0) {hacked += 0;}
   else {hacked += 1;}};
out_L(binToAscii(hacked));
```

The program magnifies a pattern similar to Programs 3 and 4 from Figure 5 to exfiltrate the sensitive data in `secretAPI()`. Specifically, the malicious code above first converts `x` to a binary string, then leaks each bit by performing a `long_computation()` whenever the bit is 0, and a `short_computation()` otherwise. Observe that both computations manipulate only secret data, thus evading any checks for implicit flows. By measuring the execution time of each branch, we can reliably learn a secret bit. In fact, the time difference is 0 only when `short_computation()` is executed. The leak can be easily magnified using a loop scanning each bit up to a predefined public constant. Finally, the bitstring is converted to an ASCII string and sent over a public channel.

Our experiments reliably exfiltrate strings of 350 bits before reaching the timeout. As a result, our timing attack can leak any ASCII string up to 50 characters. We verified the feasibility of the attacks by creating private IFTTT applets from a test user account. By restricting applets to this account, we ensured they did not affect other users. Our experiments show that a malicious applet implementing the attack can exfiltrate sensitive information such as user location (using `Location` APIs as triggers and `Gmail` APIs as actions), and voice-assisted commands (using `Google Assistant` APIs as triggers and `Gmail` APIs as actions). Other services vulnerable to our timing attack include email subjects and conversations, social network private feeds, trip details on connected cars, or phone numbers and contact data. Our monitor detects the attack as expected.

Generally, filter code is inherently basic (typically tens of lines of code) and thus naturally within the reach of our monitor. Monitored executions take around 5 milliseconds, which is tolerable as IFTTT actions are allowed up to 15 minutes to execute [42].

6.2. Remote timing leaks in VJSC

We deployed the monitor to track remote timing leaks in the encryption routines in VJSC. As it is common in this setting, the client code is pre-loaded on a voting device. Hence, a network attacker has no reference point of when its execution has started. Several assets such as the vote, the randomness seed, or the client private key must be protected from remote timing leaks. We emulate the output of an encrypted vote by a public output representing the ciphertext. The encryption algorithms make heavy use of secret branching, yet perform no time reads. Our monitor detected no remote timing leaks when covering the main encryption routines in VJSC. The results indicate that our approach can be used to analyze real-world software. JSFlow is a security-enhanced JavaScript interpreter written itself in JavaScript and our monitor inherits JSFlow's performance penalties. Monitored executions take around 10 minutes, indicating the security monitor for cryptographically-heavy scenarios can be better suited for security testing rather than for deployment at runtime.

7. Related work

We discuss related work with respect to timing-sensitive information flow control, practical remote timing attacks, and information flow in IoT apps.

Timing-sensitive information flow control As mentioned earlier, previous approaches to timing-sensitive information flow control target specific types of timing leaks, having yet to address their combination. Agat [1] suggests closing timing leaks by a transformation that inserts dummy instructions and proves that well-typed padded programs satisfy a bisimulation-based security condition. He assumes a local attacker. Barthe et al. [13] propose to remove timing leaks as defined by Agat in [1] by using transaction mechanisms. Köpf and Basin [44] study information-theoretic metrics for adaptive side-channel attacks and analyze timing and power attacks on hardware implementations.

Askarov et al. [7] show how to mitigate timing leaks by a blackbox mechanism that inserts output delays as to bound the amount of information leaked via timing as a function of elapsed time. The approach is essentially based on quantizing the time of output. If the output is produced earlier it is buffered. If the output misses the deadline, the quantum is increased to control the leak bandwidth. The elegance of this approach is that it is independent of the types of timing flow, similarly to our enforcement. A drawback is that leaks are not prevented but instead "stretched" over time. Zhang et al. [62] build on this approach to provide language support for whitebox mitigation.

Pedersen and Askarov [46] treat timing leaks via garbage collectors. The time is formalized as the number of steps taken by the program and includes the steps made by the garbage collector. Their security definition is parametric in the maximum size of the heap, which determines when the garbage collector is invoked. Other timing channels, e.g., due to cache or JIT, are orthogonal to their approach.

Brennan et al. [21] investigate JIT-based leaks in JVM programs. They present a practical study that identifies vulnerability templates and analyzes some standard Java classes for JIT-based side channels.

Recall that internal timing leaks occur when the timing behavior of threads affects the interleaving of attacker-visible events via the scheduler. There are ways to prevent schedulers from internal timing leaks [48], [49], [57].

Cache-based timing attacks can be devastating for cryptographic implementations [2], [3]. A popular approach in preventing them is to target constant-time execution (e.g., [3], [4], [18], [36]). In particular, Almeida et al. [4] observe that constant time is only needed with respect to public output, thus gaining some expressiveness in the analysis, which still deals with the local attacker and specific timing channels. Barthe et al. [12] explore the problem of preserving side-channel countermeasures by compilation of cryptographic constant-time implementations. Their security property also considers an abstract leakage function, but in contrast to our work they assume a local attacker that knows when the program started its execution.

Ene et al. [32] build on the work of Almeida et al. [4] and propose a type system with output-sensitive constant-time guarantees, accompanied by a prototype to verify LLVM implementations. Their security condition models a local attacker, similarly to Almeida et al.

Rakotonirina and Köpf [47] study information aggregation over multiple timing measurements. Similarly to us, they observe that adversary capabilities are often excessively restrictive in formal models, mismatching settings of real-world attacks. They introduce a differential-time adversary, which enables reasoning about information aggregation and study quantitative effects of divide-and-conquer attacks. The differential-time adversary is useful for modeling a weaker adversary in the presence of noise in the time measurements, which makes sense in a remote setting. Note that our attacker may combine both differential- and absolute-time capabilities because programs have access to real-time

clocks. Vasilikos et al. [59] utilize time automata to study adversaries parametrized in the granularity of the clock.

Practical remote timing attacks Remote timing attacks are (still) practical [22], [23]. Felten and Schneider [33] exploit caching in browsers to leak the browsing history of web users. Bortz and Boneh [19] demonstrate cross-site timing attacks to learn whether the user is logged in to another site. Chen et al. [29] demonstrate how a vulnerable autocompletion mechanism in a healthcare web application leaks sensitive information about the user despite the HTTPS protection of the traffic.

Micro-architectural attacks [43] can be exploited remotely in a browser. High-precision timers, such as `performance.now()` in JavaScript exacerbate the problem [45]. Although browser vendors have moved to eliminate fine-grained timers from JavaScript, researchers have uncovered other ways to measure time [53], [54].

Information flow in IoT apps An active area of research is dedicated to securing IoT apps [10], [27]. Surbatovich et al. [58] present an empirical study of IFTTT apps and categorize the apps with respect to potential security and integrity violations.

FlowFence [34] dynamically enforces information flow control in IoT apps: the flows considered by FlowFence are the ones among Quarantined Modules (QMs). QMs are pieces of code (selected by the developer) that run in a sandbox. Because all the code is encapsulated inside QMs, implicit flows are not analyzed. They are eliminated since non-sensitive code cannot evaluate values returned by QMs. In contrast, Saint [25] tracks implicit flows leveraging standard static data flow analysis on an app's intermediate representation to track information flows from sensitive sources to external sinks. Timing leaks are outside the scope of both FlowFence and Saint. IoTGuard [26] is a monitoring mechanism for enforcing security policies written in the IoTGuard policy language. Security policies describe valid transitions in an IoT app execution. Although timing leaks are not discussed in the paper, we believe that security policies related to timing leaks can be modeled in the IoTGuard policy language by using events related to time. Bastys et al. [16], [17] develop dynamic and static information flow analyses in IoT apps. They establish termination-insensitive noninterference for their enforcement. Although their dynamic enforcement implements a timeout, modeling the timeout behavior of IFTTT applets, they do not deal with leaking information through timing channels in general and their language does not have access to the clock.

8. Conclusion

Cloud-based platforms, like those for IoT apps, are powered by remote code execution. These platforms routinely run third-party apps that have access to private information of their users. Even if these third-party apps are free of explicit and implicit insecure flows, malicious app makers can set up remote timing leaks to exfiltrate the private information. E-voting libraries utilize advanced cryptographic techniques, opening up for timing channels with respect to network attackers. Motivated by these scenarios, the paper puts the spotlight on the general problem of characterizing and ruling out remote timing attacks. We present an extensional security characterization that captures the essence of remote timing attacks. We propose Clockwork, a mechanism that rules out timing leaks without being overly restrictive. We achieve a high degree of permissiveness due to identifying patterns that leaky code must follow in order to successfully set up and exploit timing leaks. We demonstrate the feasibility of the approach by implementing the mechanism as an extension of JSFlow, a state-of-the-art information flow tracker for JavaScript, and evaluating it on case studies with IFTTT and VJSC.

Static analysis techniques to track remote timing attacks are a worthwhile subject for future investigations. Static analysis can help eliminate the runtime overhead and brings additional benefits, such as discarding sensitive-upgrade restrictions. On the other hand, static analysis faces challenges in estimating the time of program runs, for example, when a potential leak might happen before or after timeout. This is not an issue for a dynamic monitor that tracks leaks in a given run before it times out. With the above caveat, we believe the intuition in Figure 6 is directly suitable to be implemented in a static analysis.

Future work will also pursue further case studies and experiments to evaluate the precision and performance of the mechanism in practice. We are interested in instantiating the approach to other cloud-based remote code execution platforms.

Another promising avenue for future research is integrating our approach with secure multi-party computation. *Secure multi-party computation* (MPC) [28], [37] relies on cryptographic primitives not to leak private information when potentially untrusted code operates on encrypted data of a user. Monitoring cloud-based MPC implementations along the lines of our approach has potential to detect and rule out implementation-level timing attacks.

Acknowledgements Thanks are due to Marco Vassena and the anonymous reviewers for feedback on early results of this work. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, by the ANR17-CE25-0014-01 CISC project, and by the Inria Project Lab SPAI. It was also partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

References

[1] J. Agat. Transforming out timing leaks. In *POPL*, 2000.

[2] M. R. Albrecht and K. G. Paterson. Lucky microseconds: A timing attack on amazon's s2n implementation of TLS. In *EUROCRYPT*, 2016.

[3] N. J. AlFardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *S&P*, 2013.

[4] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *USENIX Sec.*, 2016.

[5] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, 2008.

[6] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *S&P*, 2007.

[7] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *CCS*, 2010.

[8] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS*, 2009.

[9] M. Balliu. A logic for information flow analysis of distributed programs. In *NordSec*, 2013.

[10] M. Balliu, I. Bastys, and A. Sabelfeld. Securing IoT Apps. *IEEE Security & Privacy Magazine*, 2019.

[11] M. Balliu, D. Schoepe, and A. Sabelfeld. We are family: Relating information-flow trackers. In *ESORICS*, 2017.

[12] G. Barthe, B. Grégoire, and V. Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *CSF*, 2018.

[13] G. Barthe, T. Rezk, and A. Saabas. Proof obligations preserving compilation. In *FAST*, 2005.

[14] G. Barthe, T. Rezk, and M. Warnier. Preventing timing leaks through transactional branching instructions. *QAPL*, 2006.

[15] I. Bastys, M. Balliu, T. Rezk, and A. Sabelfeld. Clockwork: Tracking Remote Timing Attacks. Full version and code. https://www.cse.chalmers.se/research/group/security/remote-timing/, May 2020.

[16] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In *CCS*, 2018.

[17] I. Bastys, F. Piessens, and A. Sabelfeld. Tracking Information Flow via Delayed Output - Addressing Privacy in IoT and Emailing Apps. In *NordSec*, 2018.

[18] D. J. Bernstein. Cache-timing attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[19] A. Bortz and D. Boneh. Exposing private information by timing web applications. In *WWW*, 2007.

[20] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *TCS*, 2002.

[21] T. Brennan, N. Rosner, and T. Bultan. JIT Leaks: Inducing Timing Side Channels through Just-In-Time Compilation. In *S&P*, 2020.

[22] B. B. Brumley and N. Tuveri. Remote timing attacks are still practical. In *ESORICS*, 2011.

[23] D. Brumley and D. Boneh. Remote timing attacks are practical. In *USENIX Sec.*, 2003.

[24] R. G. Catalin Dima, Constantin Enea. Nondeterministic noninterference and deducible information flow. Technical report, University of Paris 12, LACL, 2006. Technical Report 200601.

[25] Z. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. D. McDaniel, and A. S. Uluagac. Sensitive information tracking in commodity iot. In *USENIX Sec.*, 2018.

[26] Z. Celik, G. Tan, and P. D. M. and. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *NDSS*, 2019.

[27] Z. B. Celik, E. Fernandes, E. Pauley, G. Tan, and P. D. McDaniel. Program analysis of commodity iot applications for security and privacy: Challenges and opportunities. *ACM Computing Surveys*, 2019.

[28] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In *ACM Symposium on Theory of Computing*, 1988.

[29] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *S&P*, 2010.

[30] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 1977.

[31] Douglas Wikström. Open Verificatum Project. https://www.verificatum.org, 2020.

[32] C. Ene, L. Mounier, and M. Potet. Output-sensitive information flow analysis. In *FORTE*, 2019.

[33] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *CCS*, 2000.

[34] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX Sec.*, 2016.

[35] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *NDSS*, 2018.

[36] C. P. Garca, B. B. Brumley, and Y. Yarom. 'Make Sure DSA Signing Exponentiations Really are Constant-Time". https://eprint.iacr.org/2016/594, 2016.

[37] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, 1987.

[38] G. L. Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *ASIAN*, 2006.

[39] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs. *JCS*, 2016.

[40] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*, 2014.

[41] D. Hedin and A. Sabelfeld. Information-Flow Security for a Core of JavaScript. In *CSF*, 2012.

[42] IFTTT: If This Then That. https://ifttt.com, 2020.

[43] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[44] B. Köpf and D. A. Basin. An information-theoretic model for adaptive side-channel attacks. In *CCS*, 2007.

[45] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *CCS*, 2015.

[46] M. V. Pedersen and A. Askarov. From trash to treasure: Timing-sensitive garbage collection. In *S&P*, 2017.

[47] I. Rakotonirina and B. Köpf. On aggregation of information in timing attacks. In *EuroS&P*, 2019.

[48] A. Russo, J. Hughes, D. A. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *ASIAN*, 2006.

[49] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *CSFW*, 2006.

[50] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[51] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW*, 2000.

[52] I. Sánchez-Rola, I. Santos, and D. Balzarotti. Clock around the clock: Time-based device fingerprinting. In *CCS*, 2018.

[53] M. Schwarz, M. Lipp, and D. Gruss. Javascript zero: Real javascript and zero side-channel attacks. In *NDSS*, 2018.

[54] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *FC*, 2017.

[55] G. Smith. A new type system for secure information flow. In *CSFW*, 2001.

[56] G. Smith and D. M. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL*, 1998.

[57] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *ICFP*, 2012.

[58] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes. In *WWW*, 2017.

[59] P. Vasilikos, H. R. Nielson, F. Nielson, and B. Köpf. Timing leaks and coarse-grained clocks. In *CSF*, 2019.

[60] D. M. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *JCS*, 1999.

[61] S. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, Ithaca, NY, USA, 2002.

[62] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *PLDI*, 2012.

Appendix

The appendix defines the equivalence relations on memories and configurations, states the auxiliary lemmas, and reports the full proof of the soundness theorem.

Definition 6 (Memory equivalence)**.** Two memories m_1 and m_2 are equivalent with respect to a type environment Γ, denoted $m_1 \sim_\Gamma m_2$, iff $\forall x \in \Gamma.\ \Gamma(x) = \mathtt{L} \Rightarrow m_1(x) = m_2(x)$.

Definition 7 (Weak configuration equivalence)**.** Two monitor configurations cfg_1 and cfg_2 are equivalent, denoted by $cfg_1 \sim cfg_2$, iff $cfg_i = \langle c, m_i, hst_i, st_i = (\mathtt{L}^m, P, \Gamma, T_i, Q)\rangle$, for $i = \{1, 2\}$ and $m_1 \sim_\Gamma m_2$.

Definition 8 (Strong configuration equivalence)**.** Two monitor configurations cfg_1 and cfg_2 are equivalent, denoted by $cfg_1 \approx cfg_2$, iff $cfg_i = \langle c, m_i, hst, st = (\mathtt{L}^m, P, \Gamma, T, Q)\rangle$, for $i = \{1, 2\}$ and $m_1 \sim_\Gamma m_2$.

Lemma 2 (No-out)**.** *Suppose $cfg = \langle c, m, hst, st\rangle$ so that $P = \mathtt{H}$ and we have $Q = \mathtt{true}$ or $T = \mathtt{true}$ in st. Then cfg will not produce further low output: if $cfg \overset{O}{\rightarrow}{}^*$ then $O|_\mathtt{L} = \emptyset$.*

Lemma 3 (Single output)**.** *Suppose $cfg = \langle c, m, hst, st\rangle$ so that $P = \mathtt{H}$, $Q = \mathtt{false}$, and $T = \mathtt{false}$ in st. Then cfg will produce at most one further low output: if $cfg \overset{O}{\rightarrow}{}^*$ then either $O|_\mathtt{L} = \emptyset$ or $O|_\mathtt{L} = o$ for some non-empty output o.*

Lemma 4 (Confinement)**.** *If $cfg_i \rightarrow cfg_i' \rightarrow^* cfg_i'' \rightarrow cfg_i'''$ for $i = \{1, 2\}$, $cfg_1 \sim cfg_2$, and $S_i = \mathtt{L}^m$ for some m, $S_i' = \mathtt{L}^m :: \mathtt{H}$, the stack stays high in all transitions $cfg_i' \rightarrow^* cfg_i''$ and $S_i''' = \mathtt{L}^m$, then $cfg_1''' \sim cfg_2'''$.*

Lemma 5 (Strong lockstep)**.** *Let $cfg_i = \langle c, m_i, hst_i, st_i\rangle$, $i = \{1, 2\}$ be two monitor configuration states such that $cfg_1 \approx cfg_2$ and $P_i = \mathtt{L}$. If $cfg_1 \overset{o}{\rightarrow} cfg_1'$ (where o is either ϵ or low output $(v, t)_\mathtt{L}$) and $P_1' = \mathtt{L}$ then $cfg_2 \overset{o}{\rightarrow} cfg_2'$ and $cfg_1' \approx cfg_2'$.*

Lemma 6 (Weak lockstep)**.** *Let $cfg_i = \langle c, m_i, hst_i, st_i\rangle$, $i = \{1, 2\}$ be two monitor configuration states such that $cfg_1 \sim cfg_2$, $P_i = \mathtt{H}$, and $lev(S_i) = \mathtt{L}$. If $cfg_1 \overset{o_1}{\rightarrow} cfg_1'$ (where o_1 is either ϵ or low output $(v, t_1)_\mathtt{L}$) for some t and t_1 then $cfg_2 \overset{o_2}{\rightarrow} cfg_2'$, $o_2 = (v, t_2)_\mathtt{L}$ for some t_2 and $cfg_1' \sim cfg_2'$.*

Theorem 1 (Soundness)**.** *Given timeout and stmp(), for any program c, initial memory m_0, and timestamp t_0, if $\langle c, m_0, t_0, st_0\rangle \overset{O}{\Rightarrow}{}^* \langle c', m, hst, st\rangle$ and $O|_\mathtt{L} = O_\mathtt{L} :: o$, then $k(c, m_0^\mathtt{L}, O_\mathtt{L} :: o) \supseteq k(c, m_0^\mathtt{L}, O_\mathtt{L}) \setminus tk(c, m_0^\mathtt{L}, O_\mathtt{L})$.*

Proof. By contradiction. Assuming the inverse of $k(c, m_0^\mathtt{L}, O_\mathtt{L} :: o) \supseteq k(c, m_0^\mathtt{L}, O_\mathtt{L}) \setminus tk(c, m_0^\mathtt{L}, O_\mathtt{L})$, there exists $m_2 = m_0^\mathtt{L} \uplus m_2^\mathtt{H}$ such that $m_2^\mathtt{H} \in k(c, m_0^\mathtt{L}, O_\mathtt{L}) \setminus tk(c, m_0^\mathtt{L}, O_\mathtt{L})$, but $m_2^\mathtt{H} \notin k(c, m_0^\mathtt{L}, O_\mathtt{L} :: o)$. At the same time, because $\langle c, m_0, t_0, st_0\rangle \overset{O}{\Rightarrow}{}^* \langle c', m, hst, st\rangle$, there exists $m_1 = m_0^\mathtt{L} \uplus m_1^\mathtt{H}$ (and hence $m_1 \sim_{\Gamma_0} m_2$) so that $m_1^\mathtt{H} \in k(c, m_0^\mathtt{L}, O_\mathtt{L} :: o)$ and $m_1 \notin tk(c, m_0^\mathtt{L}, O_\mathtt{L})$, implying $m_1^\mathtt{H} \in k(c, m_0^\mathtt{L}, O_\mathtt{L}) \setminus tk(c, m_0^\mathtt{L}, O_\mathtt{L})$.

To establish contradiction, we prove that the sequence of low outputs $O_\mathtt{L} :: o$ of the monitored execution in m_1 is mirrored by equivalent configurations in the monitored execution that originates from m_2.

The first observation is that because $m_1^\mathtt{H}, m_2^\mathtt{H} \notin tk(c, m_0^\mathtt{L}, O_\mathtt{L})$, executions on both memories do not get stuck or timeout after producing low output sequences $O_\mathtt{L}$. Moreover, the execution on $m_1^\mathtt{H}$ produces $O_\mathtt{L} :: o$ within the timeout and the execution on $m_2^\mathtt{H}$ either produces $O_\mathtt{L}$ and terminates within the timeout or produces $O_\mathtt{L} :: o'$ within the timeout for some o'. In the latter case, assume $o = (v_1, t_1')_\mathtt{L}$ and $o' = (v_2, t_2')_\mathtt{L}$. Note that v_1 must be different from v_2, otherwise we contradict $m_2^\mathtt{H} \notin k(c, m_0^\mathtt{L}, O_\mathtt{L} :: o)$.

Let $cfg_i = \langle c, m_i, t_i, st_0\rangle$ for $i = \{1, 2\}$ where $t_0 = t_1$, and t_2 is not necessarily the same as t_0.

Recall that part P of the security configurations records the first time the computation enters high context. As such, it can flip from L to H only once. In this light, the run originating from m_1 can be viewed as $cfg_1 \overset{O_1}{\rightarrow}{}^* cfg_1' \rightarrow cfg_1'' \overset{O_1''}{\rightarrow}{}^* cfg_1'''$ where $P_1' = \mathtt{L}$ and $P_1'' = \mathtt{H}$ (the primes and indices in the security variables make it clear which configuration we extract the security state from) for some O_1 and O_1'' so that $O_1|_\mathtt{L} = O_\mathtt{L}'$. Parameter P can only be flipped by a branching command, hence there is no output in the transition from cfg_1' to cfg_1''.

We can therefore apply the strong lockstep lemma (Lemma 5) to the run up to cfg_1', yielding that the run originating from m_2 can be viewed as $cfg_2 \overset{O_2}{\rightarrow}{}^* cfg_2' \rightarrow cfg_2'' \overset{O_2''}{\rightarrow}{}^* cfg_2'''$ where $cfg_1' \approx cfg_2'$ for some O_2 and O_2'' so that $O_2|_\mathtt{L} = O_\mathtt{L}'$. Note that the lemma guarantees that the same low outputs $O_\mathtt{L}'$ with the same timestamps.

We have cases on Q_1' and T_1', which must be the same as Q_2' and T_2' due to $cfg_1' \approx cfg_2'$.

If $Q_1' = \mathtt{true}$ or $T_1' = \mathtt{true}$, then we apply the no-out lemma (Lemma 2) which guarantees that neither cfg_1' nor cfg_2' will produce low output. Then $O_\mathtt{L}' = O_\mathtt{L} :: o$, produced by both runs. This implies $o = o'$, arriving at contradiction.

If $Q_1' = \mathtt{false}$ and $T_1' = \mathtt{false}$, then no output or time reads have yet taken place. By single-output lemma (Lemma 3), there will be at most one low output when running cfg_1'. Thus, $O_\mathtt{L}' = \epsilon$. In this case the executions

have the following pattern of alternating between high and low context:

$$
\begin{array}{l}
cfg_1' \rightarrow cfg_1'' \rightarrow^* cfg_1^{iv} \rightarrow cfg_1^{v} \rightarrow^* \ldots\ cfg_1^{n} \xrightarrow{o} cfg_1''' \\
\sim_{(1)} \qquad\qquad\qquad\qquad\qquad \sim_{(2)} \qquad\qquad \sim_{(3)} \\
cfg_2' \rightarrow cfg_2'' \rightarrow^* cfg_2^{iv} \rightarrow cfg_2^{v} \rightarrow^* \ldots\ cfg_2^{n} \xrightarrow{o'} cfg_2'''
\end{array}
$$

where there is a high pc on the stack in all configurations of the run from cfg_1'' to cfg_1^{iv}, there is no high pc in any configurations of the run from cfg_1^{v}, and so on, finally reaching cfg_1^{n}. Note that $lev(S_1^n) = \mathtt{L}$ because low output is not allowed in high pc.

We now show that there is a matching run from cfg_2 with equivalent configurations, obtaining equivalences (1)–(3).

Indeed, equivalence $cfg_1' \sim cfg_2'$ (1) follows from the stronger equivalence $cfg_1' \approx cfg_2'$. Equivalence $cfg_1^{v} \sim cfg_1^{v}$ (2) follows from the confinement lemma (Lemma 4): $\Gamma' = \Gamma^v$ and $m_i' \sim_{\Gamma^v} m_i^v$ for $i = \{1, 2\}$. It follows that $m_1^v \sim_{\Gamma^v} m_2^v$. The lemma also guarantees that $Q_i' = Q_i^v$ for $i = \{1, 2\}$. Clearly, the monotone P parameter has remained unchanged in both configurations remaining $\mathtt{H}$. We thus obtain the equivalence of the resulting configurations $cfg_1^v \sim cfg_2^v$.

Equivalence $cfg_1^n \sim cfg_1^n$ (3) follows from the weak lockstep lemma (Lemma 6). Hence $m_1^n \sim_{\Gamma^n} m_2^n$. Note that the equivalence ensures that cfg_1^n and cfg_2^n have the same commands. Both must be then outputs.

Assuming as before $o = (v_1, t_1')_{\mathtt{L}}$ and $o' = (v_2, t_2')_{\mathtt{L}}$, we get $v_1 = v_2 = v$ from SEC-OUTPUT-2 due to $m_1^n \sim_{\Gamma^n} m_2^n$.

Note that although t_1 and t_2 can be different, both m_1 and m_2 were both able to produce low output v, resulting in contradiction. □