Automated Formal Analysis of Side-Channel Attacks on Probabilistic Systems

Chris Novakovic and David Parker

School of Computer Science, University of Birmingham, UK
{c.novakovic,d.a.parker}@cs.bham.ac.uk

Abstract. The security guarantees of even theoretically-secure systems can be undermined by the presence of side channels in their implementations. We present Sch-imp, a probabilistic imperative language for side channel analysis containing primitives for identifying secret and publicly-observable data, and in which resource consumption is modelled at the function level. We provide a semantics for Sch-imp programs in terms of discrete-time Markov chains. Building on this, we propose automated techniques to detect worst-case attack strategies for correctly deducing a program's secret information from its outputs and resource consumption, based on verification of partially-observable Markov decision processes. We implement this in a tool and show how it can be used to quantify the severity of worst-case side-channel attacks against a selection of systems, including anonymity networks, covert communication channels and modular arithmetic implementations used for public-key cryptography.

1 Introduction

Side channels are covert channels that convey information about the behaviour of a hardware or software system implementation beyond what was intended by its design. Information from a system's side channels — most commonly via their use of resources such as time or power, or their production of emissions such as electromagnetic radiation or sound — may be combined with information gained via the system's regular output channels in such a way that an observer may be able to correlate the system's overt behaviour with information they are unable to directly observe, such as data stored in a program's memory.

Side channels are most impactful in systems that attempt to ensure the confidentiality of some secret data being processed, even in systems that are theoretically secure. Software-level attacks often leverage authorised access or exposure to the system that the attacker already has, making them particularly potent: for instance, a *timing side channel* may be exploitable by an attacker with a user account on the same system, or with a virtual machine running on the same hypervisor (e.g. [19, 22]). Hardware-level attacks — such as *power analysis* — were once prohibitively expensive to mount, but thanks to the ever-increasing quality of consumer-level gadgets and falling cost of specialist hardware, even they are now within reach of attackers with modest resources; e.g., it is now possible to use $50 software-defined radios to break widely-used cryptosystems [8, 9].

Given the potential severity and relative ease of performing successful side-channel attacks, there is a need to be able to verify that implementations of theoretically-secure systems are free of such vulnerabilities — or, in cases where side channels are an unavoidable consequence of the system's intended behaviour, that they do not leak more than a maximum permitted amount of information about the secret data being processed. When an undesirable side-channel does exist, we also want to know the execution path through the system that causes the side channel to arise, so that it can be eliminated or mitigated.

This paper presents a framework for automatically analysing systems for the presence of side channels in the face of an adversary with knowledge of the system's behaviour (although not necessarily the secret information it is processing) and the capability to observe its outputs; this is analagous to a physical attacker with (e.g.) a hardware schematic or program source code and the ability to time certain operations or empirically measure their power consumption. Since probability is an important factor in the design and implementation of many security protocols and systems, we focus on the analysis of probabilistic systems.

We have developed Sch-imp, a probabilistic language featuring control flow structures (functions, conditionals and loops), scoped variable declaration and assignment, and the ability to indicate that certain values are output publicly. The language is expressive enough that non-trivial models can be encoded succinctly. The program's secret information is stored in variables defined with the keyword initial. As with regular variables in Sch-imp, the values of initial variables are assigned according to a probability mass function (p.m.f.); however, the attacker does not necessarily know which concrete value was drawn from each p.m.f. and assigned to each initial variable, and the attacker's goal is therefore to maximise what they learn about these concrete values by observing the program's externally-visible behaviour.

A novelty of this framework is the ability to reason about the resource usage of Sch-imp programs. A *resource function* is declared alongside a Sch-imp program, which defines how (a subset of) functions declared in the program make use of resources when invoked. While our focus in this paper is on how functions consume time and power, the framework is flexible enough that any other consumable resource could be considered. We assume that the attacker is capable of monitoring the program's resource usage as it executes, and may exploit it in an effort to compromise the secrecy of its initial variables.

We provide a semantics for the execution of Sch-imp programs that is parameterised by the resource function and defined in terms of a discrete-time Markov chain (DTMC). The states of the DTMC capture two constructs of relevance to side-channel analysis: the set of concrete mappings for each initial variable declared in the program, and an *observation function* encoding all of the information about the program's behaviour that is exposed to the attacker.

First, we systematically explore and construct this DTMC representing the (probabilistic) behaviour of the system. We then use this to construct a partially-observable Markov decision process (POMDP) in which the initial variable information from each terminating state is hidden. The partial observability property

of a POMDP is ideal for modelling the uncertain knowledge of the Sch-imp program's internal state (specifically, the concrete value of each initial variable for a particular execution trace) from the attacker's perspective. We then solve the POMDP to identify the attacker's optimal strategies for learning the hidden initial variable information by observing the outputs and resource usage. In doing so, we compute the (worst-case) probability of such an attack succeeding, thus meaningfully quantifying the worst-case exposure of the program's secrets.

Our approach is fully automated and we have implemented it in a tool [1]. An analyst need only encode their system in Sch-imp, along with the resource usage of its functions (which could be empirically measured). The tool then explores the DTMC representing the system's state space and constructs and solves the POMDP modelling the attacker's uncertain knowledge of this state space using an extension [17] of the Prism [13] model checker. The two phase construction of the POMDP (via a DTMC) provides opportunities to aggressively minimise the state space of the models. This is an important consideration for any technique based on exhaustive state space exploration. We illustrate the practicality and applicability of our techniques and tool by applying them to a selection of case studies: an anonymity network, a covert communication channel, and a modular arithmetic implementation used for public-key cryptography.

1.1 Related Work

The leakage of information from a secret channel to a public channel in insecure systems is a well-known problem, and has been studied extensively. Many existing approaches use concepts from information theory to quantify the leakage; common measures include Shannon entropy, min-entropy, and mutual information. (Smith [20] performs a brief survey.) There is no single measure that is appropriate for use in all scenarios [2], and it is often difficult to interpret their concrete effect on the system's security. In contrast, our framework provides an easily-understood metric: the probability that the attacker's best possible strategy successfully manages to compromise the system's secret information.

We consider the effect of side channels on probabilistic systems in which the secret information is present at initialisation and outputs (including the use of resources) occur as the system executes and eventually terminates. Information flow and side-channel analysis frameworks for several other types of system exist, including non-terminating [3, 23] and interactive [12] systems. Although our framework does not currently consider the case where the attacker is able to interact with and observe the system simultaneously, it is intended to be extendable to this case by modelling the entire execution of the system as a POMDP and the attacker's inputs as nondeterministic choices.

There are many examples of probabilistic languages in the literature, e.g. in artificial intelligence, where reasoning under uncertainty in probabilistic environments is common. These languages are inappropriate for use in our work, as either they are too low-level to succintly encode the systems (and their resource usage) described in Section 4 (e.g. [7]), or because uncertainty of and belief about the program's state are an inherent aspect of the language (e.g. [18]); our work

infers the attacker's uncertainty as the POMDP is constructed, and does not require that complexity to be part of the language encoding the system itself. Sch-imp is most closely related to Ch-imp, a probabilistic language for information flow analysis that features in our earlier work [6]. As in Sch-imp, the execution of Ch-imp programs is defined as a semantics that induces a DTMC; however, Ch-imp has no notion of subroutines or functions that define their resource usage, which are needed for side-channel analysis.

While POMDPs are widely used in other areas of research, their application to quantitative information flow analysis is less well-studied. Marecki et al. [15] analyse unauthorised information leaks in one-to-many broadcast systems, using POMDPs to model the sender's uncertainty about the recipient's subsequent handling of the secret information; Tschantz et al. [21] have a similar concern. The covert channel example that we use as an example in Section 4 was analysed as a POMDP in [17], but that does not explicitly consider side channels or attack strategies. To the best of our knowledge, our framework is the first to use POMDPs for the formal analysis of side-channel attacks.

2 A Language for Formal Side-Channel Analysis

We now present Sch-imp, the probabilistic language used by our framework. In this section, we give the syntax of the language, explain how resource usage is modelled in Sch-imp programs, and give a formal defintion of the semantics.

2.1 The Sch-imp Language

The grammar for Sch-imp is shown in Fig. 1 and we give an illustrative example program in Fig. 2 (a larger example for one of our case studies can also be found in Appendix A). Values of variables are rational numbers, assigned according to a p.m.f. over $\mathbb{Q}$. There are two types of variables: *initial variables* (declared with the initial command at the start of the program, whose initial values are considered "secret" and therefore of interest to an attacker), and regular variables (declared with the new command, and which have no secrecy connotations). Initial variables, and regular variables declared immediately afterward, are visible to all functions, while variables declared inside function bodies and if and while

$$
\begin{aligned}
\mathbb{P} \ ::= \ & [\mathsf{initial}\ V := \rho;]^* \\
& [\mathsf{new}\ V := \rho;]^* \\
& [\mathsf{function}\ F([V]^*)\ \{\ C;\ [\mathsf{output}\ [A]^+;]^?\ \mathsf{return}\ \};]^+ \\
& F([A]^*);\ \mathsf{end} \\
C \ ::= \ & \mathsf{skip} \ \mid \ \mathsf{new}\ V := \rho \ \mid \ V := \rho \ \mid \ F([A]^*) \\
\mid \ & \mathsf{if}\ (B)\ \{\ C\ \}\ [\mathsf{else}\ \{\ C\ \}]^? \\
\mid \ & \mathsf{while}\ (B)\ \{\ C\ \} \ \mid \ C;\ C
\end{aligned}
$$

Fig. 1. The Sch-imp grammar. V is a variable name, A is an arithmetic expression, B is a Boolean expression, and ρ is a p.m.f. over arithmetic expressions.

```
initial i := {
    0 → 1/4, 1 → 1/4, 2 → 1/4, 3 → 1/4
};
function f(x) {
    new o := 1;
    if (x > 0) { o := x / x };
    output o;
    return
};
f(i);
end
```

```
{
  f → {
    (0) → {
      (5, 7) → 1/2, (6, 7) → 1/2
    },
    (_) → {
      (6, 7) → 1/2, (7, 7) → 1/2
    }
  }
}
```

Fig. 2. A Sch-imp program and resource function containing a side channel when $\mathsf{i} = 0$. _ represents any arithmetic constant permitted by Sch-imp (i.e., a rational number).

blocks are in scope only within those constructs. We consider programs that declare a variable with the same name twice in the same scope or that refer to undefined variables to be badly-formed.

Following the declaration of top-level variables, a program consists of at least one function definition followed by the invocation of one of these functions. Function bodies may invoke other functions, subject to the limitations described in Section 2.2. Before a function returns control to its caller, it may output the result of evaluating one or more arithmetic expressions with the output command; these values are considered "public" and visible to the attacker.

2.2 Resource Usage in Sch-imp Programs

While the overt behaviour of Sch-imp programs is expressed by the syntax in Fig. 1, we are primarily interested in the covert information about the program's behaviour that is revealed during its execution. In reality, this covert information is most often revealed through a system's use of available resources, typically time and power. Since functions represent the broadest level of control flow within Sch-imp programs, and because the behaviour of a function typically varies depending on the arguments passed to it, it is natural to reason about the resource usage of a program's functions based on how they are called. We therefore employ a *resource function* that defines how functions in the Sch-imp program consume time and power based on the arguments passed to them.

Definition 1 (resource function). *A* resource function $\mathcal{R}$ *for a* Sch-imp *program* $\mathbb{P}$ *ranges over a subset of the functions declared in* $\mathbb{P}$ *and, for each such function* F*, partially maps sequences of arguments* $(q_1, \ldots, q_n)$ *to probability distributions over tuples* $(\mathbb{N} \times \mathbb{N})$ *that define the number of units of time that elapse and of power that are consumed when* $F(q_1, \ldots, q_n)$ *is executed.*

Similarly to how a Sch-imp program can be seen as a formal encoding of a system, a resource function can be seen as a formal encoding of a system's resource usage; as such, the information in a resource function could (e.g.) be determined empirically from the resource usage of a system's implementation.

An example Sch-imp program and its resource function are given in Fig. 2. While the program theoretically does not overtly leak information about the secret value of the initial variable i — it ultimately has no effect on the value of o that is output and visible to the attacker — the resource function indicates that the function f() on average executes slightly faster when its parameter x is 0, perhaps because of the extra operation that is performed when $x > 0$. Because the value of x is directly related to that of i when it was declared, this in fact presents a timing side channel that leaks information about i to the attacker.

Although function bodies consist of one or more commands, we take a high-level view of their resource consumption: their commands consume resources as a single unit, rather than discretely. From the perspective of the attacker, a function that consumes a non-zero amount of time or power when it executes does so atomically, regardless of the size or complexity of its body. In order to provide a clean definition of resource usage, we introduce the notion of an *instantaneous* function, whose execution takes no time and consumes no power from the perspective of the attacker; this is defined formally below. Any other function is referred to as *non-instantaneous*.

Definition 2 (instantaneous function). *A* Sch-imp *function F with n parameters is* instantaneous *with respect to a resource function $\mathcal{R}$ iff $F \notin \mathrm{dom}\,(\mathcal{R})$ or $\mathcal{R}\,(F)\,(q_1, \ldots, q_n) = \{(0,0) \to 1\}$ for any argument sequence $(q_1, \ldots, q_n)$.*

Because function bodies may themselves invoke functions, it is unclear what information an attacker would learn about a program if a non-instantaneous function A were to invoke another non-instantaneous function B given the above definitions: because the commands in a non-instantaneous function body consume resources as a single unit, the resources consumed by B would also appear to be consumed during its invocation in A, at which point A would no longer necessarily consume the resources dictated by the resource function, thus creating a contradiction. To simplify matters, we consider programs in which non-instantaneous functions invoke non-instantaneous functions to be badly-formed. All other forms of invocation, including (bounded) recursive invocation of instantaneous functions, are permitted.

Information leakage model. The presence of side channels in a system can be characterised as a special case of information leakage in which the "public information" in the system consists not only of the overt outputs that the system produces on the public channel, but also information on other visible channels that can be correlated with the information from the public channel to form a new *multiplex* channel with a greater capacity. This creates a best-case scenario in which an attacker observing the multiplex channel learns nothing more about the system's secret information than they do by observing only the public channel; this indicates that the system is free from side channels. Alternatively, the worst-case scenario is the one in which the attacker learns nothing about the system's secret information by observing the public channel, but learns all of the secret information when observing the multiplex channel.

2.3 Semantics for the Sch-imp Language

The execution of a Sch-imp program is defined in terms of a discrete-time Markov chain (DTMC):

Definition 3 (discrete-time Markov chain). *A DTMC $\mathcal{D}$ is a tuple $(S, \bar{s}, \mathbf{P})$, where S is a finite set of states, $\bar{s} \in S$ is an initial state, and $\mathbf{P} : S \times S \to [0, 1]$ is a transition probability matrix such that $\sum_{s' \in S} \mathbf{P}(s, s') = 1$ for all $s \in S$.*

In the context of Sch-imp, the states in S define the execution status of the program at any given moment, providing a notion of a program counter, storage for bindings for in-scope variables, and information about the secret data, observable data and resource usage that has occurred up to that point during the program's execution. More formally:

Definition 4 (state). *A* Sch-imp state *is a tuple $(\mathcal{F}, \mathbb{I}, t, p, \Delta)$, where:*

- $\mathcal{F} : C \times \mathrm{seq}\,(\mathbf{Var} \to \mathbb{Q})$ *is a stack of commands (with their associated variable scope frames) that remain to be executed;*
- $\mathbb{I} : \mathbf{Var} \to \mathbb{Q}$ *is a mapping consisting of the initial variables defined during the program's execution along with their values;*
- $t : \mathbb{N}$ *is the total time that has elapsed so far during the program's execution;*
- $p : \mathbb{N}$ *is the cumulative amount of power that has been consumed so far during the program's execution;*
- $\Delta : \mathbb{N} \to \mathbb{N} \times \mathrm{seq}\,(\mathbb{Q})$ *is an* observation function *defining the cumulative amount of power consumed by and values that were output from the program at a given time.*

$\mathcal{F}$ behaves like a call stack: each element in $\mathcal{F}$ represents the commands to be executed during invocation of a single function, along with a sequence of bindings for variables that are visible to that function, which we denote with σ. The first element in $\mathcal{F}$ represents the function currently being executed. Within σ, the last element represents the program's global scope (i.e., it contains bindings for the top-level variables declared at the start of the program); the penultimate element contains bindings for the function's parameters based on the arguments present when the function was invoked, and the remaining elements represent block-level scope frames within the function, becoming narrower toward the start of the sequence. $\mathbb{I}$ maintains the secret values of the initial variables at the point at which they were declared, while the observation function Δ represents the attacker's knowledge of the program's execution status; they are respectively formalisations of the program's secret and multiplex channels described earlier.

The semantic rules for the Sch-imp commands relevant to side-channel analysis are shown in Fig. 3; the remaining rules are intuitive or result in deterministic transitions between states that are not relevant to side-channel analysis, and are omitted for brevity. We write $s \xrightarrow{p} s'$ to denote the existence of a DTMC transition from state s to state s' with probability p (i.e. $\mathbf{P}(s, s') = p$). Formally, therefore, the semantics of a Sch-imp program is a DTMC (per Definition 3), where

S is a finite set of Sch-imp states (per Definition 4), $\bar{s} = ((\mathbb{P}, (\{\})), \{\}, 0, 0, \{\})$, and $\mathbf{P}$ is defined by the rules in Fig. 3 (amongst others).

There are two sources of probabilistic behaviour in Sch-imp. The first is the initial, new and assignment commands, which bind a value to a variable according to a p.m.f. ρ. Variable scope is maintained as functions and command blocks (i.e., branches of if commands and bodies of while loops) execute via the creation and destruction of scope frames. We note that the value of a variable declared with the initial command is only considered secret *at the point at which it is declared*; thus, secrecy is a property of the *specific value* of an initial variable, rather than of the variable itself.

The second source of probabilistic behaviour is the resource function $\mathcal{R}$: when a function is invoked, the cumulative elapsed time and power consumption of the program are incremented probabilistically according to the p.m.f. $\mathcal{R}(F, (q_1, \ldots, q_n))$ (where $(q_1, \ldots, q_n)$ are the arguments passed to F after evaluation) and are stored in s'. The new time and power information is also stored in the observation function Δ, indicating that the attacker is able to observe how the program is consuming resources as it executes.

The output command indicates that the given sequence of values (following evaluation of the expressions) is revealed on the program's public channel. This sequence is associated with the current amount of elapsed time in the observation function; if values have already been output by the program at this time point, the new outputs are appended to the existing sequence. This means that the invocation of multiple instantaneous functions, all producing outputs, will appear to the attacker as an instantaneous stream of outputs on the public channel.

In this work, we assume that Sch-imp programs always eventually terminate (with probability 1) and that their semantics yields a finite state space. We define *terminating* states as those in which an end command is executed, and denote the set of all such states $\underline{S}$.

$$\frac{[\![A]\!]o :: \sigma \to q}{((\mathsf{initial}\ V := \rho;\ C, o :: \sigma) :: \mathcal{F}, \mathbb{I}, t, p, \Delta) \xrightarrow{\rho(A)} ((C, o \cup \{V \to q\} :: \sigma) :: \mathcal{F}, \mathbb{I} \cup \{V \to q\}, t, p, \Delta)}$$

$$\frac{[\![A_i]\!]\sigma \to q_i}{((\mathsf{output}\ V_1, \ldots, V_n;\ C, \sigma) :: \mathcal{F}, \mathbb{I}, t, p, \Delta) \xrightarrow{1} ((C, \sigma) :: \mathcal{F}, \mathbb{I}, t, p, \Delta \cup \{t \to (\Delta_p(t), \Delta_o(t) :: (q_1, \ldots, q_n))\})}$$

$$\frac{[\![A_i]\!]\sigma \to q_i}{((F(A_1, \ldots, A_n);\ C, \sigma) :: \mathcal{F}, \mathbb{I}, t, p, \Delta) \xrightarrow{\mathcal{R}(F, (q_1, \ldots, q_n), (t', p'))} (((\mathbf{C}(F), \{\mathbf{V}(F)_1 \to q_1, \ldots, \mathbf{V}(F)_n \to q_n\} :: \sigma_G), (C, \sigma)) :: \mathcal{F}, \mathbb{I}, t + t', p + p', \Delta \cup \{t \to (\Delta_p(t) + p', ())\})}$$

Fig. 3. The side-channel semantic rules of Sch-imp. σ_G is the global variable scope frame, $\mathbf{V}(F)$ and $\mathbf{C}(F)$ are the parameter names and body of function F respectively, $\Delta_p(t)$ and $\Delta_o(t)$ are the power consumption and the list of values output at time t.

3 Automated Detection of Side-Channel Attacks

Using the semantics defined above, we can construct a DTMC representing all possible executions of a Sch-imp program. From this, we describe how to build and analyse a partially-observable Markov decision process to detect and quantify side-channel attacks that compromise the program's secret information.

3.1 POMDPs

We model the attacker's capabilities using *partially-observable Markov decision processes* (POMDPs), which are an extension of a Markov decision processes (MDPs). POMDPs model decision-making in the context of a probabilistic system where decisions can only be made based on observable parts of the system. We summarise the key concepts below, adopting the notation of [17].

Definition 5 (POMDP). *A POMDP is a tuple* $\mathcal{P} = (S, \bar{s}, A, T, O, \mathbf{O})$*, where:* S *is a finite set of states;* $\bar{s} \in S$ *is an initial state;* A *is a set of actions;* $T : S \times A \to (S \to [0,1])$ *is a (partial) transition probability function;* O *is a finite set of observations; and* $\mathbf{O} : S \to O$ *is a labelling of states with observations.*

In each state $s \in S$ of a POMDP, there is a choice between the set of available actions $A(s) \stackrel{\text{def}}{=} \{a \in A \mid T(s,a) \text{ is defined}\}$. States with the same observation must have the same available actions, i.e., for states $s, s' \in S$ with $\mathbf{O}(s) = \mathbf{O}(s')$, we have $A(s) = A(s')$. Once an action $a \in A(s)$ is chosen in state s, the next state of the POMDP is determined by the probability distribution $T(s,a)$, i.e. it transitions to state s' with probability $T(s,a)(s')$.

A *strategy* (also known as a policy) of a POMDP $\mathcal{P}$ resolves the choice of action taken in each state, based on the history of its execution so far. Formally, it is defined as a function from any finite path of $\mathcal{P}$ to one of the actions available in the final state of the path. We are only interested in *observation-based* strategies which make decisions based purely on the observation $\mathbf{O}(s)$ for each state s of the POMDP's history. In this work, we only need *finite-memory* strategies, whose choices depend not on the full history of the POMDP, but on one of a finite set of modes, which are switched between over time. Under a given strategy σ for $\mathcal{P}$, we can define a probability measure $Pr^{\sigma}_{\mathcal{P}}$ over the set of possible paths (executions) through the POMDP [11] and use this to quantify various measures of interest. In this paper, we concern ourselves with the probability $Pr^{\sigma}_{\mathcal{P}}(\Diamond T)$ of reaching a set $T \subseteq S$ of target states. We then wish to compute the maximum probability, over all possible strategies, of reaching T, and an optimal strategy σ^* which achieves this. While this problem is known to be undecidable [14], a variety of practical techniques exist to approximate the optimal probability.

3.2 Detecting Side Channels using POMDPs

We represent the interaction of a Sch-imp program and an attacker as a POMDP. Probabilities in the POMDP are used to model the initial assignment of values

to initial variables. We use partial observation to accurately model the capabilities of an attacker, who can observe the program's multiplex channel and must make decisions about how to proceed based only on this information.

The partial observability property restricts the knowledge of the POMDP's current state s to its observations $\mathbf{O}(s) \in O$. This is useful for the purpose of modelling an attacker in Sch-imp, as it allows privileged parts of the program's status (e.g., the concrete value of each initial variable) to be hidden while exposing information available on the program's multiplex channel via O.

For a Sch-imp program $\mathbb{P}$, we will denote by $\mathcal{P}_{\mathbb{P}}$ the POMDP constructed to analyse it. The starting point for this is the DTMC representing the semantics of $\mathbb{P}$, which we denote $\mathcal{D}_{\mathbb{P}}$. Intuitively, $\mathcal{D}_{\mathbb{P}}$ represents the execution of $\mathbb{P}$, parts of which are observable by the attacker; we then allow the attacker to guess the value of the program's initial variables based on these observations.

The DTMC has a set of *terminating* states $\underline{S}$ in which an end command is executed; we assume that these states are reached with probability 1. Each state in $\underline{S}$ contains two constructs relevant to side-channel analysis of the program: $\mathbb{I}$, which contains the original (secret) value of each of its initial variables, and Δ, which contains all of its publicly-observable information — a record of when it produced its outputs, and when it consumed power.

States of $\mathcal{P}_{\mathbb{P}}$ consist of references to the representations of $\mathbb{I}$ and Δ found in the DTMC's state, along with Boolean values indicating whether the attacker's guess for the value of each initial variable in $\mathbb{I}$ is correct. The observation function $\mathbf{O}$ is used to hide $\mathbb{I}$.

The POMDP is constructed in two phases. In the first phase, each unique representation of both $\mathbb{I}$ and Δ is extracted from $\underline{S}$ and a new state for $\mathcal{P}_{\mathbb{P}}$ is constructed from each of them, with the Boolean correctness values remaining undefined. A transition from the POMDP's initial state to each of these "phase-1" states is then added, and assigned a probability equal to the probability in $\mathcal{D}_{\mathbb{P}}$ of reaching states in $\underline{S}$ containing each particular representation of $\mathbb{I}$ and Δ. The probability for all possible such values can be determined simultaneously by computing the steady-state probability distribution of $\mathcal{D}_{\mathbb{P}}$.

In the second phase, another set of states is generated in which the representations of $\mathbb{I}$ and Δ are undefined, and each of the Boolean correctness values is set to either *true* or *false*; the number of "phase-2" states is therefore 2^n, where n is the number of initial variables declared in the Sch-imp program. The actions between the "phase-1" and "phase-2" states represent the attacker guessing a concrete value for each of the initial variables; the set of available actions between the first and second phases is therefore the Cartesian product of the sets of possible values for each initial variable. Each action results in a single deterministic transition to a "phase-2" state in which the correctness variables are assigned depending on whether each guess is correct.

Finally, we compute (or approximate) the maximum probability, in $\mathcal{P}_{\mathbb{P}}$, of reaching "phase-2" states where the guesses for all (or, if preferred, a subset) of the initial variables are correct. A corresponding POMDP strategy that achieves

these values represents the attacker strategy for optimally guessing the program's secret information based on its observations.

4 Experimental Results

We have implemented the Sch-imp language and our side channel detection techniques in a software tool. Here, we describe its implementation and demonstrate the applicability of our approach by using it to detect and quantify side channels in three case studies. The tool, as well as the Sch-imp code for these examples, is available online [1].

4.1 Implementation

Our tool is primarily implemented in Java. Parsers for the Sch-imp language and resource function definitions are developed in Antlr. Construction and analysis of DTMCs and POMDPs is achieved by building upon the Prism model checker [13], in particular the POMDP extension presented in [17].

Construction of the DTMC for a Sch-imp program is achieved by implementing the semantic rules shown in Fig. 3 as well as the ones omitted from this paper for brevity. These are used in conjunction with PRISM's *model generator* interface, used to systematically construct probabilistic models in its "explicit" model checking engine. A number of optimisations are employed here to reduce the amount of time and memory required to fully explore the DTMC's state space. The most effective optimisation drastically reduces the total number of states in the model altogether: since many commands in Sch-imp result in deterministic transitions between states, paths of deterministic transitions between more than two states are collapsed into a single deterministic transition between the states at the start and end of the path. This allows our tool to be a faithful representation of the formal model presented in this paper, while still being able to analyse systems that it otherwise could not.

The construction of the POMDP representing the attack model of a Sch-imp is achieved using a second phase of the model generator interface. Transition probabilities are computed using a steady-state analysis of the DTMC. The resulting POMDP is then (approximately) solved to determine an optimal attack strategy. This is done using the approach of [17], which is based on the construction and solution of a grid-based discretisation of the *belief space* of the POMDP. For our experiments, we fixed a grid resolution of 8 (see [17] for details), which sufficed to give accurate approximations (see Section 4.5).

4.2 Traceability in Anonymous Communication Networks

Our first case study is the *DC-net* [4] communication network protocol, which provides for the anonymous transmission of a single bit of information per round amongst its constituent nodes. Assuming the nodes are arranged in a ring, each round proceeds as follows. Each pair of adjacent nodes randomly generates a

single bit that is known only to them; this is achieved by each of the nodes randomly generating a single bit and transmitting it to the other node over a private secure channel, allowing each node to independently compute the shared bit by XORing the bit they generated with the bit they received from the other node. After this process is complete, each node has knowledge of two shared bits (one shared with each node adjacent to them). Each node then XORs these two known shared bits and publicly broadcasts the output of this operation to the other nodes, with the exception of the node that wants to transmit one extra bit anonymously during this round; this node instead broadcasts the inversion of their XOR output. When all nodes have broadcasted, each node can independently verify whether one of the nodes transmitted an extra bit in this round by XORing together all of the broadcasted bits: a result of 1 indicates the transmission of an extra bit; 0 indicates the absence of an extra bit.

While the DC-net is theoretically secure — the identity of the node transmitting the extra bit of information is concealed both to other nodes in the DC-net and to external observers — a faulty implementation may nevertheless leak information about the transmitting node's identity. Many different implementation errors could cause this situation. For example, since the node attempting to communicate anonymously must perform an additional computation compared to the other nodes, an implementation that fails to account for the additional processing time this computation incurs may cause a noticeable delay before the transmitting node broadcasts. This would therefore introduce a timing side-channel into the protocol that reveals the identity of the transmitting node.

A Sch-imp encoding of one round of a four-node DC-net is shown in Appendix A. One of the nodes is chosen uniformly to become the transmitting node in this round; its identity is stored in the initial variable transmitter, indicating that the transmitting node's identity should be concealed from the attacker. Since we assume that the model itself is known to the attacker, we are also implicitly stating that the attacker knows that each node is equally likely to be the transmitter. The broadcast() function executes the protocol from the perspective of one node (whose identity is given by the index parameter), and is invoked four times by the main() function; whether or not this node is the transmitter is given by the is_transmitter parameter. The single-bit value stored in b, which is revealed publicly at the end of the function, is computed by XORing the values of my_bit and their_bit; if this node is the transmitter, the value of b is then XORed with 1 to invert its value. The extra time taken to perform the additional computation in the case of the transmitting node is reflected in the resource function definition for broadcast(), in which broadcast()'s execution consumes a constant amount of power, but which takes differing amounts of time to execute depending on whether the node is the transmitter. The question of interest, therefore, is how much information about the identity of the transmitting node is revealed to the attacker as a result of the attacker observing the timing of the four executions of the broadcast() function, and how the attacker can improve upon their *a priori* random guess (a strategy that succeeds with probability 0.25, as each node is equally likely to be the transmitter).

We consider the scenario in which the elapsed time is drawn from an approximately binomial distribution centred on 4 units of time when the broadcasting node is the transmitter and 3 units of time otherwise, modelling a situation in which the transmitter will on average take longer to broadcast but with enough of an intersection between the two distributions that the attacker cannot be sure of their identity based solely on the timing side channel. In this scenario, Sch-imp identifies an attacker strategy that successfully deanonymises the transmitter with probability ≈ 0.527, a significant improvement over the probability of 0.25 expected of the ideal implementation.

4.3 Covert Information Flows over a Unidirectional Network

The purpose of a *unidirectional network* is to provide a means for, and enforcement of, one-way communication between hosts. An example is the National Research Laboratory's Network Pump [10], intended for use in classified networks: it divides the network into "low-security" and "high-security" partitions and, while hosts in the low-security partition may send messages to hosts in the high-security partition, it forbids information from being communicated in the opposite direction. However, the Network Pump also provides confirmation of receipt of messages, which introduces the possibility of a covert channel being created between the partitions via collusive timing delays in message receipt confirmations: if hosts in each partition can mutually agree on a scheme for encoding bits of information in the time taken between the low-security node sending its message and the high-security node confirming receipt of that message, a forbidden side channel from the high-security node to the low-security node can be created. Although a well-designed unidirectional network will introduce noise into this side channel by probabilistically inserting its own delay between receiving the confirmation from the high-security node and forwarding it on to the low-security node, the nodes will always be able to defeat the network by agreeing on a sufficiently long delay; there is therefore a trade-off to be made between limiting the capacity of the side channel (i.e., by maximising the delay) and maintaining network performance.

In the Sch-imp model of this scenario, a high-security node attempts to covertly communicate a secret value in an initial variable h (which is equally likely to be 0 or 1) to a low-security node via a unidirectional network. The acknowledgement delay introduced by the high-security node lasts for h0 units of time when h is 0 and h1 units of time when h is 1. The network introduces its own probabilistic delay of $1/2^{\mathsf{h}n}$ units of time, where n is the value of h. If the low-security node does not receive an acknowledgement after 10 units of time, it assumes the message has been lost. The nodes may exchange up to m messages in an attempt to communicate the value of h. By fixing the value of h0 and varying the values of h1 and m, we can identify the artificial delays that the high-security node can choose to insert to maximise the probability that the value of h is leaked successfully within the permitted number of messages while maintaining network performance.

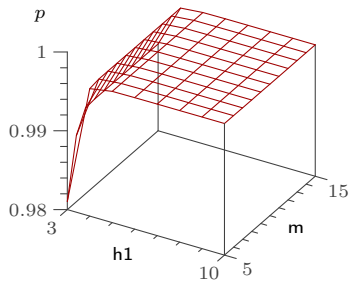

Fig. 4. The vulnerability of the unidirectional network to side-channel attacks for a fixed value of h0 (2) and varying values of h1 and m.

Fig. 4 shows the probability of h successfully being communicated for a fixed value of 2 for h0, values of h1 from 3–10, and values of m from 5–15. The colluding nodes quickly benefit from diminishing returns as h1 and m both increase: when h1 = 4, the nodes can already leak h with probability ≈ 0.997 within 4 messages, and the success rate does not improve significantly either by increasing the artificial delay or by exchanging more messages.

4.4 Power Consumption of Square-and-Multiply Algorithms

Modular exponentiation — a modular arithmetic variant of the exponentiation operation — is a fundamental operation in public-key cryptography. Operations are of the form $b^n \bmod m$. While it is cheap to compute directly for small values of n, more efficient algorithms are required when computing modular exponentiations for larger values of n, such as those used as private or public keys in public-key cryptography. *Square-and-multiply* is one such algorithm: starting with $r = 1$, for each bit n_i in n, r is squared modulo m and then multiplied by b modulo m if bit n_i is 1; the result of the modular exponentiation is the final value of r. While this algorithm is able to compute modular exponentiations with lower space and time complexities than the direct method due to the efficiency with which the squaring operation can be performed in hardware, the multiplication operation is still comparatively expensive. Crucially, because this expense only occurs for certain bits of n, naive implementations of the algorithm leak information about n. This has been the basis of power analysis side-channel attacks against cryptosystems that rely on the impracticality of inverting the modular exponentiation by computing the discrete logarithm (e.g., [16]).

In the Sch-imp modelling of this scenario, we assume a naive implementation of square-and-multiply is being used to compute a ciphertext for a public-key cryptosystem, so the exponent n is in fact a private key e whose value is secret; we select values of n from a uniform distribution over 0–7. The values of b and m are unimportant in this scenario, so we arbitrarily fix them at $b = 42$ and $m = 13$. The core of the algorithm is implemented in the sq_mult() function, which in turn calls the functions sq_mod() and mult_mod() depending on the

values of the individual bits of e. Since modular multiplication is a more expensive operation than modular squaring, the resource function assigns a greater consumption of power to mult_mod() than to sq_mod(). The function outputs the result of the exponentiation.

Even in an implementation free of side channels, the sq_mult() function necessarily leaks information, as our attacker model assumes that the attacker knows the value of m and the range of values for b (because of their knowledge of the system's behaviour) as well as the result of the exponentiation (by observing the outputs). The question is therefore how much *more* likely it is that the system leaks information about n due to the presence of side channels. In the ideal case — i.e., in which there is no time or power cost to invoking either sq_mod() or mult_mod(), and therefore no side channel to exploit — the attacker finds a strategy that successfully recovers e with probability ≈ 0.406. On the other hand, when sq_mod() draws power from an approximately binomial distribution centred on 3 units and mult_mod() from another centred on 5 units — simulating not only the additional power draw of the more complex function, but also the imprecise nature of power consumption and power analysis — the probability of finding an attack strategy that successfully recovers e increases to ≈ 0.964, almost certainly compromising the secrecy of the key.

There are various alternative modular exponentiation algorithms that mitigate this side channel, usually at the expense of efficiency. One example is *square-and-multiply-always*, in which the modular multiplication is performed for every bit of n and the result discarded if it is not needed. While Sch-imp verifies that this algorithm is free of side channels — a successful attack is found with probability ≈ 0.406, indicating that it is an ideal implementation — it is clearly wasteful. Chevallier-Mames et al. [5] propose a number of more efficient side-channel-resistant alternatives that rely on modular multiplication alongside standard arithmetic operations such as addition and bitwise XOR; they assume that these standard operations are side-channel-equivalent, assumptions that we also make for Sch-imp's model (i.e., that they consume the same resources when executing regardless of their operands). Sch-imp is able to verify that the algorithms in Fig. 2(b) and Fig. 4(b) of [5] are also equivalent to the ideal square-and-multiply implementation.

4.5 Evaluation

Table 1 summarises the performance of our tool with these case studies. The "Result" columns show the approximate probability p of the attacker's best possible strategy succeeding. The "error" values refer to the absolute difference between the lower and upper bounds returned by the approximate POMDP solution technique of [17] used in our tool. The largest error we encountered was 0.03 (in the unidirectional network examples where $\mathsf{h1} \geq 8$). Tighter bounds can be obtained if required using a finer grid resolution, at a cost of additional time and memory. The table also shows the size of both the DTMC and POMDP constructed, and the time required for the full process (we ran our experiments on a 2.1 GHz machine with the Java virtual machine allocated 8GB of memory).

Table 1. The DTMC and POMDP sizes for a selection of examples, along with the result of (and total time taken for) the analysis.

	States		Result		
Example	DTMC	POMDP	p	Error	Time (min)
DC-net	93333	20003	0.527	0.017	63
Uni. network: $\mathsf{h1} = 3$, $\mathsf{m} \leq 15$	142547	36009	0.991	0.000	13
Uni. network: $\mathsf{h1} = 10$, $\mathsf{m} \leq 5$	43461	12403	0.997	0.003	1
Square-and-multiply: naive	60749	27003	0.964	0.000	16

The framework and tool both rely on the ability to identify the Sch-imp program's terminating states and the probability of reaching each of them; this requires the exploration of the program's entire state space, which is infeasible in practice for large systems due to the excessive time and space complexities. However, our tool makes a number of aggressive optimisations to reduce the complexity of the DTMC model of a program's execution; most significantly, paths consisting of multiple deterministic transitions are collapsed into a single transition, reducing both the number of states and transitions without affecting the accuracy of the analysis. This explains the much larger analysis time for the DC-net example in Table 1 compared to the other examples, even though the number of states is similar: the DC-net program induces several orders of magnitude more deterministic transitions than the others, and while these transitions must be explored (hence the higher execution time), the tool only stores states and transitions that affect the side-channel analysis. Without such optimisations, the DC-net example would otherwise be infeasible to analyse.

5 Conclusion

We have presented a framework for formally analysing probabilistic systems for the presence of side channels; systems are specified in Sch-imp, an imperative probabilistic language, and are then systematically analysed through the construction and solution of a POMDP. This identifies possible side-channel attacks in the face of an adversary with knowledge of the system's behaviour, outputs and resource usage, and culminates in an easily-understood metric: the probability that the adversary's most effective attack successfully compromises the system's secret information, plus the strategy it employs to do so. We implemented our approach in a tool and applied it to several case studies. Future work will analyse extended attack models, for example where we also consider the most efficient way in which an attacker can observe the system.

Acknowledgements. This work was supported by the PRINCESS project (contract FA8750-16-C-0045) funded by the DARPA BRASS programme.

References

1. The Sch-imp Tool. https://www.cs.bham.ac.uk/research/projects/schimp/ (2019)
2. Alvim, M.S., Chatzikokolakis, K., McIver, A., Morgan, C., Palamidessi, C., Smith, G.: Axioms for Information Leakage. In: Proceedings of the 29th IEEE Computer Security Foundations Symposium (CSF 2016), pp. 77–92 (2016)
3. Biondi, F., Legay, A., Nielsen, B.F., Malacaria, P., Wasowski, A.: Information Leakage of Non-Terminating Processes. In: Proceedings of the 34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014), pp. 517–529 (2014)
4. Chaum, D.: The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. Journal of Cryptology **1**, 65–75 (1988)
5. Chevallier-Mames, B., Ciet, M., Joye, M.: Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. IEEE Transactions on Computers **53**(6), 760–768 (2004)
6. Chothia, T., Kawamoto, Y., Novakovic, C., Parker, D.: Probabilistic Point-to-Point Information Leakage. In: Proceedings of the IEEE 26th Computer Security Foundations Symposium (CSF 2013), pp. 193–205 (2013)
7. Dekhtyar, M.I., Dekhtyar, A., Subrahmanian, V.S.: Hybrid Probabilistic Programs: Algorithms and Complexity. In: Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI '99), pp. 160–169 (1999)
8. Genkin, D., Pachmanov, L., Pipman, I., Tromer, E.: Stealing Keys from PCs Using a Radio: Cheap Electromagnetic Attacks on Windowed Exponentiation. In: Proceedings of the 17th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2015), pp. 207–228 (2015)
9. Genkin, D., Shamir, A., Tromer, E.: RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. In: Proceedings of the 34th Annual Cryptology Conference (CRYPTO 2014), Part I, pp. 444–461 (2014)
10. Kang, M.H., Moore, A.P., Moskowitz, I.S.: Design and Assurance Strategy for the NRL Pump. IEEE Computer **31**(4), 56–64 (1998)
11. Kemeny, J., Snell, J., Knapp, A.: Denumerable Markov Chains. Springer-Verlag, 2nd edn. (1976)
12. Köpf, B., Basin, D.: An Information-Theoretic Model for Adaptive Side-Channel Attacks. In: Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS 2007), pp. 286–296 (2007)
13. Kwiatkowska, M., Norman, G., Parker, D.: Prism 4.0: Verification of Probabilistic Real-time Systems. In: Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11), pp. 585–591 (2011)
14. Madani, O., Hanks, S., Condon, A.: On the undecidability of probabilistic planning and related stochastic optimization problems. Artificial Intelligence **147**(1–2), 5–34 (2003)
15. Marecki, J., Srivatsa, M., Varakantham, P.: A Decision Theoretic Approach to Data Leakage Prevention. In: Proceedings of the 2010 IEEE Second International Conference on Social Computing (PASSAT 2010), pp. 776–784 (2010)

16. Messerges, T.S., Dabbish, E.A., Sloan, R.H.: Power Analysis Attacks of Modular Exponentiation in Smartcards. In: Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems (CHES 1999), pp. 144–157 (1999)
17. Norman, G., Parker, D., Zou, X.: Verification and control of partially observable probabilistic systems. Real-Time Systems **53**(3), 354–402 (2017)
18. Pfeffer, A.: IBAL: A Probabilistic Rational Programming Language. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001), pp. 733–740 (2001)
19. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: Proceedings of the 2009 ACM Conference on Computer and Communications Security (CCS 2009), pp. 199–212 (2009)
20. Smith, G.: On the Foundations of Quantitative Information Flow. In: Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS 2009), pp. 288–302 (2009)
21. Tschantz, M.C., Datta, A., Wing, J.M.: Purpose Restrictions on Information Use. In: Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS 2013), pp. 610–627 (2013)
22. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: Proceedings of the 23rd USENIX Security Symposium, pp. 719–732 (2014)
23. Zhang, D., Askarov, A., Myers, A.C.: Predictive mitigation of timing channels in interactive systems. In: Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011), pp. 563–574 (2011)

Appendix A SCH-IMP Model for Section 4.2

SCH-IMP program:

```
initial transmitter := { 1 → 1/4, 2 → 1/4, 3 → 1/4, 4 → 1/4 };
new nodes := 4;
new last_my_bit := { 0 → 1/2, 1 → 1/2 };
new last_their_bit := 0;
function broadcast(index, is_transmitter, their_bit) {
   new my_bit := 0;
   if (index == nodes - 1) {
      my_bit := last_my_bit
   } else {
      my_bit := { 0 → 1/2, 1 → 1/2 }
   };
   new b := my_bit xor their_bit;
   if (is_transmitter == 1) { b := b xor 1 };
   last_their_bit := my_bit;
   output b;
   return
};
function main() {
   new i := 0;
   while (i < nodes) {
      new is_transmitter := 0;
      if (i + 1 == transmitter) { is_transmitter := 1 };
      if (i == 0) {
         broadcast(i, is_transmitter, last_my_bit)
      } else {
         broadcast(i, is_transmitter, last_their_bit)
      };
      i := i + 1
   };
   return
};
main();
end
```

Resource function:

```
{
  broadcast → {
     (_, 0, _) → {(1, 1) → 1/10, (2, 1) → 1/5, (3, 1) → 2/5, (4, 1) → 1/5, (5, 1) → 1/10},
     (_, 1, _) → {(2, 1) → 1/10, (3, 1) → 1/5, (4, 1) → 2/5, (5, 1) → 1/5, (6, 1) → 1/10}
  }
}
```