10-2019

Parametric timed model checking for guaranteeing timed opacity

Étienne ANDRÉ

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Citation

ANDRÉ, Étienne and SUN, Jun. Parametric timed model checking for guaranteeing timed opacity. (2019). *Proceedings of the 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31*. 115-130.
Available at: https://ink.library.smu.edu.sg/sis_research/4966

Parametric Timed Model Checking for Guaranteeing Timed Opacity

Étienne André[1,2,3](✉) and Jun Sun[4]

[1] Université Paris 13, LIPN, CNRS, UMR 7030,
93430 Villetaneuse, France
eandre93430@lipn13.fr
[2] JFLI, CNRS, Tokyo, Japan
[3] National Institute of Informatics, Tokyo, Japan
[4] School of Information Systems,
Singapore Management University,
Singapore, Singapore

Abstract. Information leakage can have dramatic consequences on systems security. Among harmful information leaks, the timing information leakage is the ability for an attacker to deduce internal information depending on the system execution time. We address the following problem: given a timed system, synthesize the execution times for which one cannot deduce whether the system performed some secret behavior. We solve this problem in the setting of timed automata (TAs). We first provide a general solution, and then extend the problem to parametric TAs, by synthesizing internal timings making the TA secure. We study decidability, devise algorithms, and show that our method can also apply to program analysis.

1 Introduction

Timed systems combine concurrency and possibly hard real-time constraints. Information leakage can have dramatic consequences on the security of such systems. Among harmful information leaks, the *timing information leakage* is the ability for an attacker to deduce internal information depending on timing information. In this work, we focus on the execution time, i. e., when a system works as an almost black-box, with the ability of an attacker to mainly observe its execution time.

We address the following problem: given a timed system, a private state denoting the execution of some secret behavior and a final state denoting the completion of the execution, synthesize the execution times to the final state for which one cannot deduce whether the system has passed through the private state. We solve this problem in the setting of timed automata (TAs), which is

This work is partially supported by the ANR national research program PACS (ANR-14-CE28-0002) and by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST.

Y.-F. Chen et al. (Eds.): ATVA 2019, LNCS 11781, pp. 115–130, 2019.
https://doi.org/10.1007/978-3-030-31784-3_7

a popular extension of finite-state automata with clocks [2]. We first prove that this problem is solvable, and we provide an algorithm, that we implement and apply to a set of benchmarks containing notably a set of Java programs known for their (absence of) timing information leakage.

Then we consider a higher-level problem by allowing (internal) timing parameters in the system, that can model uncertainty or unknown constants at early design stage. The setting becomes parametric timed automata [3], and the problem asks: given a timed system with timing parameters, a private state and a final state, synthesize the timing parameters and the execution times for which one cannot deduce whether the system has passed through the private state. Although we show that the problem is in general undecidable, we provide a decidable subclass; then we devise a general procedure not guaranteed to terminate, but that behaves well on examples from the literature.

2 Related Works

This work is closely related to the line of work on defining and analyzing information flow in timed automata. It is well-known (see e. g., [8,16]) that time is a potential attack vector against secure systems. That is, it is possible that a non-interferent (secure) system can become interferent (insecure) when timing constraints are added [13]. In [7], a first notion of *timed* non-interference is proposed. In [13], Gardey *et al.* define timed strong non-deterministic non-interference (SNNI) based on timed language equivalence between the automaton with hidden low-level actions and the automaton with removed low-level actions. Furthermore, they show that the problem of determining whether a timed automaton satisfies SNNI is undecidable. In contrast, timed cosimulation-based SNNI, timed bisimulation-based SNNI and timed state SNNI are decidable. In [9], the problem of checking opacity for timed automata is considered: even for the restricted class of event-recording automata, it is undecidable whether a system is opaque, i. e., whether an attacker can deduce whether some set of actions was performed, by only observing a given set of observable actions (with their timing). In [19], Vasilikos *et al.* define the security of timed automata in term of information flow using a bisimulation relation and develop an algorithm for deriving a sound constraint for satisfying the information flow property locally based on relevant transitions. In [8], Benattar *et al.* study the control synthesis problem of timed automata for SNNI. That is, given a timed automaton, they propose a method to automatically generate a (largest) sub-systems such that it is non-interferent if possible. Different from the above-mentioned work, our work considers parametric timed automata, i. e., timed systems with unknown design parameters, and focuses on synthesizing parameter valuations which guarantee information flow property. As far as we know, this is the first work on parametric model checking for timed automata for information flow property. Compared to [8], our approach is more realistic as it does not require change of program structure. Rather, our result provides guidelines on how to choose the timing parameters (e. g., how long to wait after certain program statements) for avoiding information leakage.

In [18], the authors propose a type system dealing with non-determinism and (continuous) real-time, the adequacy of which is ensured using non-interference. We share the common formalism of TA; however, we mainly focus on leakage as execution time, and we *synthesize* internal parts of the system (clock guards), in contrast to [18] where the system is fixed.

This work is related to work on mitigating information leakage through time side channel. For example, in [20], Wang *et al.* proposed to automatically generate masking code for eliminating side channel through program synthesis. In [21], Wu *et al.* proposed to eliminate time side channel through program repair. Different from the above-mentioned works, we reduce the problem of mitigating time side channel as a parametric model checking problem and solve it using parametric reachability analysis techniques.

This work is related to work on identifying information leakage through timing analysis. In [10], Chattopadhyay *et al.* applied model checking to perform cache timing analysis. In [11], Chu *et al.* performed similar analysis through symbolic execution. In [1], Abbasi *et al.* apply the NuSMV model checker to verify integrated circuits against information leakage through side channels. In [12], a tool is developed to identify time side channel through static analysis. In [22], Sung *et al.* developed a framework based on LLVM for cache timing analysis.

3 Preliminaries

We assume a set $\mathbb{X} = \{x_1, \dots, x_H\}$ of *clocks*, i. e., real-valued variables that evolve at the same rate. A clock valuation is $\mu : \mathbb{X} \rightarrow \mathbb{R}_{\geq 0}$. We write $\mathbf{0}$ for the clock valuation assigning 0 to all clocks. Given $d \in \mathbb{R}_{\geq 0}$, $\mu + d$ is s.t. $(\mu + d)(x) = \mu(x) + d$, for all $x \in \mathbb{X}$. Given $R \subseteq \mathbb{X}$, we define the *reset* of a valuation μ, denoted by $[\mu]_R$, as follows: $[\mu]_R(x) = 0$ if $x \in R$, and $[\mu]_R(x) = \mu(x)$ otherwise.

We assume a set $\mathbb{P} = \{p_1, \dots, p_M\}$ of *parameters*. A parameter *valuation* v is $v : \mathbb{P} \rightarrow \mathbb{Q}_+$. We assume $\bowtie \in \{<, \leq, =, \geq, >\}$. A guard g is a constraint over $\mathbb{X} \cup \mathbb{P}$ defined by a conjunction of inequalities of the form $x \bowtie \sum_{1 \leq i \leq M} \alpha_i p_i + d$, with $p_i \in \mathbb{P}$, and $\alpha_i, d \in \mathbb{Z}$. Given g, we write $\mu \models v(g)$ if the expression obtained by replacing each x with $\mu(x)$ and each p with $v(p)$ in g evaluates to true.

Definition 1 (PTA). *A PTA $\mathcal{A}$ is a tuple $\mathcal{A} = (\Sigma, L, \ell_0, \mathbb{X}, \mathbb{P}, I, E)$, where: (i) Σ is a finite set of actions, (ii) L is a finite set of locations, (iii) $\ell_0 \in L$ is the initial location, (iv) $\mathbb{X}$ is a finite set of clocks, (v) $\mathbb{P}$ is a finite set of parameters, (vi) I is the invariant, assigning to every $\ell \in L$ a guard $I(\ell)$, (vii) E is a finite set of edges $e = (\ell, g, a, R, \ell')$ where $\ell, \ell' \in L$ are the source and target locations, $a \in \Sigma$, $R \subseteq \mathbb{X}$ is a set of clocks to be reset, and g is a guard.*

Example 1. Consider the PTA in Fig. 1 (inspired by [13, Fig. 1b]), containing one clock x and two parameters p_1 and p_2. ℓ_0 is the initial location, while ℓ_1 is the (only) accepting location.

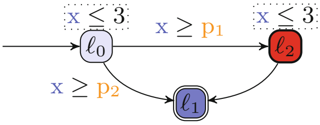

Fig. 1. A PTA example

Given v, we denote by $v(\mathcal{A})$ the non-parametric structure where all occurrences of a parameter p_i have been replaced by $v(p_i)$.

The *synchronous product* (using strong broadcast, i. e., synchronization on a given set of actions) of several PTAs gives a PTA.

Definition 2 (Semantics of a TA). *Given a PTA $\mathcal{A} = (\Sigma, L, \ell_0, \mathbb{X}, \mathbb{P}, I, E)$, and a parameter valuation v, the semantics of $v(\mathcal{A})$ is given by the timed transition system (TTS) $(S, s_0, \rightarrow)$, with*

- $S = \{(\ell, \mu) \in L \times \mathbb{R}^H_{\geq 0} \mid \mu \models v(I(\ell))\}$, $s_0 = (\ell_0, \mathbf{0})$,
- $\rightarrow$ *consists of the discrete and (continuous) delay transition relations: (i) discrete transitions: $(\ell, \mu) \overset{e}{\mapsto} (\ell', \mu')$, if $(\ell, \mu), (\ell', \mu') \in S$, and there exists $e = (\ell, g, a, R, \ell') \in E$, such that $\mu' = [\mu]_R$, and $\mu \models v(g)$. (ii) delay transitions: $(\ell, \mu) \overset{d}{\mapsto} (\ell, \mu + d)$, with $d \in \mathbb{R}_{\geq 0}$, if $\forall d' \in [0, d], (\ell, \mu + d') \in S$.*

Moreover we write $(\ell, \mu) \xrightarrow{(e,d)} (\ell', \mu')$ for a combination of a delay and discrete transition if $\exists \mu'' : (\ell, \mu) \overset{d}{\mapsto} (\ell, \mu'') \overset{e}{\mapsto} (\ell', \mu')$.

Given a TA $v(\mathcal{A})$ with concrete semantics $(S, s_0, \rightarrow)$, we refer to the states of S as the *concrete states* of $v(\mathcal{A})$. A *run* of $v(\mathcal{A})$ is an alternating sequence of concrete states of $v(\mathcal{A})$ and pairs of edges and delays starting from the initial state s_0 of the form $s_0, (e_0, d_0), s_1, \cdots$ with $i = 0, 1, \ldots$, $e_i \in E$, $d_i \in \mathbb{R}_{\geq 0}$ and $s_i \xrightarrow{(e_i, d_i)} s_{i+1}$. The *duration* of a finite run $\rho : s_0, (e_0, d_0), s_1, \cdots, s_i$ is $dur(\rho) = \sum_{0 \leq j \leq i-1} d_j$. Given $s = (\ell, \mu)$, we say that s is reachable in $v(\mathcal{A})$ if s appears in a run of $v(\mathcal{A})$. By extension, we say that ℓ is reachable. Given $\ell, \ell' \in L$ and a run ρ, we say that ℓ is reachable on the way to ℓ' in ρ if ρ is of the form $(\ell_0), (e_0, d_0), \cdots, (e_n, d_n), \cdots (e_m, d_m) \cdots$ for some $m, n \in \mathbb{N}$ such that $\ell_n = \ell$, $\ell_m = \ell'$ and $\forall 0 \leq i \leq n-1, \ell_i \neq \ell'$. Conversely, ℓ is unreachable on the way to ℓ' in ρ if ρ is of the form $(\ell_0), (e_0, d_0), \cdots, (e_m, d_m) \cdots$ with $\ell_m = \ell'$ and $\forall 0 \leq i \leq m-1, \ell_i \neq \ell$.

Example 2. Consider again the PTA $\mathcal{A}$ in Fig. 1, and let v be such that $v(p_1) = 1$ and $v(p_2) = 2$. Consider the following run ρ of $v(\mathcal{A})$: $(\ell_0, x = 0), (e_2, 1.4), (\ell_2, x = 1.4), (e_3, 1.3), (\ell_1, x = 2.7)$, where e_2 is the edge from ℓ_0 to ℓ_2 in Fig. 1, and e_3 is the edge from ℓ_2 to ℓ_1. We write "$x = 1.4$" instead of "μ such that $\mu(x) = 1.4$". We have $dur(\rho) = 1.4 + 1.3 = 2.7$. In addition, ℓ_2 is reachable on the way to ℓ_1 in ρ.

We will use reachability synthesis to solve the problems in Sect. 4. This procedure, called EFsynth, takes as input a PTA $\mathcal{A}$ and a set of target locations T,

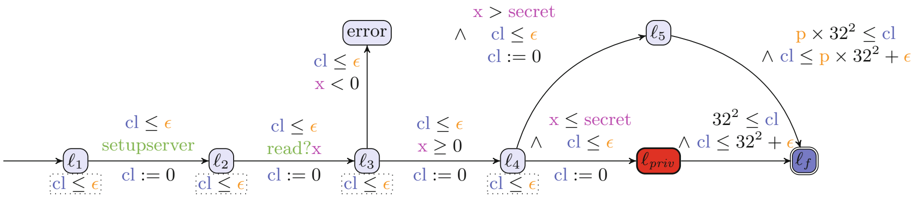

Fig. 2. A Java program encoded in a PTA

and attempts to synthesize all parameter valuations v for which T is reachable in $v(\mathcal{A})$. $\mathsf{EFsynth}(\mathcal{A}, T)$ was formalized in e. g., [15] and is a procedure that may not terminate, but that computes an exact result (sound and complete) if it terminates. $\mathsf{EFsynth}$ traverses the *parametric zone graph* of $\mathcal{A}$, which is a potentially infinite extension of the well-known zone graph of TAs (see, e. g., [5,15]).

Example 3. Consider again the PTA $\mathcal{A}$ in Fig. 1. $\mathsf{EFsynth}(\mathcal{A}, \{\ell_1\}) = p_1 \leq 3 \vee p_2 \leq 3$. Intuitively, it corresponds to all parameter constraints in the parametric zone graph associated to symbolic states with location ℓ_1.

4 Timed-Opacity Problems

Let us first introduce two key concepts to define our notion of opacity. $\mathit{DReach}^{v(\mathcal{A})}_{\ell}(\ell')$ (resp. $\mathit{DReach}^{v(\mathcal{A})}_{\neg\ell}(\ell')$) is the set of the durations of the runs for which ℓ is reachable (resp. unreachable) on the way to ℓ'. Formally: $\mathit{DReach}^{v(\mathcal{A})}_{\ell}(\ell') = \{d \mid \exists \rho \text{ in } v(\mathcal{A}) \text{ such that } d = \mathit{dur}(\rho) \wedge \ell \text{ is reachable on the way to } \ell' \text{ in } \rho\}$ and $\mathit{DReach}^{v(\mathcal{A})}_{\neg\ell}(\ell') = \{d \mid \exists \rho \text{ in } v(\mathcal{A}) \text{ such that } d = \mathit{dur}(\rho) \wedge \ell \text{ is unreachable on the way to } \ell' \text{ in } \rho\}$.

Example 4. Consider again the PTA in Fig. 1, and let v be such that $v(p_1) = 1$ and $v(p_2) = 2$. We have $\mathit{DReach}^{v(\mathcal{A})}_{\ell_2}(\ell_1) = [1, 3]$ and $\mathit{DReach}^{v(\mathcal{A})}_{\neg\ell_2}(\ell_1) = [2, 3]$.

Definition 3 (timed opacity w.r.t. D). *Given a TA $v(\mathcal{A})$, a private location ℓ_{priv}, a target location ℓ_f and a set of execution times D, we say that $v(\mathcal{A})$ is* opaque w.r.t. ℓ_{priv} on the way to ℓ_f for execution times D *if* $D \subseteq \mathit{DReach}^{v(\mathcal{A})}_{\ell_{priv}}(\ell_f) \cap \mathit{DReach}^{v(\mathcal{A})}_{\neg\ell_{priv}}(\ell_f)$.

Example 5. Consider the PTA $\mathcal{A}$ in Fig. 2 where cl is a clock, while ϵ, p are parameters. We use a sightly extended PTA syntax: read?x reads the value input on a given channel read, and assigns it to a (discrete, global) variable x. secret is a constant variable of arbitrary value. If both x and secret are finite-domain variables (e. g., bounded integers) then they can be seen as syntactic sugar for locations. Such variables are supported by most model checkers, including Uppaal and IMITATOR.

This PTA encodes a server process from the DARPA Space/Time Analysis for Cybersecurity (STAC) library, that compares a user-input variable with a given secret and performs different actions taking different times depending on this secret. In our encoding, a single instruction takes a time in $[0, \epsilon]$, while p is a (parametric) factor to one of the `sleep` instructions of the program (originally, $v(p) = 2$). For sake of simplicity, we abstract away instructions not related to time, and merge subfunctions calls.

Fix $v(\epsilon) = 1$, $v(p) = 2$. For this example, $\mathit{DReach}^{v(\mathcal{A})}_{\ell_{priv}}(\ell_f) = [1024, 1029]$ while $\mathit{DReach}^{v(\mathcal{A})}_{\neg\ell_{priv}}(\ell_f) = [2048, 2053]$. Therefore, $v(\mathcal{A})$ is opaque w.r.t. ℓ_{priv} on the way to ℓ_f for execution times $D = [1024, 1029] \cap [2048, 2053] = \emptyset$.

Now fix $v(\epsilon) = 2$, $v(p) = 1.002$. $\mathit{DReach}^{v(\mathcal{A})}_{\ell_{priv}}(\ell_f) = [1024, 1034]$ while $\mathit{DReach}^{v(\mathcal{A})}_{\neg\ell_{priv}}(\ell_f) = [1026.048, 1036.048]$. Therefore, $v(\mathcal{A})$ is opaque w.r.t. ℓ_{priv} on the way to ℓ_f for execution times $D = [1026.048, 1034]$.

We can now define the timed-opacity computation problem, which consists in computing the possible execution times ensuring opacity w.r.t. a private location. In other words, the attacker model is as follows: the attacker has only access to the computation time between the start of the program and the time it reaches a given (final) location.

Timed-opacity Computation Problem:
Input: A TA $v(\mathcal{A})$, a private location ℓ_{priv}, a target location ℓ_f
Problem: Compute the execution times D for which $v(\mathcal{A})$ is opaque w.r.t. ℓ_{priv} on the way to ℓ_f for execution times D

The synthesis counterpart allows for a higher-level problem by also synthesizing the internal timings guaranteeing opacity.

Timed-opacity Synthesis Problem:
Input: A PTA $\mathcal{A}$, a private location ℓ_{priv}, a target location ℓ_f
Problem: Synthesize the parameter valuations v and the execution times D for which $v(\mathcal{A})$ is opaque w.r.t. ℓ_{priv} on the way to ℓ_f for execution times D

Note that the execution times can depend on the parameter valuations.

5 Timed-Opacity Computation for Timed Automata

5.1 Answering the Timed-Opacity Computation Problem

Proposition 1 (timed-opacity computation). *The timed-opacity computation problem is solvable for TAs.*

This positive result can be put in perspective with the negative result of [9], that proves that it is undecidable whether a TA (and even the more restricted subclass of event-recording automata) is opaque, in a sense that the attacker can deduce some actions, by looking at observable actions together with their timing. The difference in our setting is that only the global time is observable, which can be seen as a single action, occurring once only at the end of the computation. In other words, our attacker is less powerful than the attacker in [9].

5.2 Checking for Timed-Opacity

If one does not have the ability to tune the system (i. e., change internal delays, or add some `sleep()` or `Wait()` statements in the program), one may be first interested in knowing whether the system is opaque for all execution times.

Definition 4 (timed opacity). *Given a TA $v(\mathcal{A})$, a private location ℓ_{priv} and a target location ℓ_f, we say that $v(\mathcal{A})$ is* opaque w.r.t. ℓ_{priv} on the way to ℓ_f *if* $\mathit{DReach}^{v(\mathcal{A})}_{\ell_{priv}}(\ell_f) = \mathit{DReach}^{v(\mathcal{A})}_{\neg\ell_{priv}}(\ell_f)$.

That is, a system is opaque if, for any execution time d, a run of duration d reaches ℓ_f after passing by ℓ_{priv} iff another run of duration d reaches ℓ_f without passing by ℓ_{priv}.

Remark 1. This definition is symmetric: a system is not opaque iff an attacker can deduce ℓ_{priv} or $\neg\ell_{priv}$. For instance, if there is no path through ℓ_{priv} to ℓ_f, but a path to ℓ_f, a system is not opaque w.r.t. Definition 4.

As we have a procedure to compute $\mathit{DReach}^{v(\mathcal{A})}_{\ell_{priv}}(\ell_f)$ and $\mathit{DReach}^{v(\mathcal{A})}_{\neg\ell_{priv}}(\ell_f)$, (see Proposition 1), Definition 4 gives an immediate procedure to decide timed opacity. Note that, from the finiteness of the region graph, $\mathit{DReach}^{v(\mathcal{A})}_{\ell_{priv}}(\ell_f)$ and $\mathit{DReach}^{v(\mathcal{A})}_{\neg\ell_{priv}}(\ell_f)$ come in the form of a finite union of intervals, and their equality can be effectively computed.

Example 6. Consider again the PTA $\mathcal{A}$ in Fig. 1, and let v be such that $v(p_1) = 1$ and $v(p_2) = 2$. Recall from Example 4 that $\mathit{DReach}^{v(\mathcal{A})}_{\ell_2}(\ell_1) = [1, 3]$ and $\mathit{DReach}^{v(\mathcal{A})}_{\neg\ell_2}(\ell_1) = [2, 3]$. Thus, $\mathit{DReach}^{v(\mathcal{A})}_{\ell_2}(\ell_1) \neq \mathit{DReach}^{v(\mathcal{A})}_{\neg\ell_2}(\ell_1)$ and therefore $v(\mathcal{A})$ is *not* opaque w.r.t. ℓ_2 on the way to ℓ_1.

Now, consider v' such that $v'(p_1) = v'(p_2) = 1.5$. This time, $\mathit{DReach}^{v'(\mathcal{A})}_{\ell_2}(\ell_1) = \mathit{DReach}^{v'(\mathcal{A})}_{\neg\ell_2}(\ell_1) = [1.5, 3]$ and therefore $v'(\mathcal{A})$ is opaque w.r.t. ℓ_2 on the way to ℓ_1.

6 Decidability and Undecidability

We address here the following decision problem, that asks about the emptiness of the parameter valuations and execution times set guaranteeing timed opacity.

Timed-opacity Emptiness Problem:
INPUT: A PTA $\mathcal{A}$, a private location ℓ_{priv}, a target location ℓ_f
PROBLEM: Is the set of valuations v such that $v(\mathcal{A})$ is opaque w.r.t. ℓ_{priv} on the way to ℓ_f for a non-empty set of execution times empty?

Dually, we are interested in deciding whether there exists at least one parameter valuation for which $v(\mathcal{A})$ is opaque for at least some execution time.

With the rule of thumb that all non-trivial decision problems are undecidable for general PTAs [4], the following result is not surprising, and follows from the undecidability of reachability-emptiness for PTAs (Fig. 3).

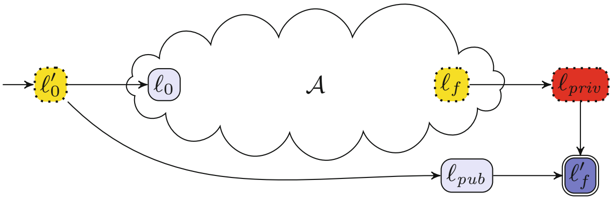

Fig. 3. Reduction from reachability-emptiness

Proposition 2 (undecidability). *The timed-opacity emptiness problem is undecidable for general PTAs.*

We now show that the timed-opacity emptiness problem is decidable for the subclass of PTAs called L/U-PTAs [14]. Despite early positive results for L/U-PTAs, more recent results mostly proved undecidable properties of L/U-PTAs [4], and therefore this positive result is welcome.

Definition 5 (L/U-PTA). *An* L/U-PTA *is a PTA where the set of parameters is partitioned into lower-bound parameters and upper-bound parameters, where each upper-bound (resp. lower-bound) parameter p_i must be such that, for every guard or invariant constraint $x \bowtie \sum_{1 \leq i \leq M} \alpha_i p_i + d$, we have: $\alpha_i > 0$ implies $\bowtie \in \{\leq, <\}$ (resp. $\bowtie \in \{\geq, >\}$).*

Example 7. The PTA in Fig. 1 is an L/U-PTA with $\{p_1, p_2\}$ as lower-bound parameters, and $\emptyset$ as upper-bound parameters.

The PTA in Fig. 2 is not an L/U-PTA, because p is compared to cl both as a lower-bound (in "$\mathrm{p} \times 32^2 \leq \mathrm{cl}$") and as an upper-bound ("$\mathrm{cl} \leq \mathrm{p} \times 32^2 + \epsilon$").

Theorem 1 (decidability). *The timed-opacity emptiness problem is decidable for L/U-PTAs.*

Remark 2. The class of L/U-PTAs is known to be relatively meaningful, and many case studies from the literature fit into this class, including case studies proposed even before this class was defined in [14]. Even though the PTA in Fig. 2 does not fit in this class, it can easily be transformed into an L/U-PTA, by duplicating p into p^{l} (used in lower-bound comparisons with clocks) and p^{u} (used in upper-bound comparisons with clocks).

7 Parameter Synthesis for Opacity

Despite the negative theoretical result of Proposition 2, we now address the timed-opacity synthesis problem for the full class of PTAs. Our method may not terminate (due to the undecidability) but, if it does, its result is correct. Our workflow can be summarized as follows.

1. We enrich the original PTA by adding a Boolean flag b and a final synchronization action;

2. We perform *self-composition* (i. e., parallel composition with a copy of itself) of this modified PTA;
3. We perform reachability-synthesis using $\mathsf{EFsynth}$ on ℓ_f with contradictory values of b.

We detail each operation in the following. In this section, we assume a PTA $\mathcal{A}$, a given private location ℓ_{priv} and a given final location ℓ_f.

Enriching the PTA. We first add a Boolean flag b initially set to false, and then set to true on any transition leading to ℓ_{priv} (in the line of the proof of Proposition 1). Therefore, $b =$ true denotes that ℓ_{priv} has been visited. Second, we add a synchronization action finish on any transition leading to ℓ_f. Third, we add a new clock x_{abs} (never reset) together with a new parameter p_{abs}, and we guard all transitions to ℓ_f with $x_{abs} = p_{abs}$. This will allow to measure the (parametric) execution time. Let $\mathsf{Enrich}(\mathcal{A}, \ell_{priv}, \ell_f)$ denote this procedure.

Self-composition. We use here the principle of *self-composition*, i. e., composing the PTA with a copy of itself. More precisely, given a PTA $\mathcal{A}' = \mathsf{Enrich}(\mathcal{A}, \ell_{priv}, \ell_f)$, we first perform an identical copy of $\mathcal{A}'$ *with distinct variables*: that is, a clock x of $\mathcal{A}'$ is distinct from a clock x in the copy of $\mathcal{A}'$—which can be trivially performed using variable renaming.[1] Let $\mathsf{Copy}(\mathcal{A}')$ denote this copy of $\mathcal{A}'$. We then compute $\mathcal{A}' \parallel_{\{\text{finish}\}} \mathsf{Copy}(\mathcal{A}')$. That is, $\mathcal{A}'$ and $\mathsf{Copy}(\mathcal{A}')$ evolve completely independently due to the interleaving—except that they are forced to enter ℓ_f at the same time, thanks to the synchronization action finish.

Synthesis. Then, we apply reachability synthesis $\mathsf{EFsynth}$ (over all parameters, i. e., the "internal" timing parameters, but also the p_{abs} parameter) to the following goal location: the original $\mathcal{A}'$ is in ℓ_f with $b =$ true while its copy $\mathsf{Copy}(\mathcal{A}')$ is in ℓ'_f with $b' =$ false (primed variables denote variables from the copy). Intuitively, we synthesize timing parameters and execution times such that there exists a run reaching ℓ_f with $b =$ true (i. e., that has visited ℓ_{priv}) and there exists another run of same duration reaching ℓ_f with $b =$ false (i. e., that has not visited ℓ_{priv}).

Let $\mathsf{SynthOp}(\mathcal{A}, \ell_{priv}, \ell_f)$ denote the entire procedure. We formalize $\mathsf{SynthOp}$ in Algorithm 1, where "$\ell_f \wedge b =$ true" denotes the location ℓ_f with $b =$ true. Also note that $\mathsf{EFsynth}$ is called on a set made of a single location of $\mathcal{A}' \parallel_{\{\text{finish}\}} \mathsf{Copy}(\mathcal{A}')$; by definition of the synchronous product, this location is a *pair* of locations, one from $\mathcal{A}'$ (i. e., "$\ell_f \wedge b =$ true") and one from $\mathsf{Copy}(\mathcal{A}')$ (i. e., "$\ell'_f \wedge b' =$ false").

Example 8. Consider again the PTA $\mathcal{A}$ in Fig. 2. Fix $v(\epsilon) = 1$, $v(\text{p}) = 2$. We then perform the synthesis applied to the self-composition of $\mathcal{A}'$ according to

[1] In fact, the fresh clock x_{abs} and parameter p_{abs} can be shared to save two variables, as x_{abs} is never reset, and both PTAs enter ℓ_f at the same time, therefore both "copies" of x_{abs} and p_{abs} always share the same values.

Algorithm 1. $\mathsf{SynthOp}(\mathcal{A}, \ell_{priv}, \ell_f)$

```
input  : A PTA $\mathcal{A}$, locations $\ell_{priv}, \ell_f$
output : Constraint $K$ over the parameters
$\mathcal{A}' \leftarrow \mathsf{Enrich}(\mathcal{A}, \ell_{priv}, \ell_f)$
$\mathcal{A}'' \leftarrow \mathcal{A}' \parallel_{\{\text{finish}\}} \mathsf{Copy}(\mathcal{A}')$
return $\mathsf{EFsynth}\Big(\mathcal{A}'', \big\{(\ell_f \wedge b = \text{true}, \ell'_f \wedge b' = \text{false})\big\}\Big)$
```

Algorithm 1. The result obtained with IMITATOR is: $p_{abs} = \emptyset$ (as expected from Example 5).

Now fix $v(\epsilon) = 2$, $v(\mathrm{p}) = 1.002$. We obtain: $p_{abs} \in [1026.048, 1034]$ (again, as expected from Example 5).

Now let us keep all parameters unconstrained. The result of Algorithm 1 is the following 3-dimensional constraint: $5 \times \epsilon + 1024 \geq \mathrm{p}_{abs} \geq 1024 \wedge 1024 \times \mathrm{p} + 5 \times \epsilon \geq \mathrm{p}_{abs} \geq 1024 \times \mathrm{p} \geq 0$.

Soundness. We will state below that, whenever $\mathsf{SynthOp}(\mathcal{A}, \ell_{priv}, \ell_f)$ terminates, then its result is an exact (sound and complete) answer to the timed-opacity synthesis problem.

Theorem 2 (correctness). *Assume $\mathsf{SynthOp}(\mathcal{A}, \ell_{priv}, \ell_f)$ terminates with result K. Assume v. The following two statements are equivalent:*

1. *There exists a run of duration $v(p_{abs})$ such that ℓ_{priv} is reachable on the way to ℓ_f in $v(\mathcal{A})$ and there exists a run of duration $v(p_{abs})$ such that ℓ_{priv} is unreachable on the way to ℓ_f in $v(\mathcal{A})$.*
2. *$v \models K$.*

8 Experiments

We use IMITATOR [6], a tool taking as input networks of PTAs extended with several handful features such as shared global discrete variables, PTA synchronization through strong broadcast, etc. We ran experiments using IMITATOR 2.10.4 "Butter Jellyfish" on a Dell XPS 13 9360 equipped with an Intel® Core$^{\text{TM}}$ i7-7500U CPU @ 2.70 GHz with 8 GiB memory running Linux Mint 18.3 64 bits.[2]

8.1 Translating Programs into PTAs

We will consider case studies from the PTA community and from previous works focusing on privacy using (parametric) timed automata. In addition, we will be interested in analyzing programs too. In order to apply our method to the analysis of programs, we need a systematic way of translating a program (e. g.,

[2] Sources, models and results are available at doi.org/10.5281/zenodo.3251141.

a Java program) into a PTA. In general, precisely modeling the execution time of a program using models like timed automata is highly non-trivial due to complication of hardware pipelining, caching, OS scheduling, etc. The readers are referred to the rich literature in, for instance, [17]. In this work, we instead make the following simplistic assumption on execution time of a program statement and focus on solving the parameter synthesis problem. How to precisely model the execution time of programs is orthogonal and complementary to our work.

We assume that the execution time of a program statement other than `Thread.sleep(n)` is within a range $[0, \epsilon]$ where ϵ is a small integer constant (in milliseconds), whereas the execution time of statement `Thread.sleep(n)` is within a range $[n, n + \epsilon]$. In fact, we choose to keep ϵ *parametric* to be as general as possible, and to not depend on particular architectures.

Our test subject is a set of benchmark programs from the DARPA Space/-Time Analysis for Cybersecurity (STAC) program.[3] These programs are being released publicly to facilitate researchers to develop methods and tools for identifying STAC vulnerabilities in the programs.

8.2 A Richer Framework

The symbolic representation of variables and parameters in IMITATOR allows us to reason *symbolically* concerning variables. That is, instead of enumerating all possible (bounded) values of x and secret in Fig. 2, we turn them to parameters (i. e., unknown constants), and IMITATOR performs a symbolic reasoning. Even better, the analysis terminates for this example even when no bound is provided on these variables. This is often not possible in (non-parametric) timed automata based model checkers, that usually have to enumerate these values. Therefore, in our PTA representation of Java programs, we turn all user-input variable and secret constant variables to parameters. Other local variables are implemented using IMITATOR discrete (shared, global) variables.

8.3 Experiments

Benchmarks. As a proof of concept, we applied our method to a set of examples from the literature. The first five models come from previous works from the literature [8,13,19], also addressing non-interference or opacity in timed automata. In addition, we used two common models from the (P)TA literature, not necessarily linked to security: a toy coffee machine (`Coffee`) used as benchmark in a number of papers, and a model Fischer's mutual exclusion protocol (`Fischer-HRSV02`) [14]. In both cases, we added manually a definition of private location (the number of sugars ordered, and the identity of the process entering the critical section, respectively), and we verified whether they are opaque w.r.t. these internal behaviors.

We also applied our approach to a set of Java programs from the aforementioned STAC library. We use identifiers of the form `STAC:1:n` where `1` denotes

[3] https://github.com/Apogee-Research/STAC/.

Table 1. Experiments: timed opacity

Model			Transf. PTA			Result	
Name	$\lvert\mathcal{A}\rvert$	$\lvert\mathbb{X}\rvert$	$\lvert\mathcal{A}\rvert$	$\lvert\mathbb{X}\rvert$	$\lvert\mathbb{P}\rvert$	Time (s)	Vulnerable?
[19, Fig. 5]	1	1	2	3	3	0.02	(√)
[13, Fig. 1b]	1	1	2	3	1	0.04	(√)
[13, Fig. 2a]	1	1	2	3	1	0.05	(√)
[13, Fig. 2b]	1	1	2	3	1	0.02	(√)
Web privacy problem [8]	1	2	2	4	1	0.07	(√)
`Coffee`	1	2	2	5	1	0.05	×
`Fischer-HSRV02`	3	2	6	5	1	5.83	(√)
`STAC:1:n`			2	3	6	0.12	(√)
`STAC:1:v`			2	3	6	0.11	√
`STAC:3:n`			2	3	8	0.72	×
`STAC:3:v`			2	3	8	0.74	(√)
`STAC:4:n`			2	3	8	6.40	√
`STAC:4:v`			2	3	8	265.52	√
`STAC:5:n`			2	3	6	0.24	×
`STAC:11A:v`			2	3	8	47.77	(√)
`STAC:11B:v`			2	3	8	59.35	(√)
`STAC:12c:v`			2	3	8	18.44	√
`STAC:12e:n`			2	3	8	0.58	√
`STAC:12e:v`			2	3	8	1.10	(√)
`STAC:14:n`			2	3	8	22.34	(√)

the identifier in the library, while `n` (resp. `v`) denotes non-vulnerable (resp. vulnerable). We manually translated these programs to parametric timed automata, following the method described in Sect. 8.1. We used a representative set of programs from the library; however, some of them were too complex to fit in our framework, notably when the timing leaks come from calls to external libraries (`STAC:15:v`), when dealing with complex computations such as operations on matrices (`STAC:16:v`) or when handling probabilities (`STAC:18:v`). Proposing efficient and accurate ways to represent arbitrary programs into (parametric) timed automata is orthogonal to our work, and is the object of future works.

Timed-Opacity Computation. First, we *verified* whether a given TA model is opaque, i. e., if for all execution times reaching a given final location, both an execution passes by a given private location and an execution does not pass by this private location. To this end, we also answer the timed-opacity computation problem, i. e., to synthesize all execution times for which the system is opaque. While this problem can be verified on the region graph (Proposition 1), we use the same framework as in Sect. 7, but without parameters in the original TA. That is, we use the Boolean flag b and the parameter p_{abs} to compute all possible execution times. In other words, we use a parametric analysis to solve a non-parametric problem.

We tabulate the experiments results in Table 1. We give from left to right the model name, the numbers of automata and of clocks in the original timed automaton (this information is not relevant for Java programs as the original model is not a TA), the numbers of automata, of clocks and of parameters in the transformed PTA, the computation time in seconds (for the timed-opacity computation problem), and the result. In the result column, "×" (resp. "√")

denotes that the model is opaque (resp. is not opaque), while "($\surd$)" denotes that the model is not opaque, but could be fixed. That is, although $\mathit{DReach}^{v(\mathcal{A})}_{\ell_{priv}}(\ell_f) \neq \mathit{DReach}^{v(\mathcal{A})}_{\neg\ell_{priv}}(\ell_f)$, their intersection is non-empty and therefore, by tuning the computation time, it may be possible to make the system opaque. This will be discussed in Sect. 8.4.

Even though we are interested here in timed opacity computation (and not in synthesis), note that all models derived from Java programs feature the parameter ϵ. The result is obtained by variable elimination, i. e., by existential quantification over the parameters different from p_{abs}. In addition, the number of parameters is increased by the parameters encoding the symbolic variables (such as x and secret in Fig. 2).

Discussion. Overall, our method is able to answer the timed-opacity computation problem relatively fast, exhibiting which execution times are opaque (timed-opacity computation problem), and whether *all* execution times indeed guarantee opacity (timed-opacity problem).

In many cases, while the system is not opaque, we are able to *infer* the execution times guaranteeing opacity (cells marked "($\surd$)"). This is an advantage of our method w.r.t. methods outputting only binary answers.

We observed some mismatches in the Java programs, i. e., some programs marked `n` (non-vulnerable) in the library are actually vulnerable according to our method. This mainly comes from the fact that the STAC library uses some statistical analyses on the execution times, while we use an exact method. Therefore, a very small mismatch between $\mathit{DReach}^{v(\mathcal{A})}_{\ell_{priv}}(\ell_f)$ and $\mathit{DReach}^{v(\mathcal{A})}_{\neg\ell_{priv}}(\ell_f)$ will lead our algorithm to answer "not opaque", while statistical methods may not be able to differentiate this mismatch from noise. This is notably the case of `STAC:14:n` where some action lasts either 5,010,000 or 5,000,000 time units depending on some secret, which our method detects to be different, while the library does not. For `STAC:1:n`, using our data, the difference in the execution time upper bound between an execution performing some secret action and an execution not performing it is larger than 1%, which we believe is a value which is not negligible, and therefore this case study might be considered as vulnerable. For `STAC:4:n`, we used a different definition of opacity (whether the user has input the correct password, vs. information on the real password), which explains the mismatch.

Concerning the Java programs, we decided to keep the most abstract representation, by imposing that each instruction lasts for a time in $[0, \epsilon]$, with ϵ a parameter. However, fixing an identical (parametric) time ϵ for all instructions, or fixing an arbitrary time in a constant interval $[0, \epsilon]$ (for some constant ϵ, e. g., 1), or even fixing an identical (constant) time ϵ (e. g., 1) for all instructions, significantly speeds up the analysis. These choices can be made for larger models.

Timed Opacity Synthesis. Then, we address the timed-opacity synthesis problem. In this case, we *synthesize* both the execution time and the internal

Table 2. Experiments: timed opacity synthesis

Model				Transf. PTA			Result	
Name	$\lvert\mathcal{A}\rvert$	$\lvert\mathbb{X}\rvert$	$\lvert\mathbb{P}\rvert$	$\lvert\mathcal{A}\rvert$	$\lvert\mathbb{X}\rvert$	$\lvert\mathbb{P}\rvert$	Time (s)	Constraint
[19, Fig. 5]	1	1	0	2	3	4	0.02	K
[13, Fig. 1b]	1	1	0	2	3	3	0.03	K
[13, Fig. 2]	1	1	0	2	3	3	0.05	K
Web privacy problem [8]	1	2	2	2	4	3	0.07	K
`Coffee`	1	2	3	2	5	4	0.10	$\top$
`Fischer-HSRV02`	3	2	2	6	5	3	7.53	K
`STAC:3:v`			2	2	3	9	0.93	K

values of the parameters for which one cannot deduce private information from the execution time.

We consider the same case studies as for timed-opacity computation; however, the Java programs feature no internal "parameter" and cannot be used here. Still, we artificially enriched one of them (`STAC:3:v`) as follows: in addition to the parametric value of ϵ and the execution time, we parameterized one of the `sleep` timers. The resulting constraint can help designers to refine this latter value to ensure opacity.

We tabulate the results in Table 2, where the columns are similar to Table 1. A difference is that the first $|\mathbb{P}|$ column denotes the number of parameters in the original model (without counting these added by our transformation). In addition, Table 2 does not contain a "vulnerable?" column as we *synthesize* the condition for which the model is non-vulnerable, and therefore the answer is non-binary. However, in the last column ("Constraint"), we make explicit whether no valuations ensure opacity ("$\bot$"), all of them ("$\top$"), or some of them ("K").

Discussion. An interesting outcome is that the computation time is comparable to the (non-parametric) timed-opacity computation, with an increase of up to 20 % only. In addition, for all case studies, we exhibit at least some valuations for which the system can be made opaque. Also note that our method always terminates for these models, and therefore the result exhibited is complete. Interestingly, `Coffee` is opaque for any valuation of the 3 internal parameters.

8.4 "Repairing" a Non-opaque PTA

Our method gives a result in time of a union of polyhedra over the internal timing parameters and the execution time. On the one hand, we believe tuning the internal timing parameters should be easy: for a program, an internal timing parameter can be the duration of a `sleep`, for example. On the other hand, tuning the execution time of a program may be more subtle. A solution is to enforce a minimal execution time by adding a second thread in parallel with a `Wait()` primitive to ensure a minimal execution time. Ensuring a *maximal* execution time can be achieved with an exception stopping the program after a given time; however there is a priori no guarantee that the result of the computation is correct.

9 Conclusion

We proposed an approach based on parametric timed model checking to not only decide whether the model of a timed system can be subject to timing information leakage, but also to *synthesize* internal timing parameters and execution times that render the system opaque. We implemented our approach in a framework based on IMITATOR, and performed experiments on case studies from the literature and from a library of Java programs. We now discuss future works.

Theory. We proved decidability of the timed-opacity computation problem for TAs, but we only provided an upper bound (EXPSPACE) on the complexity. It can be easily shown that this problem is at least PSPACE, but the exact complexity remains to be exhibited.

Finally, while we proved for the class of L/U-PTAs the decidability of the timed-opacity emptiness problem, i. e., the non-existence of a valuation for which the system is opaque, our result does not necessarily mean that *exact (complete) synthesis* is possible. In fact, some results for L/U-PTAs were proved to be such that the emptiness is decidable but the synthesis is intractable: that is notably the case of reachability-emptiness, which is decidable [14] while synthesis is intractable [15]. Therefore, studying the timed-opacity synthesis problem remains to be done for L/U-PTAs.

Applications. The translation of the STAC library required some non-trivial creativity: proposing automated translations of (possibly annotated) programs to timed automata dedicated to timing analysis is on our agenda.

Acknowledgements. We thank Sudipta Chattopadhyay for helpful suggestions, Jiaying Li for his help with preliminary model conversion, and a reviewer for suggesting Remark 1.

References

1. Abbasi, I.H., Lodhi, F.K., Kamboh, A.M., Hasan, O.: Formal verification of gate-level multiple side channel parameters to detect hardware Trojans. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2016. CCIS, vol. 694, pp. 75–92. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-53946-1_5
2. Alur, R., Dill, D.L.: A theory of timed automata. TCS **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
3. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: Kosaraju, S.R., Johnson, D.S., Aggarwal, A. (eds.) STOC, pp. 592–601. ACM, New York (1993). https://doi.org/10.1145/167088.167242
4. André, É.: What's decidable about parametric timed automata? STTT **21**(2), 203–219 (2019). https://doi.org/10.1007/s10009-017-0467-0
5. André, É., Chatain, T., Encrenaz, E., Fribourg, L.: An inverse method for parametric timed automata. IJFCS **20**(5), 819–836 (2009). https://doi.org/10.1142/S0129054109006905

6. André, É., Fribourg, L., Kühne, U., Soulat, R.: IMITATOR 2.5: a tool for analyzing robustness in scheduling problems. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 33–36. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_6
7. Barbuti, R., Francesco, N.D., Santone, A., Tesei, L.: A notion of non-interference for timed automata. FI **51**(1–2), 1–11 (2002)
8. Benattar, G., Cassez, F., Lime, D., Roux, O.H.: Control and synthesis of non-interferent timed systems. Int. J. Control **88**(2), 217–236 (2015). https://doi.org/10.1080/00207179.2014.944356
9. Cassez, F.: The dark side of timed opacity. In: Park, J.H., Chen, H.-H., Atiquzzaman, M., Lee, C., Kim, T., Yeo, S.-S. (eds.) ISA 2009. LNCS, vol. 5576, pp. 21–30. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02617-1_3
10. Chattopadhyay, S., Roychoudhury, A.: Scalable and precise refinement of cache timing analysis via model checking. In: RTSS, pp. 193–203 (2011). https://doi.org/10.1109/RTSS.2011.25
11. Chu, D., Jaffar, J., Maghareh, R.: Precise cache timing analysis via symbolic execution. In: RTAS, pp. 293–304 (2016). https://doi.org/10.1109/RTAS.2016.7461358
12. Doychev, G., Feld, D., Köpf, B., Mauborgne, L., Reineke, J.: Cacheaudit: a tool for the static analysis of cache side channels. In: King, S.T. (ed.) USENIX Security Symposium, pp. 431–446. USENIX Association (2013)
13. Gardey, G., Mullins, J., Roux, O.H.: Non-interference control synthesis for security timed automata. ENTCS **180**(1), 35–53 (2007). https://doi.org/10.1016/j.entcs.2005.05.046
14. Hune, T., Romijn, J., Stoelinga, M., Vaandrager, F.W.: Linear parametric model checking of timed automata. JLAP **52–53**, 183–220 (2002). https://doi.org/10.1016/S1567-8326(02)00037-1
15. Jovanović, A., Lime, D., Roux, O.H.: Integer parameter synthesis for real-time systems. TSE **41**(5), 445–461 (2015). https://doi.org/10.1109/TSE.2014.2357445
16. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68697-5_9
17. Lv, M., Yi, W., Guan, N., Yu, G.: Combining abstract interpretation with model checking for timing analysis of multicore software. In: RTSS, pp. 339–349. IEEE Computer Society (2010). https://doi.org/10.1109/RTSS.2010.30
18. Nielson, F., Nielson, H.R., Vasilikos, P.: Information flow for timed automata. In: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (eds.) Models, Algorithms, Logics and Tools. LNCS, vol. 10460, pp. 3–21. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63121-9_1
19. Vasilikos, P., Nielson, F., Nielson, H.R.: Secure information release in timed automata. In: Bauer, L., Küsters, R. (eds.) POST 2018. LNCS, vol. 10804, pp. 28–52. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_2
20. Wang, C., Schaumont, P.: Security by compilation: an automated approach to comprehensive side-channel resistance. SIGLOG News **4**(2), 76–89 (2017). https://doi.org/10.1145/3090064.3090071
21. Wu, M., Guo, S., Schaumont, P., Wang, C.: Eliminating timing side-channel leaks using program repair. In: Tip, F., Bodden, E. (eds.) ISSTA, pp. 15–26. ACM (2018). https://doi.org/10.1145/3213846.3213851
22. Zhang, J., Gao, P., Song, F., Wang, C.: SCInfer: refinement-based verification of software countermeasures against side-channel attacks. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 157–177. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_12