Validation of Abstract Side-channel Models for Computer Architectures

Andreas Lindner[*], Hamed Nemati[†], Matthias Stockmayer[‡], Pablo Buiras[*], Roberto Guanciale[*], Swen Jacobs[†]
[*]KTH Royal Institute of Technology
{lindnera, buiras, robertog}@kth.se
[†]Helmholtz Center for Information Security (CISPA)
{hnnemati, jacobs}@cispa.saarland
[‡]Saarland University
s8mastoc@stud.uni-saarland.de

***Abstract*—Modern computer architectures include complex features that make it infeasible to analyze their effects on channels that may compromise program security. Abstract side-channel models have been proposed to approximate these flows in terms of system state observations, thus making the analysis tractable. However, using these models to verify security properties relies on the assumption that states with equivalent observations would be indistinguishable to the attacker on real hardware. In this work, we introduce a methodology and tool to validate side-channel models, testing program inputs that lead to equivalent observations in automatically-generated programs, and measuring against channels on the hardware. We partition the input state space based on the observation model and rely on an adaptive refinement of the model to guide the validation.**

***Index Terms*—side channel analysis, model validation, information flow security**

I. Introduction

Information flow analysis of systems that use complex hardware need abstractions. In fact, it is infeasible to precisely model and analyse the wealth of features of a modern processor, like caches and pipelining, and their effects on channels that can be accessed by an attacker, like execution time and power consumption. Several analyses [1], [2] use observational models [3] to handle this complexity. These models overapproximate information flow to channels in terms of system state observations performed by the attacker. In essence, states that lead to the same attacker observations should be indistinguishable by measuring all channels available to the attacker.

The amount of details hidden by these observational models makes it hard to trust their soundness. Multiple implementations of the same architecture, the possibility of executing multiple instructions simultaneously, speculative execution, and other optimizations can introduce side channels that may be overlooked by the abstract models. This has been demonstrated by the recent Spectre attack [4]. Therefore, it is essential to validate if the abstract models adequately reflect all information flows introduced by low level features.

In this work we propose a framework and a tool (Scam-V[1]) for automatically validating observational models of processors and their implementation. We iterate over test programs which consist of short but interesting sequences of instructions. We use an automatic procedure to partition the program input state space into classes, where all elements of a class generate the same attacker observations. This is done via synthesizing an observation equivalence relation for a given model. Validation of the model is then performed by testing on real hardware to ensure that two states of the same class are indeed indistinguishable. Each pair of input states is generated by using an SMT solver and searching for satisfiability of the synthesised relation.

[1]Side-Channel Abstract Model Validator

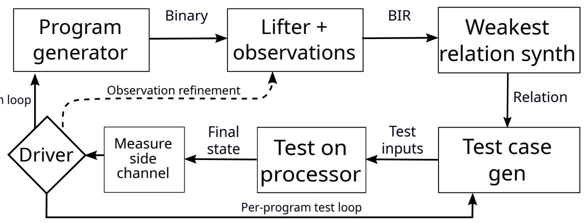

Fig. 1. Validation framework architecture

In Section II, we present the design of Scam-V. We build our framework on top of the existing binary analysis tool TrABin [5], which allows us to analyse ARMv8-A and Cortex-M0 programs by translating them to an intermediate language. Our extension annotates the programs with observations according to a model under validation, which is described in details in Section II. Section III describes the synthesis of the observation equivalence relation by executing the test program symbolically. In Section IV, we present our strategy to guide the search towards cases that are more likely to demonstrate unsoundness. We adaptively refine the observational model, searching for more state information that could affect indistinguishability of executions. Finally, Sections V and VI briefly present the concluding remarks and the related work.

II. Design

At a high level, Scam-V generates random binaries and attempts to construct initial states that lead to equivalent observations at the level of the model, but distinguishable by real

```
//   b.eq l2
[l0: CJMP Z l2 l1]
//   mul x1 x2 x3
[l1: X1= X2*X3; JMP l2]
//   ldr x2 {x1} +8
[l2: X2= LOAD(MEM, X1); X1= X1+8; HALT]
```

Fig. 2. BIR lifting example

```
[l0: CJMP Z l2 l1]
[l1: X1= X2*X3; JMP l2]
[l2: OBS(sline(X1),[tag(X1)]);
     X2= LOAD(MEM, X1); X1= X1+8; HALT]
```

Fig. 3. BIR observation example

hardware. In essence, finding such counterexamples implies that the model of the side channel is not precise enough, and leads to a potential vulnerability. Figure 1 illustrates the main workflow of Scam-V.

The first step, program generation, involves generation of binaries for a given architecture. This can be purely random or based on some heuristics, like model counting [6] or feedback directed [7], to generate more meaningful binaries.

In order to achieve a degree of hardware independence, we rely on an architecture-agnostic intermediate representation known as BIR [5], which has been introduced in previous work. BIR is an abstract assembly language that includes statements that work on memory, arithmetic expressions, and jumps. The second step of the workflow takes the specific binary from the generator and transforms it into a BIR program, a process known as *lifting*. Figure 2 shows an example of code in a generic assembly language and its lifting to BIR. In this code we perform a conditional jump to *l2* when $\mathbf{Z}$ holds, and otherwise we set $\mathbf{X1}$ to the multiplication $\mathbf{X2} * \mathbf{X3}$. Then, at *l2* we load a word from memory at address $\mathbf{X1}$ into $\mathbf{X2}$, and finally add 8 to the pointer $\mathbf{X1}$. Note that the structure of BIR requires the program to be organized into blocks, which consist of jump-free statements and end in either a conditional jump (`CJMP`), an unconditional jump (`JMP`), or `HALT`.

BIR also has explicit support for *observations*, which are produced by statements that evaluate a list of expressions in the current state. During the lifting process, we insert observation statements into the resulting BIR program to represent the observational power of the side channel. To account for expressive observation models, BIR allows conditional observation. The condition is represented by an expression attached to the observation statement. The observation itself happens only if this condition evaluates as true in the current state. Figure 3 illustrates the observations added to the previous example. We consider a scenario where the system has data caches that have been partitioned: some lines are exclusively accessible by the victim (i.e. the program), some lines can be shared with the attacker. We add the statement `OBS(sline(`$\mathbf{X1}$`),[tag(`$\mathbf{X1}$`)])` for the load instruction, which consists of an observation condition (`sline(`$\mathbf{X1}$`)`) and a list of expressions to observe (`[tag(`$\mathbf{X1}$`)]`). The function `sline` checks that its argument is in a shared line and therefore visible to the attacker. The `tag` function extracts the cache tag in which its argument is stored.

Comparing traces of these observations leads to a notion of *observational equivalence*, defined as a relation on program states. States s_1 and s_2 are observationally equivalent for program P (written as $s_1 \sim_P s_2$), if the observation traces of running P on s_1 and s_2 are equal. Note that this notion is, in principle, different from the notion of *indistinguishability* – two states are indistinguishable if a real-world attacker is not able to distinguish between them by means of the side channel in the real hardware. However, the use of an abstract model to reason about side channels relies on an assumption that observational equivalence implies indistinguishability to the real-world attacker. It is precisely this assumption – the soundness of the model – that we attempt to validate in this work.

The next step in the pipeline consists in synthesizing the weakest relation on program states that guarantees observational equivalence. This relation is used to drive the generation of *test cases* – states that can be used as inputs to the program – by means of a standard SMT solver. At this point, we run the original binary on those states using the real hardware, and then measure the side channel. In general, the expectation is that, since the chosen test cases satisfy the synthesized relation, there should be no way to distinguish between them on the real hardware either. Unless we have found a counterexample, the *driver* takes over and decides among three alternatives: generating more test cases for the same program, generating a new binary to start the process anew, or refining the observational model to reduce the search space of the analysis. The latter will be discussed in Section IV.

III. Synthesis of Weakest relation

Synthesis of the weakest relation is based on standard symbolic execution techniques. In the following we use $\mathbf{X}$ to range over symbols, and $\mathbf{c}$, $\mathbf{e}$, and $\mathbf{p}$ to range over symbolic expressions. A symbolic state σ consists of a concrete program counter i_σ, a path condition $\mathbf{p}_\sigma$, and a mapping $\mathbf{M}_\sigma$ from variables to symbolic expressions. We write $e(\sigma) = \mathbf{e}$ for the symbolic evaluation of the expression e in σ, and $\mathbf{e}(s)$ for the value obtained by substituting the symbols of the symbolic expression $\mathbf{e}$ with the values of the variables in s, where s is a concrete state.

Symbolic execution is straightforward and produces one terminating state for each possible execution path: a terminating state is produced when `HALT` is encountered, and the execution of `CJMP` $c\ l_1\ l_2$ from state σ follows both branches using the path conditions $c(\sigma)$ and $\neg c(\sigma)$. Symbolic execution of the example in Figure 3 produces two terminating states σ_1 and σ_2. For the first branch we have $\mathbf{p}_{\sigma_1} = \mathbf{Z}$ and $\mathbf{M}_{\sigma_1} = \{X_1 \to \mathbf{X}_1 + 8, X_2 \to \texttt{LOAD}(\mathbf{M}, \mathbf{X}_1)\}$ (we omit the variables that are not updated), and for the second branch $\mathbf{p}_{\sigma_2} = \neg\mathbf{Z}$ and $\mathbf{M}_{\sigma_2} = \{X_1 \to \mathbf{X}_2 * \mathbf{X}_3 + 8, X_2 \to \texttt{LOAD}(\mathbf{M}, \mathbf{X}_2 * \mathbf{X}_3)\}$.

We extend the standard symbolic execution to handle observations. That is, we add to each symbolic state a list $\mathbf{l}_\sigma$, and the execution of $\texttt{OBS}\ c\ \vec{e}$ in σ appends the pair $(\mathbf{c}, \vec{\mathbf{e}})$ to $\mathbf{l}_\sigma$, where $\mathbf{c} = c(\sigma)$ and $\vec{\mathbf{e}}[i] = \vec{e}[i](\sigma)$ are the symbolic evaluation of the condition and expressions of the observation. For instance, in the example the list for the terminating states are

$$\begin{aligned}\mathbf{l}_{\sigma_1} &= [(\texttt{sline}(\mathbf{X}_1), [\texttt{tag}(\mathbf{X}_1)])] \\ \mathbf{l}_{\sigma_2} &= [(\texttt{sline}(\mathbf{X}_2 * \mathbf{X}_3), [\texttt{tag}(\mathbf{X}_2 * \mathbf{X}_3)])]\end{aligned}$$

Let Σ be the set of terminating states produced by the symbolic execution, s be a concrete state and $\sigma \in \Sigma$ be a symbolic state such that $\mathbf{p}_\sigma(s)$ holds, and executing the program from the initial state s produces the value $\mathbf{M}_\sigma(X)(s)$ for the variable X. Moreover, let $\mathbf{l}_\sigma = [(\mathbf{c}_1, \vec{\mathbf{e}}_1) \dots (\mathbf{c}_n, \vec{\mathbf{e}}_n)]$, then the generated observations are $(\mathbf{c}_1, \vec{\mathbf{e}}_1)(s) \circ \dots \circ (\mathbf{c}_n, \vec{\mathbf{e}}_n)(s)$, where $\circ$ represents list concatenation and $(\mathbf{c}_1, \vec{\mathbf{e}}_1)(s) = [\vec{\mathbf{e}}_1(s)]$ if $\mathbf{c}_1(s)$ otherwise $[]$.

The observation equivalence relation (denoted by $\sim$) is synthesized by ensuring that every possible pair of execution paths have equivalent lists of observations. Formally $s_1 \sim s_2$ is defined as:

$$\bigwedge_{(\sigma_1,\sigma_2)\in\Sigma\times\Sigma} (\mathbf{p}_{\sigma_1}(s_1) \wedge \mathbf{p}_{\sigma_2}(s_2) \Rightarrow \mathbf{l}_{\sigma_1}(s_1) = \mathbf{l}_{\sigma_2}(s_2))$$

For the purpose of generating input test cases we can synthesize a smaller relation, since we treat the pairs (σ_1, σ_2) and (σ_2, σ_1) as the same. In the example, the synthesized relation (after simplification) is as follows (primed symbols represent variables of the second state):

$$\begin{aligned}&\mathbf{Z} \wedge \mathbf{Z}' \Rightarrow (\texttt{sline}(\mathbf{X}_1) = \texttt{sline}(\mathbf{X}'_1) \wedge \\ &\quad \texttt{sline}(\mathbf{X}_1) \Rightarrow (\texttt{tag}(\mathbf{X}_1) = \texttt{tag}(\mathbf{X}'_1))) \\ &\wedge \mathbf{Z} \wedge \neg\mathbf{Z}' \Rightarrow (\texttt{sline}(\mathbf{X}_1) = \texttt{sline}(\mathbf{X}'_2 * \mathbf{X}'_3) \wedge \\ &\quad \texttt{sline}(\mathbf{X}_1) \Rightarrow (\texttt{tag}(\mathbf{X}_1) = \texttt{tag}(\mathbf{X}'_2 * \mathbf{X}'_3))) \\ &\wedge \neg\mathbf{Z} \wedge \neg\mathbf{Z}' \Rightarrow (\texttt{sline}(\mathbf{X}_2 * \mathbf{X}_3) = \texttt{sline}(\mathbf{X}'_2 * \mathbf{X}'_3) \wedge \\ &\quad \texttt{sline}(\mathbf{X}_2 * \mathbf{X}_3) \Rightarrow (\texttt{tag}(\mathbf{X}_2 * \mathbf{X}_3) = \texttt{tag}(\mathbf{X}'_2 * \mathbf{X}'_3)))\end{aligned}$$

If the cache has 128 lines, 64 bytes per line, and only the first 10 lines are shared, then the following states satisfy the relation, because they both lead the program to access the third cache line:

$$s_1 = \left\{\begin{array}{l} Z = T \\ X_1 = 130 \\ X_2 = 123546 \\ X_3 = 87465 \end{array}\right\} \sim \left\{\begin{array}{l} Z = F \\ X_1 = 37846 \\ X_2 = 2 \\ X_3 = 64 \end{array}\right\} = s_2$$

Also the following states satisfy the relation, because they both lead the program to access cache lines that are not shared, therefore they generate no observations:

$$s'_1 = \left\{\begin{array}{l} Z = F \\ X_1 = 3246 \\ X_2 = 64 \\ X_3 = 30 \end{array}\right\} \sim \left\{\begin{array}{l} Z = F \\ X_1 = 856 \\ X_2 = 12 \\ X_3 = 64 \end{array}\right\} = s'_2$$

$$\begin{tikzcd}
 & \top & \\
 & & l2 : OBS(T, X_1) \arrow[ul] \\
M'_2 \arrow[uur, "\sqsubseteq_P"] & l2 : OBS(T, tag(X_1)) \arrow[ur] & l2 : OBS(sline(X_1), X_1) \arrow[u] \\
 & l2 : OBS(sline(X_1), tag(X_1)) \arrow[ul] \arrow[u] \arrow[ur] & \\
 & \bot \arrow[u] &
\end{tikzcd}$$

Fig. 4. Model lattice

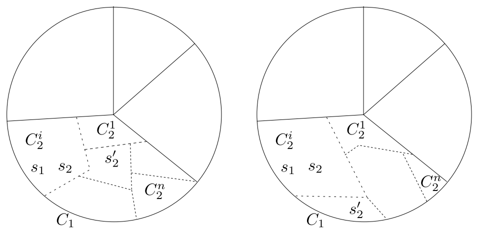

Fig. 5. Re-partitioning of the observational equivalence class using two more-restrictive models. If the model under test is sound then all states in C_1 are indistinguishable.

IV. Observation refinement

In practice, the size of an observation equivalence class can be enormous, because there are many variations to the initial states that cannot have effects on the channels available to the attacker. In the example, every state in $S_1 = \{s'_1\{v/X_1\} \mid v \in 2^{64}\}$ and $S_2 = \{s'_2\{v/X_1\} \mid v \in 2^{64}\}$ are in the same equivalence class. Testing every pair in $S_1 \times S_2$, in addition to being infeasible, is unlikely to find any distinguishable executions, because if the flag Z is not set then the register X_1 is irrelevant for the execution. The same happens for every register or memory location that does not affect the execution at all or that cannot affect the side channel (i.e. in the majority of architectures the arguments of integer additions do not affect the execution time). Finding a needle (two states that are distinguishable on real hardware) in the haystack (an observation equivalence class) seldom succeeds without proper guidance.

We use properties of observation equivalence to guide our search in meaningful directions. Intuitively, for every architecture there is an observational model that is sound: the model that observes the complete state after each instruction. For this model, which we represent by $\top$, the observation equivalence relation is the identity. Moreover, the model that produces no observations, represented by $\bot$, considers all states as indistinguishable. Given two observational models M_1 and M_2 and a program p, we say that M_1 is *less-restrictive* than M_2, and we write $M_1 \sqsubseteq_p M_2$, if $\sim_p^{M_2} \subseteq \sim_p^{M_1}$, i.e. if observational equivalence w.r.t. M_2 entails observational equivalence w.r.t. M_1. Model M_1 is *generally-less-restrictive* than M_2, written $M_1 \sqsubseteq M_2$, if $M_1 \sqsubseteq_p M_2$ for any program p. Both $\sqsubseteq$ and $\sqsubseteq_p$ form bounded lattices, whose top and bottom elements are $\top$ and $\bot$ respectively (see Figure 4).

We use this lattice to refine the model under test in the validation loop. For a program p, the initial model M_1 induces an initial partition of the input states into observation equivalence classes (see Figure 5). We use the SMT solver and the relation synthesis to find two states s_1 and s_2 that belong to the same class C_1 and we test their indistinguishability on the hardware. In case of indistinguishability, we pick up a refined model M_2 such that $M_1 \sqsubseteq_p M_2$. This allows us to repartition C_1 into M_2-observation equivalence classes $C_2^1 \dots C_2^n$. Let $s_1 \in C_2^i$, we now use the SMT solver and the relation synthesis to find a state s_2' that is in C_1 and not in C_2^i and we test its indistinguishability with s_1. Their indistinguishability validates the hypothesis that the additional observations introduced by M_2 are not needed. We then repeat this for several other models that satisfy $M_1 \sqsubseteq_p M_2'$.

The generation of the refined models requires knowledge of the architecture and is mainly syntax-driven. For example, we may add bits to the masks $\texttt{sline}(e)$ and $\texttt{tag}(e)$ for testing different cache geometries and properties, add $\texttt{OBS}(\mathit{true}, X_1)$ to $l2$ for checking variations of memory latency for different addresses, add observation of the arguments of multiplication to $l1$ for checking if multiplication is not constant-time, add observation of the program counter to every block for accounting branch effects. The same mechanism can be used in conjunction with shadow variables to analyze the effect of hidden state components. For example, we may introduce a shadow variable cache, which is a mapping from addresses to boolean and is never updated by the program, to represent if an address is in the cache before the program starts, and add the observation $\texttt{OBS}(\mathit{true}, \mathit{cache}(X_1))$ to line $l2$.

V. Ongoing validation and Future work

We are performing experiments using Scam-V on two architectures: ARMv8 and Cortex-M0. For ARMv8, we consider a model that abstracts the side channels resulting from the multi-way data-cache. The attacker can observe the tag and operation (i.e. load or store) of every memory access that targets a shareable cache line. The cache line offsets are not observable, because caches read and write complete lines from memory. We use Raspberry PI 3 for validation and we leverage TrustZone instructions for cache inspection to measure the channel. Since the CPU does not support speculative or out-of-order execution, we expect the model to be correct if the data-cache provides isolation among lines. This hypothesis could be wrong in case of prefetching (i.e. loading adjacent addresses in cache of cache misses) or global state for replacement policy.

For Cortex-M0, which is a cacheless microcontroller, we validate an abstract model of a timing channel, where the attacker can observe the program counter (and therefore the executed instruction). We use MicroBit for validation and measure execution time using the internal system clock. We expect the model to be sound, because the only instruction with variable execution time is conditional branch and its execution affects the observable program counters. This hypothesis is violated if the system memory has different latency for different addresses.

Scam-V is implemented in HOL4 and parts of its infrastructure (e.g. the transpiler) consists of proof-producing procedures. We plan to develop proof-producing versions of the symbolic execution and synthesis of weakest relation. This provides a certifying analysis of a program's *observation determinism* by demonstrating that low-equivalence implies the weakest relation.

We are also investigating other uses of the lattice of observational models. Intuitively, if a counterexample is found, the more restrictive relation can be used to search for sound models and repair the channel abstraction. On the other hand, once a model is validated, we could test less-restrictive models to find a better abstraction and, if all less-restrictive models are incorrect, validate the precision of the model.

VI. Related work

Verification of processor architecture or validating their abstract model is well studied of late [8], [9]. However, all these approaches focus on functional correctness and are not designed to identify violations of information flow properties. In contrast, we use *relational analysis* based on *observation determinism*, as introduced in [10], to validate models of non-functional properties.

Different observation models have been proposed. The program counter security model [3] has been used when the execution time depends on the control flow of the victim. Extensions of this model add to the observations the data that may affect the execution time of an instruction, like the arguments of the instruction in multi-cycle multiplication. In case of caches, the memory address accessed by the program is considered observable.

Many analysis tools use these observation models. Ct-verif [1] implements a sound information flow analysis by proving observation equivalence constructing a product program. CacheAudit [2] quantifies information leakage by using abstract interpretation.

The risks of using unsound models for these analyses have been demonstrated by the recent Spectre attack family [4], which exploits speculation to leak data through caches. Several other low-level architectural details require special caution when using these abstract models; some properties assumed by the models could be unmet, for instance cache clean operations do not always clean residual state in the implementation of replacement policies [11]. Furthermore, the most common general-purpose processors do not provide enough means to close all leakage, i.e. some shared state is not possible to clean properly on a context switch [12].

Acknowledgments

This work has been supported by the TrustFull project financed by the Swedish Foundation for Strategic Research, by the KTH CERCES Center for Resilient Critical Infrastructures financed by the Swedish Civil Contingencies Agency and by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

References

[1] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying Constant-Time Implementations," in *USENIX Security*, 2016, pp. 53–70.

[2] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "CacheAudit: A Tool for the Static Analysis of Cache Side Channels," *ACM Trans. Inf. Syst. Secur.*, vol. 18, no. 1, pp. 4:1–4:32, 2015.

[3] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks," in *ICISC*, 2006, pp. 156–168.

[4] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.

[5] A. Lindner, R. Guanciale, and R. Metere, "TrABin: Trustworthy analyses of binaries," *Science of Computer Programming*, vol. 174, pp. 72 – 89, 2019.

[6] S. Heinz and M. Sachenbacher, "Using Model Counting to Find Optimal Distinguishing Tests," in *CPAIOR*, 2009, pp. 117–131.

[7] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," in *ICSE*, 2007, pp. 75–84.

[8] J. Bhadra, M. S. Abadir, L.-C. Wang, and S. Ray, "A survey of hybrid techniques for functional verification," *Design & Test of Computers*, no. 2, pp. 112–122, 2007.

[9] B. Campbell and I. Stark, "Randomised Testing of a Microprocessor Model Using SMT-Solver State Generation," in *FMICS*, 2014, pp. 185–199.

[10] M. Balliu, M. Dam, and R. Guanciale, "Automating Information Flow Analysis of Low Level Code," in *CCS*, 2014, pp. 1080–1091.

[11] Q. Ge, Y. Yarom, and G. Heiser, "Do Hardware Cache Flushing Operations Actually Meet Our Expectations," *ArXiv e-prints*, 2016.

[12] Q. Ge, Y. Yarom, F. Li, and G. Heiser, "Your Processor Leaks Information-and There's Nothing You Can Do About It," *CoRR*, vol. abs/1612.04474, 2017.