ABSTRACT
How will future microarchitectures impact the security of existing cryptographic implementations? As we cannot keep reducing the size of transistors, chip vendors have started developing new microarchitectural optimizations to speed up computation. A recent study (Sanchez Vicarte et al., ISCA 2021) suggests that these optimizations might open the Pandora’s box of microarchitectural attacks. However, there is little guidance on how to evaluate the security impact of future optimization proposals.

To help chip vendors explore the impact of microarchitectural optimizations on cryptographic implementations, we develop (i) an expressive domain-specific language, called LmSpec, that allows them to specify the leakage model for the given optimization and (ii) a testing framework, called LmTest, to automatically detect leaks under the specified leakage model within the given implementation. Using this framework, we conduct an empirical study of 18 proposed microarchitectural optimizations on 25 implementations of eight cryptographic primitives in five popular libraries. We find that every implementation would contain secret-dependent leaks, sometimes sufficient to recover a victim’s secret key, if these optimizations were realized. Ironically, some leaks are possible only because of coding idioms used to prevent leaks under the standard constant-time model.

1 INTRODUCTION
As reducing the size of transistors is increasingly more challenging due to physical limits, chip vendors are looking into microarchitectural optimizations as an alternative to further speed up computations. These optimizations exploit the spatial and temporal locality exhibited by programs to predict future behavior and perform some of the computation in advance. Thus, the processor can cache or prefetch data in anticipation of a near-future use, or it can predict future instruction flow and even computation results in an effort to better exploit the processor’s inherent parallelism.

However, a recent study [60] suggests that this surge of new microarchitectures opens the Pandora’s box of microarchitectural attacks [23], which exploit the side effects of microarchitectural optimizations on a program’s execution time to compromise the confidentiality of an otherwise secure computation. Although the impact of existing optimizations, like caching [65, 79] or speculative execution [35], is well understood and captured by secure programming guidelines [5, 13], there is little guidance on how to evaluate the security impact of possible future microarchitectures.

As a first step in this direction, Sanchez Vicarte et al. [60] performed a systematic review of recent optimizations proposed by the computer architecture community and suggested that many of those would likely leak secret information in unexpected ways. However, the authors left open how exactly these optimizations would impact existing cryptographic implementations and whether existing mitigations, like leakage-resistant coding idioms [9, 10], would help in preventing leaks.

To substantiate this prognosis, in this paper we develop a framework for assessing the side-channel guarantees of programs against
(future) microarchitectural optimizations. Our framework consists of (1) \texttt{LmSpec}, an expressive domain-specific language that allows chip vendors and software developers to specify the leakage model associated with a given optimization in terms of leakage traces, and (2) \texttt{LmTest}, a testing approach that generates random inputs and automatically detects secret-dependent leaks for the specified \texttt{LmSpec} leakage models within a given cryptographic implementation. We use this framework to (3) perform a large scale study of the impact of the optimization proposals studied in [60] on the side-channel guarantees of mainstream cryptographic implementations. Next, we overview these contributions in more detail.

\textbf{LmSpec language:} We develop the \texttt{LmSpec} language for specifying leakage models at the ISA-level. Following the formalism of Guarnieri et al. [27], we define an \texttt{LmSpec} leakage model as (a) a \textit{leakage clause} (§3), which specifies what observations are leaked during the execution of a program, and (b) a \textit{prediction clause} (§4), which specifies the prediction mechanisms supported by the microarchitecture and what their effects are. Hence, \texttt{LmSpec} models map each program execution to a \textit{leakage trace} capturing the leaked information. Importantly, models specified in \texttt{LmSpec} are \textit{executable} for the x86 ISA (§5.1). That is, given an initial program state, we can execute an arbitrary x86 binary and derive the leakage trace generated by the \texttt{LmSpec} model. To achieve this, \texttt{LmSpec} models are automatically translated into handlers for the Unicorn CPU emulator [1] capturing the relevant information to generate leakage traces for a program execution.

\textbf{LmTest testing tool:} We develop the \texttt{LmTest} testing framework for detecting leaks in x86 programs with respect to a given \texttt{LmSpec} leakage model (§5). \texttt{LmTest} takes as input the binary of the program under test, an entry point in the program together with labels for each program input (indicating whether the parameter is public or secret), and the \texttt{LmSpec} model. To detect leaks, \texttt{LmTest} adopts a relational testing approach [50]: (1) it randomly generates test cases consisting of \textit{pairs} of initial program states that are low-equivalent, i.e., that differ only in the value of secret inputs, (2) it then executes the program on both inputs in a test case to derive the leakage traces according to the \texttt{LmSpec} model, and (3) it finally checks for differences in the two leakage traces in a test case. A test case demonstrating different leakage traces indicates a leak of secret information. We remark that differently from other random relational testing approaches [28, 50], which target \textit{fixed} leakage models, \texttt{LmTest} is parametric in the \texttt{LmSpec} leakage model, thereby allowing one to study the security implications of any optimization proposal whose leakage profile can be represented in \texttt{LmSpec}.

\textbf{Case study:} To evaluate our framework and to assess the risks that proposed optimizations pose to current software, we report on a large-scale empirical study of the impact of microarchitectural optimizations on popular cryptographic libraries (§6). We consider 18 optimizations, under six different execution models, and 25 cryptographic implementations selected from five popular cryptographic libraries, including libsodium and rust-crypto. We find that every implementation would contain secret-dependent leaks if these optimizations were realized. In some cases, an optimization-induced leak would be sufficient to recover a victim’s secret key directly from the leakage trace (e.g., for the X25519 implementation in libsodium [40]). Ironically, some leaks are possible \textit{only because} of coding idioms, such as constant-time swap or bit-masking, used to prevent leaks under the standard constant-time model.

\textbf{Summary of contributions:} In summary, the paper makes the following contributions: (1) the \texttt{LmSpec} language for rapidly prototyping leakage models (§3–4), (2) the \texttt{LmTest} testing tool for detecting secret-dependent leaks in programs against an arbitrary \texttt{LmSpec} model (§5), and (3) a large-scale case study analyzing the side-channel guarantees of mainstream cryptographic implementations against recent microarchitectural proposals (§6).

\textbf{Artifacts:} The implementation of \texttt{LmTest} and \texttt{LmSpec}, together with all leakage and prediction clauses from our case study as well as scripts to reproduce all our results are available at https://github.com/hw-sw-contracts/leakage-model-testing.

2 OVERVIEW

Here, we illustrate the key components of our approach. We start by illustrating how leakage (§2.1) and speculation (§2.2) can be modeled using the \texttt{LmSpec} language. Next, we show how \texttt{LmSpec}’s executable semantics can be used to derive leakage traces (§2.3). We then overview our notion of side-channel security (§2.4), and we conclude by illustrating how the \texttt{LmTest} testing tool can be used to detect leaks in programs given an \texttt{LmSpec} model (§2.5).

2.1 Modeling leaks with \texttt{LmSpec}

In \texttt{LmSpec}, leaks are modeled by specifying \textit{leakage clauses} (described in §3), which describe what an attacker might observe through side-channels during program execution. As an example, Figure 1 depicts the leakage clause formalizing the constant-time model in \texttt{LmSpec}. We start by defining and naming the clause with the defLeakage statement (line 1). Next, we model the three usual requirements on constant-time programs as \textit{handlers} (lines 2–7). Each handler consists of a guard, which specifies the operation being handled and the names of its arguments, and a body that computes leakage observations.

The first handler deals with memory load operations (lines 2–3); it assigns the target address to a new variable called \texttt{addr} and the target size (in bytes) to a new variable called \texttt{sz}. The body of the handler is the expression \texttt{#("load" addr)}; it produces a tuple consisting of the string “load” and the target address passed to the handler. The second handler deals with memory store operations (lines 4–5) and is defined similarly.

The last handler (lines 6–7) targets control-flow instructions (represented as \texttt{jumps} in \texttt{LmSpec}). The handler assigns the jump target address to \texttt{addr}, and the result of the jump condition to variable \texttt{n}. For unconditional jumps, \texttt{n} is simply assigned \texttt{True}. The body of the handler (line 7) returns a tuple consisting of the string “jump” and the final jump target, which we calculate by evaluating the if-then-else expression. If \texttt{n} is true, then the jump will be taken, so we leak the instruction target addr. Otherwise, we leak the address of the next instruction calculated using the special context variables \texttt{&pc} and \texttt{&insn}, where \texttt{&pc} holds the current instruction’s address and \texttt{&insn}’s size contains its size.

We describe other examples of leakage clauses in §3.
2.2 Modeling speculation with \texttt{LmSpec}

By default, \texttt{LmSpec} assumes that programs are executed following the instruction set architectural semantics. However, processors achieve significant performance improvements by speculating on the values of intermediate computations and continuing the execution based on these predictions. Speculation does not affect the correctness of the computation, because the processor always checks the correctness of its guesses and squashes all speculative execution steps in case of a misprediction. However, the microarchitectural effects of such instructions are not reversed; the resulting information leakage can be exploited to recover secrets [35].

To model the effects of new leakage models under speculation, \texttt{LmSpec} also allows modeling the effects of speculatively executed instructions. For this, \texttt{LmSpec} relies on prediction clauses (described in §4) describing (a) which instructions might trigger speculation and (b) how the predicted values (for control or data flows) are computed. As an example, Figure 2 depicts a prediction clause capturing the effects of speculating over branch instructions [35]. Prediction clauses are specified using \texttt{defpredictor} statements (line 1) and, similarly to leakage clauses, they are defined using handlers. Here, however, a handler’s body models the set of predicted values used when speculatively executing instructions. For instance, the \texttt{branchSpec} clause consists of a single handler (line 2–4) that deals with control-flow instructions \texttt{jump} with target \texttt{addr} and condition \texttt{n}. The handler uses a \texttt{when} block to test whether the current instruction is a conditional branch (line 3).

Importantly, the prediction clause \texttt{only} specifies the predicted values; it does not specify how speculative instructions are squashed or under which conditions the processor checks for misprediction. These aspects are automatically handled by \texttt{LmSpec}, which will speculatively explore all wrong predictions produced by the prediction clause following the \texttt{always mispredict} strategy [26]. In our example, the handler computes the mispredicted target in line 4 (where \texttt{"PC"} indicates a control-flow prediction). \texttt{LmSpec}, then, will use the target computed by the handler to speculatively explore, for a fixed number of steps, the mispredicted branch.

In §4, we describe in detail how prediction clauses work and model in \texttt{LmSpec} different forms of speculative execution.

2.3 Generating leakage traces

Testing for leakage with respect to a given \texttt{LmSpec} model requires deriving \texttt{leakage traces}, following the model’s leakage and prediction clauses, directly from program executions. For this, we implemented an executable version of \texttt{LmSpec} on top of the Unicorn CPU emulator [1], which allows simulating architectural executions. Given an \texttt{LmSpec} model, we automatically compile leakage and execution clauses into Unicorn \texttt{event hooks}. These hooks are used to instrument program execution whenever user-defined architectural events are triggered. For leakage clauses, \texttt{LmSpec} handlers are translated into hooks monitoring events and generating leakage observations. For instance, the \texttt{load} and \texttt{store} handlers from Figure 1 are translated into hooks intercepting memory requests and recording the address as part of the leakage trace. For prediction clauses, \texttt{LmSpec} handlers are translated into hooks that instrument program execution to explore speculative paths in addition to the architectural execution. As an example, the clause from Figure 2 is translated into a hook that starts the speculative execution of the mispredicted branch. Thus, the executable version of \texttt{LmSpec} allows us to directly derive the leakage traces associated with each program execution for arbitrary x86 binaries. We provide further details on \texttt{LmSpec}’s executable version in §5.1.

2.4 Specifying side-channel security

As standard, we formalize side-channel security as non-interference with respect to a leakage model [5]. Informally, a program is secure if its leakage traces do not leak secrets. In more detail, the notion of security is parametrized by a labeled interface for the program under consideration. The interface declares, for each program argument, its security level (secret or public) and additional constraints such as size. This interface defines a notion of valid input, and a notion of low-equivalence between two valid inputs: informally, two inputs are low-equivalent if they only differ in their secrets. Then, a program \texttt{is non-interfering} iff executing the program on pairs of low-equivalent valid inputs yield equal leakage traces.

2.5 Detecting leaks with \texttt{LmTest}

Following our formalization of side-channel security, security violations are pairs of program executions that (i) yield different leakage traces; and (ii) start from valid low-equivalent inputs, i.e., program inputs that only differ in their secrets.

To automatically discover such violations, we developed the \texttt{LmTest} testing tool, whose workflow is depicted in Figure 3. \texttt{LmTest} takes as input: (a) the binary of the \texttt{program} under test; (b) an entry point in a cryptographic library, with a labeled interface; and (c) an \texttt{LmSpec} leakage model. \texttt{LmTest} starts by generating random test cases, each one consisting of a pair of valid low-equivalent inputs. For generating a test case, \texttt{LmTest} first generates a random input satisfying the constraints of the labeled interface and then re-randomizes all secret bytes to generate a public-equivalent comparison input. Next, \texttt{LmTest} derives the leakage traces associated with each input in a test case using \texttt{LmSpec}’s executable semantics. Finally, \texttt{LmTest} compares the traces in a test case and reports all cases where traces differ, i.e., indicating secret-dependent leaks.
We developed the Leakage Model consists of a leakage clause (defined by defleakage statements, explained below) and a prediction clause (defined by defpredictor statements, explained in §4).

Each leakage clause is defined by a name, a list of state variables with their initial values, and a list of handlers. The state variables are used to capture stateful leakage clauses, as we show in our model of the computation reuse optimization in §3.4. Their initial values can be defined using arbitrary Python expressions.

Leakage handlers, instead, are used to generate sequences of leakage observations based on specific events, which we call micro-operations, happening during program execution. LmSpec considers seven different micro-operations: read, write, expr, addr, load, store, and jump. The read and write micro-operations refer to read and write operations on registers, whereas the expr micro-operation denotes computations. There is also a dedicated addr micro-operation for the computation of memory addresses. Next, load and store micro-operations refer to memory reads and writes. Finally, jump micro-operations refer to (potentially conditional) control-flow changes, i.e., changes to the program counter.

A leakage handler consists of a guard and a body. The guard of a leakage handler uop-binding is an operation with bindings for its arguments. In contrast, the body of a leakage handler (body_{\text{leak}}) is a Python expression that evaluates to a tuple of values representing leakage observations, or to None if no observation is leaked. Note that bodies can update state variables, and can freely use special context variables, such as those in Figure 5.

Next, we illustrate how LmSpec captures concisely different kinds of leaks. In §6, we describe variants of these leakage clauses as well as further clauses used in our case study.

3.2 Silent store suppression

Silent store suppression [33, 37, 60] is an optimization that suppresses a write-back to memory when the associated store operation does not change the value at that memory location. This could result in timing leaks, e.g., due to lowering the pressure on the CPU’s pipeline. Note that silent stores are considered, for instance, in the RISC-V architecture [72] and have been observed (on specific values) in some Intel CPUs [18].
The leakage clause capturing these leaks emits an observation if
the value about to be stored matches the value already in memory
at the target address. This is formalized in LmSpec as follows:

```lisp
(defvar leakage SilentStore ()
  (on [[store addr1 sz := val]]
    (when (= val (mem.read addr sz))
      #("ss" addr val)))
```

The guard of the handler is a store expression (line 2). It binds
the variable `addr` to the target address, `sz` to the value size, and `val` to
the target value to be written. The final part of the handler’s body
(line 4) returns a triple consisting of a logging string (here “ss” for
“Silent Store”), the current address, and the stored value. The first
part of the handler’s body is more interesting. It is a conditional
when block (line 3) that uses the special `&mem` context variable to
retrieve values from memory. In this case, the method `&mem.read`
is called with the target address `addr` and size `sz` to get the current
value in memory that the store operation would overwrite. The
when block compares this result to the target store value `val`, and
generates an observation if these values are equal, i.e., a necessary
condition for store suppression.

### 3.3 Register file compression

Register file compression [8, 64] is an optimization that maps multiple
logical (architectural) registers to the same physical (microarchi-
tectural) register file entry when they hold the same value. This
allows processors to perform more computations in parallel. Its
corresponding leakage model emits an observation whenever a
register is updated with a value that is already stored in another
register. This is captured by the following LmSpec leakage clause:

```lisp
(defvar leakage RegisterFileCompression ()
  (on [[write reg := val]]
    (when (and (in reg X86_64_GPRS)
                (exists reg_i X86_64_GPRS
                  :where (= reg_i reg)
                )
      #("rfc" reg val))))
```

The guard of the handler (line 2) is a write operation. It binds `reg`
with the target register name and `val` with the target value. The
body of the handler (lines 3–7) checks if `reg` is a general purpose
register—as opposed to segment registers or other special registers—
using the predicate `(in reg X86_64_GPRS)` (line 3) and scans for
another general purpose register that holds the target value (lines
4–6). For this, it uses the special context variable `&regs` to query
the logical registers by name, and the built-in `exists` function to
iterate over all general purpose register names (skipping the
register currently being written). If any other register already holds
the target value, then the handler exposes the target register and
the shared value in the leakage observation (line 7).

### 3.4 Computation reuse

Computation reuse [62] is an optimization that caches results of
recent arithmetic instructions in a hardware memoization table.
Computation reuse thus avoids re-executing any instruction whose
results are already present in the table. Its corresponding leakage
model emits an observation if a computation is performed twice.
This is captured by the following LmSpec leakage clause:

```lisp
(defvar leakage ComputationReuse
  (on [(expr (op * val))]
    (when (in op CACHING_OPS)
      #("cr" op * vs)
    )
    (update memo &pc val))))
```

We first explain the initializer. To mimic a memoization table and
to have it persist between calls to our handler, we use a `state vari-
able`—these are declared within the brackets following the model
name in a defleakage definition as a sequence of names and initial
values. In line 1, we declare a state variable called `memo`; we initial-
ize `memo` to an empty `OrderedDict`, which is a map data structure
from Python’s standard library that allows (re)ordering its keys, allowing us to simulate a simple (n-way) LRU cache of operand
values, indexed by instruction address.

Next, we turn to the guard (line 2). The form `expr` is used to
model general arithmetic instructions. It uses Hy’s list-assignment
syntax to write `(expr (op * val))`, where `op` is assigned the
instruction mnemonic corresponding to the operation and `val` is
assigned the list of operand values.

We now turn to the handler’s body (lines 3–6). It uses a predic-
ate `(in op CACHING_OPS)` to check if the current instruction is
included in a set of common arithmetic instructions that we are
explicitly caching (line 3). Then, on line 4, it retrieves the cache way
corresponding to the current instruction from `memo`, indexed by the
current instruction address (`&pc`). If there is no such mapping yet,
it returns the empty collection literal `()`. Then, if the operand list
`vals` exists as an entry within the retrieved way, the body returns
the tag `cr`, the operation and its operands as leakage observation.
Otherwise, the body calls a function `update` to update the memo
cache with the current instruction address and operands.

## 4 Modeling Speculation in LmSpec

In this section, we first show how speculation can be modeled in
LmSpec using prediction clauses (§4.1). These clauses allow users to
explore the interactions between speculative execution and leakage
clauses. We then illustrate LmSpec’s flexibility by giving examples of
control-flow (§4.2) and data-flow speculation (§4.3).

### 4.1 Prediction clauses in LmSpec

LmSpec supports prediction clauses that can be used to model spec-
ulative execution. These clauses allow users to specify the possible
predictions for every operation. For simplicity, we do not require
users to specify when the processor checks for mispeculation or
how mis Speculated instructions are squashed; these checks are
built-in into LmSpec’s semantics. In particular, LmSpec adopts the
conservative `always mispredict` approach for capturing the effects
of speculative execution [26]. Whenever the execution reaches an
instruction that can result in control flow or data prediction (as
indicated by the prediction clauses), LmSpec’s semantics (1) exe-
cutes the prediction clause to retrieve all predictions, (2) explores

---

1The choice of n is left open, as we are not modeling any known processor design. We
expect processor designers and library developers to tailor leakage models to their
own needs.
all executions associated with wrong predictions,\(^2\) before (3) finally proceeding along the path of correct execution, i.e., the architectural path. These design choices allow us to drastically reduce the user effort for specifying speculative execution in LmSpec.

The syntax of prediction clauses is given in Figure 4. Prediction clauses consist of a name, a list of state variables with their initial values, and a list of prediction handlers. Prediction handlers are very similar to leakage handlers, and support flexible and generic speculation. This contrasts with prior tools [26, 50], which rely on hard-coded rules for identifying prediction points and possible mispredictions. The core difference is that prediction handlers output a list of predicted values for control-flow or data predictions. A control-flow prediction consists of the “PC” keyword and the address of the next predicted instruction. In contrast, a data flow prediction consists of a keyword (either "MEM" indicating that we are predicting a memory value or "REG" indicating that we are predicting a register value) and a prediction, which can be a triple (address size value) for memory predictions or a pair (registerId value) for register predictions. These predictions are then used by LmSpec when exploring speculative paths, i.e., whenever LmSpec starts a new speculative path, it first selects one of the outstanding predictions and applies it to the program state before starting the speculative execution.

In the rest of this section, we show examples of how control-flow (§4.2) and data speculation (§4.3) can be implemented in LmSpec. In §6, we describe other variants of the prediction clauses that we used in our case study.

### 4.2 Control-flow speculation

We already presented a first example of control-flow speculation, i.e., misprediction of conditional branch instructions, in §2.2. Here, we present two other examples: straight-line speculation and speculation using a return stack buffer.

#### Straight-line speculation: Some AMD processors can, under certain circumstances, execute the instructions following any branch instruction [77], including unconditional branches. This is captured by the following prediction clause:

\[
\begin{align*}
&\text{defpredictor StraightLineSpec} \\
&(\text{on} \ [\text{jump } addr : n]) \rightarrow \text{("PC" (+ $pc & inssz.size))})
\end{align*}
\]

The handler always adds a prediction for the instructions following the branch (line 3). In certain cases, e.g., when a conditional branch is not taken, the following instruction is the correct next address. To prevent treating this as a misprediction, LmSpec always removes the correct outcome from the list of possible mispredictions returned by the handler.

#### Return stack buffer: To speculate over return instructions, processors employ an auxiliary data structure called the return stack buffer (RSB), which stores the return addresses of recent call instructions and uses them as prediction for the actual return address. We model RSB speculation (where the RSB is implemented using a circular buffer) with the following prediction clause:

\[
\begin{align*}
&\text{defpredictor RSBCircular} \ [\text{stack} \ (x [0] \ RSB\_SIZE) \\
&(\text{id} x) \\
&(\text{on} \ [\text{jump } addr : n]) \\
&(\text{cond}) \\
&(\text{&insn.group CS_GRP_CALL}) \\
&(\text{do} \ (\text{assoc} \ \text{stack} \ \text{id} \\
&(\text{&pc & inssz.size})) \\
&(\text{setv} \ \text{id} \ %) \\
&(\text{do} \ (\text{setv} \ \text{id} \ %) \\
&(\text{RSB\_SIZE}))})
\end{align*}
\]

The code first defines a 16-entry buffer used for the RSB (line 1). When handling jump operations, the code checks if these are calls or returns (lines 6 and 10). In the case of calls, the code pushes the return address into the stack (lines 7–9). For return instruction, instead, the code pops an entry from the stack using it as a predicted destination (lines 11–12).

### 4.3 Data speculation

So far, we have shown how control-flow speculation can be formalized in LmSpec. Here, we show an example of data speculation. In particular, we model speculation over store bypasses (exploited in Spectre-STL attacks [30]), in which the processor speculates (possibly incorrectly) that an older store does not conflict with a younger load thereby accessing stale data. This can be modeled with the following prediction clause:

\[
\begin{align*}
&\text{defpredictor StoreBypassSpec} \ [\text{buf} \ (\text{deque} \ \text{maxlen} \ \text{SIZE})] \\
&(\text{on} \ [\text{load} \ \text{sz}]) \\
&(\text{ifor} \ [\text{addr} \ \text{wsz} \ \text{val}] \ \text{buf}) \\
&(\text{if} \ (= \ [\text{addr} \ \text{sz}] \ [\text{waddr} \ \text{wsz2}]) \\
&(\text{("MEM"} \ [\text{addr} \ \text{sz} \ \text{val}]) \\
&(\text{[store} \ [\text{addr}] \text{sz} := \text{val}) \\
&(\text{.append buf} [\text{addr} \ \text{sz} \ \text{&mem.read addr sz}])))
\end{align*}
\]

The clause keeps track of recent store instructions and the contents they overwrite using the buf state variable, which is a buffer of size SIZE (defined in line 1). In particular, when executing a store micro-operation, the old overwritten value (extracted by the &mem.read addr sz expression) is appended to buf (lines 6–7). When executing a load micro-operation, the code compares its address and size with those in the store buffer, returning the old value as a potential prediction (lines 2–5).

### 5 TESTING FOR LEAKS

In this section, we introduce our approach for automatically testing the side-channel guarantees of programs against leaks captured by LmSpec models. We first describe our implementation of LmSpec’s executable semantics (§5.1) on top of the Unicorn CPU emulator [1], which enables us to derive leakage traces for arbitrary x86 program executions. We then proceed to describe our testing approach (§5.2), which we implement in the LaTEST testing tool.

#### 5.1 Generating leakage traces for LmSpec

To study the security implications associated with a given LmSpec model for real-world cryptographic implementations, we need an automated way of deriving leakage traces directly from program executions. To address this, we implemented an executable version
Testing side-channel security of cryptographic implementations against future microarchitectures

of LmSpec on top of the Unicorn emulator [1], which allows simulating x86 programs architecturally. As mentioned in §2.3, we do so by translating leakage and prediction clauses in LmSpec into event hooks in Unicorn. Event hooks allow injecting arbitrary instrumentation code that is automatically executed by the Unicorn emulator whenever specific events happen during program execution. Our implementation is inspired by the Revizor tool [50], which implements fixed leakage models on top of Unicorn using event hooks.

**Leakage clauses**: We compile each LmSpec leakage clause with \( n \) handlers into \( n \) different event hooks that monitor the execution of the current instruction and record the associated leakage observations. In particular, handlers of Load and Store operations are compiled into memory hooks, which are executed whenever the emulator executes a memory request (as a result of an instruction). Handlers for all other micro-operations, instead, are compiled into instruction hooks, which are executed whenever the emulator fetches a new instruction. The body of a leakage handler, which records the leakage observation, is a Hy expression. We directly compile it into Python (using the Hy backend [32]) as the body of the event hook.

**Prediction clauses**: By default, Unicorn emulates instructions following the x86 architectural semantics. To capture the effects of speculatively executed instructions, we extended Unicorn with an always mispredict speculation model [26, 50]. First, we compile LmSpec prediction clauses into event hooks. These hooks, however, compute a list of predictions, rather than observations (as do the hooks associated with leakage clauses). Following [50], whenever the emulator executes an instruction that triggers a hook associated with a prediction clause, it (1) takes a checkpoint of the computation state, (2) retrieves the predicted values computed by the hook (and, if present, removes the correct value from the list of predictions) and (3) continues the (speculative) simulation based on one of the predictions. When speculative execution terminates,\(^3\) the emulator restores the computation state, using the previously taken checkpoint, and either explores another speculative path (if there are other predictions) or restarts the architectural simulation.

**Context variables**: At every point during the simulation, the values of LmSpec context variables (see Figure 5) are derived by inspecting the emulator’s state. For instance, a call to &mem. read is translated into a call to the Unicorn’s API for reading memory. Similarly, the &insn structure is initialized by retrieving the current instruction from the emulator’s state, disassembling it using the Capstone library [17], and extracting the necessary information.

**Deriving traces**: Given a program, an initial program state, and an LmSpec model, the corresponding leakage trace is obtained by simulating the program execution using the Unicorn emulator extended with the event hooks obtained from the LmSpec model. In particular, the hooks associated with the prediction clause will trigger speculative execution whereas those associated with the leakage clause will track the leakage observations during execution.

---

\(^3\)Following the always mispredict model [26], this happens in one of the following cases: (1) the predefined speculation window is exhausted, (2) the execution encounters a speculation barrier, e.g., an lfence instruction, or (3) the program terminates.

### 5.2 LmTest testing tool

Here, we describe our testing approach, implemented in the LmTest tool, for detecting leaks in a program given an LmSpec model.

As anticipated in §2.5, we characterize side-channel security as a non-interference property [5]. In particular, we say that program \( P \) is secure under LmSpec model \( LM \) if for all pairs of initial program states \( s, s' \), if \( s \) and \( s' \) only differ in their secrets, then the leakage traces (according to the model \( LM \)) associated with \( P \)'s execution on states \( s \) and \( s' \) must be the same. Hence, a leak in program \( P \) under model \( LM \) consists of a violation of this non-interference property, that is, a pair of low-equivalent program states that result in different leakage traces.

#### Algorithm 1 LmTest testing approach

**Require**: Program \( P \), labeled interface \( I \), LmSpec model \( LM \), number of test cases \( N \)

1: \( s_1, \ldots, s_N \leftarrow \text{genInitConf}(I, N) \) \( \triangleright \) generate seed states
2: for \( s \in \{s_1, \ldots, s_N\} \) do
3: \( s' \leftarrow \text{mutate}(s, I) \) \( \triangleright \) mutate state
4: \( t \leftarrow \text{getTrace}(P, s, LM) \) \( \triangleright \) collect traces
5: \( t' \leftarrow \text{getTrace}(P, s', LM) \)
6: if \( t \neq t' \) then \( \triangleright \) Trace comparison
7: \( \text{return} \) Violation detected \( (s, s') \)
8: \( \text{return} \) No violation detected

To detect leaks, we use a relational random testing approach inspired by Revizor [50] and ct-fuzz [28]. LmTest approach for detecting leaks is summarized in Algorithm 1. We start by generating \( N \) randomly selected initial states following the user-provided labelled interface \( I \) (line 1). For each state \( s \), we then generate a low-equivalent state \( s' \) (line 3). Next, we derive the leakage traces associated with \( s \) and \( s' \) by simulating the program under test using the Unicorn emulator extended with the LmSpec model \( LM \) as described in §5.1 (lines 4–5). Finally, we compare the generated traces (lines 6–7): any difference in the traces is caused by a secret-dependent leak (since \( s \) and \( s' \) are low-equivalent). If we detect a difference, we report it (line 7). Otherwise, we continue the testing and move to the next test case.

Before concluding, we provide further details on the labelled interface and our test generation strategy.

**Labelled interfaces**: The labelled interface provides LmTest with a description of the program inputs. For each input, the interface describes (1) whether it is stored in a register or in memory, (2) its size in bytes, (3) whether it is public or secret, and (4) optionally) its location in memory. This interface is provided by the user and allows LmTest to generate the test cases, which, as already mentioned, are pairs of low-equivalent initial states.

**Generating states**: To generate a random initial state, we instantiate each input with a sequence of randomly generated bytes of the appropriate length, as specified in the labelled interface. This simple strategy (which might not work for structured data) is sufficient to generate valid initial states for all cryptographic implementations we tested in §6, whose inputs consist of, e.g., secret keys or messages.

**Mutating states**: To mutate a random state \( s \), we replace its secret inputs (as indicated by the interface) with sequences of fresh
randomly generated bytes of the appropriate length. This ensures that the mutated state \( s' \) is low-equivalent to the original state \( s \).

6 EVALUATION AND CASE STUDY

In this section, we study the impact of microarchitectural optimizations on the side-channel security of widely used cryptographic algorithms. As part of this case study, we identify three core research questions, which we address in the following sections:

- **RQ1** Does \( \text{LmSpec} \) provide an expressive and concise framework for specifying leakage models? (§6.1)
- **RQ2** Are real-world cryptographic libraries secure under the different models and can \( \text{LmTest} \) detect leaks in them? (§6.2)
- **RQ3** Can the leaks be exploited? (§6.3)

6.1 RQ1: Expressiveness of \( \text{LmSpec} \)

We use \( \text{LmSpec} \) to model 18 leakage clauses (§6.1.1) and six prediction clauses (§6.1.2) that we implemented in \( \text{LmSpec} \), which combined result in 108 \( \text{LmSpec} \) leakage models. Full implementations in \( \text{LmSpec} \) of all our clauses are given in Appendices A–B.

6.1.1 Leakage clauses. We implemented in \( \text{LmSpec} \) 18 different leakage clauses capturing the leaks introduced by eight different classes of microarchitectural optimizations, from the classic constant time model (capturing cache-related leaks), to complex optimizations like cache compression [66] and prefetching [68] as well as security-critical optimization proposals [60]. Implemented clauses include:

- **Constant-time (CT):** We implemented the baseline model (denoted by CT) shown in Figure 1.
- **Silent stores (SS):** We implemented the baseline model (denoted by SS) shown in §3.2, and two variants. The SS0 variant restricts observations only to silent stores where the value being written is all-zeros [18]. The SSI0 variant produces an observation only on silent stores involving memory locations that have already been initialized by the program—the latter uses initialization handlers, an \( \text{LmSpec} \) feature not presented in the paper.
- **Register file compression (RFC):** We implemented the baseline model (denoted by RFC) shown in §3.3, and two variants. The RFC0 variant only compresses registers that are all-zero [64]. The NRFC variant only compresses narrow values [19]; registers whose values are less than 16 bits are compressed into the same physical register. For the latter, our model checks whether the value being written to a register is under 16 bits and emits an observation whenever another register also stores a value that is under 16 bits.
- **Computation simplification (CS):** We implemented three models. Two models (denoted respectively by CS and CST) capture simplification strategies proposed by Atoofian and Baniasadi [6], which simplify arithmetic and logical operations on values like \( 0 \) and 1. The third, CSN, captures the effects of simplifying multiplication instructions for narrow operands (under 32 bits).
- **Operand packing (OP):** Operand packing [11] compresses multiple in-flight instructions (of the same type) with narrow operands into a single “compressed” instruction that is forwarded to the execution units. We model it by tracking the latest \( n \ \text{expr} \) micro-operations during program execution, checking whether some of these micro-operations can be compressed (i.e., they all have operands that are less than 16 bits), and producing an observation if this is the case.
- **Computation reuse (CR):** Computation reuse optimizations [62] cache recent computation results and avoid re-executing computations whenever their results are cached. We implemented two models capturing the effects of reuse optimizations from [62]. The first model (denoted CR and presented in §3.4) captures leaks introduced by computation reuse over arithmetic operations. The second model (denoted CRA) additionally captures leaks by computation reuse on address calculations in address micro-operations.
- **Cache compression (CC):** Cache compression optimizations aims at compressing cache lines, thereby increasing the amount of data that can be stored in caches. We implemented models capturing the effects of two compression strategies: Frequency Pattern Compression (FPC) [2] and Base-Delta-Immediate compression (BDI) [56]. The former compresses several common data patterns whereas the latter compresses narrow ranges of values. In a nutshell, both models monitor the execution of memory operations and produce an observation whenever the corresponding memory request might result in compression according to the given strategy.
- **Prefetching (PF):** Prefetchers aim at loading memory blocks into the cache hierarchy before these blocks are requested by instructions. We implemented \( \text{LmSpec} \) models capturing the effects of three different prefetching strategies: (1) next-line prefetching (PFNL) [7], which prefetches the next memory block for any load operation, (2) stream prefetching [59] (PFS), which detects whether the program is accessing addresses at a regular stride and prefetches further memory blocks along the stride, and (3) data-dependent prefetcher (PFDD) based on behavior observed on the Apple M1 chip [68]. All these models (1) monitor the execution of load micro-operations, (2) check whether further memory blocks need to be prefetched according to the corresponding strategy, and (3) append the prefetched memory addresses to the leakage trace.

6.1.2 Prediction clauses. We implemented in \( \text{LmSpec} \) six different prediction clauses, the default sequential prediction clause (denoted by SEQ and represented in \( \text{LmSpec} \) by the absence of any \text{defpredictor} statement), and five additional clauses, capturing different speculation mechanisms. In particular, our models cover all speculation mechanisms that have been formalized in the literature [20]. The implemented models are:

- **Conditional branch speculation (PHT):** The prediction clause for this model, presented in Figure 2, captures speculating over branch instructions following the so-called “always mispredicts” semantics [26].
- **Straight-line speculation (SLS):** We implemented an \( \text{LmSpec} \) model, illustrated in §4.2, that captures the effects of straight-line speculation implemented in some AMD cores [77]. The model always speculates for a fixed number of steps beyond any jump micro-operation.
- **Store bypass speculation (STL):** The \( \text{LmSpec} \) model illustrated in §4.3 captures the effects of speculation over store bypasses [30].
Following [20], our model speculatively ignores issued store micro-
operations for a fixed number of steps.

**Return address speculation (Rsb):** We implemented two clauses
modeling speculation over return instructions. Both models employ
a return stack buffer to determine the speculation target, but they
differ in how they handle buffer under- and over-flows. One model,
denoted Rsb\_a, and presented in §4.2, uses a circular buffer that
wraps around on over- or underflows. The other, denoted as Rsb\_b,
is inspired by the return speculation in [20]. It simply drops its
oldest entry on overflow and halts (refuses to speculate further) on
underflow.

6.1.3 Assessment. In terms of expressiveness, we observe that our
leakage models span a large class of microarchitectural leaks. In
particular, our leakage clauses cover both standard models like
constant-time as well as advanced optimizations (cache compres-
sion and prefetching) and examples from all optimization proposals
studied in [60]. Similarly, our prediction clauses cover multiple
different speculation mechanisms and all speculation mechanisms
used in state-of-the-art tools [20, 50]. To the best of our knowl-
dge, this is the largest library of leakage models (108 models) to
be implemented in a single language/tool.

Regarding the conciseness of our LmSpec implementations, we
observe that most of the leakage and prediction clauses are imple-
mented in a few lines of LmSpec. For instance, the prediction clauses
are all implemented in less than 10 lines of LmSpec code each. We
remark that LmSpec allows directly leveraging Python’s standard
library and data structures, which greatly simplifies implementing
more complex models (like computation reuse or cache compres-
sion). Overall, all the aforementioned leakage and prediction clauses
can be implemented in about 500 lines of code, including Python
code for handling data structures like the computation reuse cache
and C code for the implementation of cache compression strategies.

6.2 RQ2: Robustness of cryptographic libraries

Here, we report the results of our analysis of the security of sev-
eral real-world cryptographic libraries against the leakage models
from §6.1, which we conducted using LmTest. In the following, we
first present the test subjects (§6.2.1) and the overall experimental
setup (§6.2.2). We conclude by presenting our results (§6.2.3).

6.2.1 Test subjects. We focus on eight commonly used crypto-
graphic algorithms that span a variety of use cases for crypto-
graphic software: AES and SHA512 are widely used primitives for
encryption and hashing, respectively; poly1305 and salsa20 are the
default primitives used for authenticated encryption in the popu-
lar cryptographic library libodium; and ed25519 and x25519 are
elliptic-curve primitives. We also analyze the HMAC and stream-
XOR constructions to see any leakage effects from higher-level
cryptographic operations.

We test the implementations of these algorithms from five differ-
ent cryptographic libraries: libodium [40], a popular cryptographic
library written in C and designed for ease-of-use and safe defaults;
cryptlib [14] and libninetle [49] as alternative libraries also written
in C, to compare results between different implementations; Rust
Crypto [55] to examine how the high level safety guarantees pro-
vided by Rust [43] may affect leakage results; and libjade [22], a
library written in the Jasmin secure assembly language [4] and verified
to be constant-time.

We compiled each library with their respective default settings
and compilers.4 Since some libraries only implement a subset of
the studied algorithms, this results in 25 different cryptographic
implementations. For each implementation, we manually created
a wrapper function that (1) runs the algorithm implementation, and
(2) annotates the algorithm inputs as secret or public for LmTest.

6.2.2 Experimental setup. For each of the test subjects, we use Lm-
Test to test their security against all the leakage models from §6.1.
Specifically, for each test subject trg and leakage model M, we use
LmTest to (a) generate 100 test cases, where each test case consists
of a pair of low-equivalent initial states, (b) run trg for all test cases
and collect the leakage traces generated by the model M, and (c)
analyze the leakage traces for leaks (i.e., checking if the traces for
a single test case are different). For each test subject and leakage
model, we impose a total timeout of 240 minutes and a per-test-case
timeout of 30 minutes, and stop the testing whenever one of the
timeouts expires. We ran our testing campaign on an Intel Xeon
Gold 6132 running Ubuntu 22.04.2 LTS.

6.2.3 Assessment. Table 1 summarizes the results of our testing
campaign. The overall finding is that all the analyzed implementa-
tions leak. This confirms that the optimization proposals studied in
[60], if implemented, could potentially introduce security issues in
modern cryptographic implementations. The testing campaign also
highlights several important observations:

- For the majority of leakage clauses and implementations, LmTest
can already detect leaks under the StQ prediction clause.
- There are several cases, however, where LmTest can only detect
leaks under the speculative prediction clauses (e.g., computation
reuse optimizations in libninetle, rust-crypto, and libjade). This
indicates that, similarly to what happens in Spectre attacks [35],
the interplay between speculative execution and microarchitectural
optimizations can weaken the security of implementations.
- The use of memory-safe languages like Rust does not seem to
significantly improve the security against the new leakage models.
In most cases, the implementations from the rust-crypto library
are as leaky as the corresponding C implementations of other
libraries.
- Constant-time programming does not prevent leaks under these
new leakage models. Despite all analyzed implementations except
AES-CBC begin constant-time (and libjade being provably so),
LmTest still detects leaks in them. As we show in §6.3, specific
constant-time idioms, such as constant-time swaps, introduce
leaks under several of the studied leakage clauses and result in
leaks that allow recovering secret keys directly from leakage
traces.

6.3 RQ3: Exploitation of leaks

To illustrate the security relevance of the leaks identified by Lm-
Test, we performed an in-depth analysis of our findings against

4We compiled libodium v1.0.18 with gcc v11.4.0, cryptlib v3.4.6 with clang v14.0.0,
libninetle v3.8 with clang v14.0.0, Rust Crypto using sha2 v1.10, salsa20 v10.10.2, poly1305
v0.8.0, and x25519-dalak v2.0.0 all with cargo 1.73.0-nightly, and libjade v2023.05-1
with jasmine v2023.06.0.
5libninetle and cryptlib implement AES using non-constant-time lookup tables.
Table 1: Results of our testing campaign. Filled-in diamonds ♦ denote that LMTEST detected a secret-dependent leak for a specific configuration (library, algorithm, leakage clause, prediction clause), while outlined diamonds ◊ denote that LMTEST detected no leaks during testing. Crosses × indicate that LMTEST timed out before completing the testing. To aid readability, we visually combine results for (library, algorithm, leakage clause) triples if they are the same for all prediction clauses.

<table>
<thead>
<tr>
<th></th>
<th>CT</th>
<th>SS</th>
<th>RFC</th>
<th>CS</th>
<th>OP</th>
<th>CR</th>
<th>CC</th>
<th>PF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>libodium</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>salsa20</td>
<td>♦</td>
<td>◊</td>
<td>♦</td>
<td>♦</td>
<td>◊</td>
<td>♦</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>poly1305</td>
<td>◊</td>
<td></td>
<td>◊</td>
<td></td>
<td></td>
<td>◊</td>
<td></td>
<td>◊</td>
</tr>
<tr>
<td>sha512</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td></td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>hmac</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>ed25519</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>x25519</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>stream-xor</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td><strong>cryptlib</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aes-cbc</td>
<td>♦</td>
<td>◊</td>
<td>♦</td>
<td>♦</td>
<td>◊</td>
<td>♦</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>sha512</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td><strong>libnnettele</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aes-cbc</td>
<td>♦</td>
<td>◊</td>
<td>♦</td>
<td>♦</td>
<td>◊</td>
<td>♦</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>salsa20</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td></td>
<td></td>
<td>◊</td>
<td></td>
<td>◊</td>
</tr>
<tr>
<td>sha512</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>hmac</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>ed25519</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>x25519</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td><strong>rust-crypto</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>salsa20</td>
<td>♦</td>
<td>◊</td>
<td>♦</td>
<td>♦</td>
<td>◊</td>
<td>♦</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>poly1305</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>sha512</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>x25519</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>stream-xor</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td><strong>libjade</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>salsa20</td>
<td>♦</td>
<td>◊</td>
<td>♦</td>
<td>♦</td>
<td>◊</td>
<td>♦</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>poly1305</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>sha512</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>x25519</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>stream-xor</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
<td>◊</td>
</tr>
</tbody>
</table>

The results for prediction clauses are: (Σ) leak found / (Π) not found / (τ) time out.

6.3.1 X25519 compare-and-swap. Figure 6 depicts the function showing a simplified version of the compare-and-swap algorithm used in X25519. The two curve points to be swapped, \( f \) and \( g \), are encoded as 5-element uint64_t arrays. The secret bit \( b \) controls whether the elements should be swapped. Since the function is called for each bit of the secret key, learning \( b \) at each iteration allows an attacker to recover the secret key.

The conditional swap is implemented without using conditional branches. First, the condition bit \( b \) is expanded into 64-bit mask
Whenever the mask is 0, then the mask will also be 0 and thus too will become 0; performing the xors will leave each structure unchanged. On the other hand, if b was 1, then the mask will be 0xffff...ff and x will remain unchanged. Since x is already the xor of each structure, xor’ing it back into each structure will serve to swap the values. Finally, the curve points f and g are written back to memory.

Under the constant-time leakage model, this implementation has no secret-dependent leaks, as it contains no branches nor secret-dependent memory accesses. Unfortunately, it still leaks under several of the leakage models from §6.1, as we show next.

6.3.2 Register compression (RFC0, RFCN). The value of mask is derived from the secret bit b and is then used to mask x (line 7). Whenever the mask is 0, the resulting operations will also produce 0; in the RFC models, we will observe this as a cluster of compressions to the zero register, which are recorded in the leakage traces. This allows us to determine whether the original condition bit was 0 or 1 in each loop iteration.

6.3.3 Computation simplification (CST). After the temporary value x has been masked (line 7), it is xor’ed back against f and g respectively to perform the swap (lines 9–10). Whenever the mask is 0, the x will also be 0. However, the xor’ing 0 with any value leaves it unchanged; thus in the CST model, we will observe any such xor operations as being simplified, which is recorded as part of the leakage trace. Again, this allows us to recover the condition bit b.

6.3.4 Silent stores (SS). After the values of f and g are possibly-swapped, the two points are written back to memory. Whenever the values are not swapped, the memory writes on lines 9–10 do not modify the memory. In the SS model, these stores will be suppressed and produce observations in the leakage traces, which let us recover b.

6.3.5 Assessment. Our analysis highlights that, for several leakage models, the leaks detected by LmTest can be exploited to recover a secret key from the leakage traces. In particular, the leaks are directly caused by the constant-time implementation of the compare-and-swap algorithm introduced to prevent side-channel leaks under the standard constant-time model. We also remark that the exploited leaks are already present under the SkQ prediction clause, i.e., they do not result from speculative instructions.

7 DISCUSSION

Scope of the models: The goal of the leakage models presented in this paper is to capture the core aspects associated with the studied microarchitectural optimizations while (a) enabling testing of real-world cryptographic implementations and (b) illustrating the expressiveness of the LmSpec language. As a result, our models simplify many aspects of modern CPUs, which might influence their faithfulness to any specific hardware. Note also that some of our models are associated with optimization proposals rather than concrete implementations; different microarchitectural implementations of the same proposal, therefore, might result in different leakage profiles.

Scope of the results in §6: Our investigation highlighted that optimization proposals can potentially compromise the security of current cryptographic implementations, despite the use of constant-time programming. Lifting the results of our testing campaign to real-world CPUs, however, is only possible to the extent that our leakage models precisely capture the microarchitectural information flows in these CPUs. Even for the optimizations currently implemented by modern processors (e.g., prefetching), LmTest results might incorrectly classify programs as secure or insecure due to mismatches between actual and modeled leaks.

Regarding the exploitability analysis in §6.3, an important caveat is that our analysis only refers to leakage models, given that the studied optimizations are (to the best of our knowledge) not yet implemented in modern CPUs. Even considering a CPU implementing the target optimizations, lifting our analysis to a full-blown attack might be challenging for two reasons. First, our models abstract away from many detailed aspects of modern CPU microarchitectures and, thus, might not faithfully capture all leaks happening on a specific CPU. Second, our analysis assumes that every leakage observation is immediately visible to an attacker. However, in a practical setting, the attacker will not have access to such precise observations. Instead, they will only be able to observe (noisy) measurements, e.g., variations in program execution time.

Scope of LmSpec and LmTest: We see LmSpec and LmTest as a way for both hardware vendors and cryptographic developers alike to easily study the security implications of microarchitectural optimization proposals. This will enable identifying potential leaks during the development of new microarchitectural optimizations before their implementation in silicon, thereby enabling programmers to develop program-level countermeasures early on.

8 RELATED WORK

Comparison with the Pandora works: Here, we review [60], which is a direct inspiration for our work, and a recent follow-up paper [21].

Sanchez Vicarte et al. [60] conduct a systematic review of the security implications of microarchitectural optimizations and perform an in-depth analysis of seven classes of microarchitectural optimizations. Their work provides semi-formal descriptions of the leakage models associated to several optimizations (e.g., operand packing and computation reuse). Their descriptions are based on the notion of microarchitectural leakage descriptors (MLDs). However, these descriptors are informal and often incomplete. For instance,
Attacks related to the studied leakage models: Here, we review existing attacks targeting leaks related with the leakage models studied in this paper.

Ciphertext attacks [38, 39] exploit the memory encryption in AMD SEV-SNP which employs a tweakable encryption mode where a ciphertext depends on a plaintext and a physical address. Whenever a store operation at location \( n \) happens, an attacker can infer whether the new value is different from the old value at that location by observing changes in the ciphertext at \( n \) (ciphertexts are different iff plaintexts are different). This is exactly the same leakage model as for silent stores ([S3.2]). Hence, mitigations against ciphertext attacks [16, 75] should be effective also against silent store leaks.

Finally, the Augury attack [68] exploits the 1-level pointer chasing prefetcher implemented in M1 processors, whereas the SafeCracker attack [66] exploits data compression schemes in caches to infer the content of cache lines. The leakage clauses for prefetching (PFDD) and cache compression (CC) used in our case study are inspired by the leaks exploited in [66, 68].

Attacks outside our leakage models: Several recent works have exploited leakage through power consumption of the CPU, either directly [41] or indirectly through the impact of the power consumption on the CPU heat and frequency [63, 70, 71]. As these attacks do not observe the microarchitecture directly, it is not clear whether they can be modeled in LmSpec.

RAMBleed [36] exploits the Rowhammer effect [34] to leak data from memory. The attack leaks data at rest, and it is not affected by execution models. Thus it is less compatible with our framework. Similarly, attacks that exploit data compression [5, 61, 69] rely on vulnerabilities in software, which are out of scope for LmSpec.

Formal models and analysis: Many formal models capturing timing leaks at microarchitectural level have been proposed. Initially, researchers proposed models capturing leaks associated with “constant-time” [5, 46], e.g., by instrumenting a program’s semantics to produce leakage traces exposing memory accesses and control-flow. More recently, researchers have proposed models capturing leaks associated with speculatively executed instructions. Some models [20, 26, 27, 47, 54] extend program-level semantics with dedicated observations to capture microarchitectural side effects (like cache accesses) and capture the effects of speculatively executed instructions at high level by allowing the program semantics to explore mispredicted paths for a fixed number of steps [26]. Other models [13, 25, 27, 44, 67] rely on more complex models that explicitly capture components like pipeline stages, caches, and branch predictors.

The LmSpec language provides a way of rapidly prototyping and formalizing these leakage models. As a proof of LmSpec’s expressiveness, we used it to capture a large class of leakage models. In particular, beyond the constant-time model (Figure 1), we successfully implemented in LmSpec (a) leakage clauses capturing the leaks induced by all optimization classes studied by Sanchez Vicarte et al. [60], and (b) speculative models capturing speculation over branch, store, and return instructions as well as straight-line speculation. We remark that LmSpec’s design took inspiration from prior work: (1) the notions of leakage and prediction clauses is inspired by the models from [27], (2) the modeling of speculation is inspired by the always mispredict speculative semantics from [26], and (3) its executable implementation on top of the Unicorn emulator is inspired by the Revizor testing tool [50].

Testing for leaks: Here, we review relevant prior work on detecting leaks using testing-based approaches.

There are several approaches for detecting leaks in programs against specific leakage models. For instance, CtFuzz [28] and CtGrind [15] detect leaks against the constant-time model (see Figure 1 for its LmSpec encoding). In particular, CtFuzz constructs a self-composition of the program under test with itself, which is then fuzzed for violations (i.e., by inspecting the traces produced by self-composed program) using the Artfuzz. This is different from LmTest, which executes the program under test on individual inputs and compares pairs of traces. In contrast, CtGrind [15] allows checking violations of constant-time on top of ValGrind [48] using taint-tracking. Finally, SpecFuzz [52] detects speculative bound check bypasses (BCB) against an always-mispredict speculation.
model capturing speculation over branch instruction (i.e., the model in Figure 2). Differently from LmTest, however, all these approaches are tied to specific leakage models.

Rather than relying on a leakage model, Dudect [58] detects side-channel leaks in programs by directly performing hardware measurements. Therefore, Dudect can detect actual leaks against commercial processors. Finally, Microwalk [74, 76] employs binary instrumentation to collect leakage log from functions, detecting leakage when logs are affected by change in secret variables.

Testing has also been used to automatically discover leaks in processors rather than specific programs. For instance, tools like Scam-V [12, 47] and Revizor [29, 50, 51] can be used to detect leaks in commercial processors against a leakage model used as a specification. Other approaches [24, 45, 73], instead, detect leaks by analyzing hardware measurements without the help of a formal leakage model. Finally, tools like SpecDoctor [31], SIGFuzz [57], and AutoCC [53] can test processor designs for leaks and they are applicable in the pre-silicon phase. We remark, however, that all these tools differ in scope from LmTest: LmTest detects leaks in programs, whereas these tools detect leaks in the CPU under test.

Finally, Pensieve [78] is a framework for evaluating the security of early-stage microarchitectural defenses against Spectre attacks. It allows hardware developers to (a) specify a given countermeasure on top of an out-of-order processor model, and (b) find counterexamples to speculative non-interference [26] using model checking. Thus, Pensieve aims at finding problems in hardware countermeasures. In contrast, our approach aims that evaluating the implications of microarchitectural proposals on the side-channel guarantees of programs (even beyond speculative execution).

9 CONCLUSION

In the future, chip vendors are expected to implement new and more aggressive microarchitectural optimizations to speed up computation. Given the extended development time of hardware mitigations, and the performance cost and often slow adoption rate of software countermeasures, it is critical that the security analysis of these new optimizations happens early on during their development, ideally before their availability in commercial processors.

To enable this early-stage security analysis, we proposed a framework (consisting of the LmSpec language and the LmTest testing tool) for evaluating the side-channel guarantees of programs against (future) microarchitectural optimizations. With our framework, we performed a large-scale study of the implications of a large class of optimizations, recently identified as security-critical, on the security of mainstream cryptographic libraries. Our results confirmed that these optimizations, if implemented, would compromise the security of all analyzed libraries.

ACKNOWLEDGMENTS

We would like to thank Boris Köpf for his feedback on earlier versions of this paper.

This work was partially supported by the Spanish Ministry of Science and Innovation under the project TPD2021-132464B-I00 PRODIGY; the Spanish Ministry of Science and Innovation under the Ramón y Cajal grant RYC2021-032614-I; the Spanish Ministry of Science and Innovation under the project PID2022-142290OB-I00 ESPADA; the Australian Research Council Discovery Project DP210102670; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany’s Excellence Strategy - EXC 2092 CASA - 390781972; the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425 the Defense Advanced Research Projects Agency (DARPA) under contract number W911CG-23-C-0022 the National Science Foundation (NSF) under grant number CNS-1954712 and gifts from Intel, Qualcomm, and Cisco.

REFERENCES

[23] Qian Ge, Yuval Yarom, David A. Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary
Testing side-channel security of cryptographic implementations against future microarchitectures

A.1 SSI

```lisp
(defleakage SilentStoreInitializedOnly [initialized 
  (setv)]
  (on [write :reg := val]
    (when (and (in :reg X86_64_GPRS)
      (= :val 0))
      (exists reg_i X86_64_GPRS
        (=> reg_i X86_64_GPRS
          (when (!= :reg_i reg)
            (= :val (sfor :read reg_i))))
        #("rfc" :reg val)))))
```

A.2 SSI0

```lisp
(defleakage SilentStore0InitializedOnly [initialized 
  (setv)]
  (on [write :reg := val]
    (when (and (in :reg X86_64_GPRS)
      (= :val 0))
      (exists reg_i X86_64_GPRS
        (when (and was-init
          (= :val (sfor :read reg_i))))
        #("rfc" :reg val)))))
```

A.3 RFC0

```lisp
(defleakage RegisterFileCompression0 []
  (on [write :reg := val]
    (when (and (in :reg X86_64_GPRS)
      (= :val 0))
      (exists reg_i X86_64_GPRS
        (when (!= :reg_i reg)
          (= :val (sfor :read reg_i))))
        #("rfc" :reg val)))))
```

A.4 RFCN

```lisp
(defleakage NarrowRegisterFileCompression []
  (on [write :reg := val]
    (when (and (in :reg X86_64_GPRS)
      (= :val 0))
      (exists reg_i X86_64_GPRS
        (when (!= :reg_i reg)
          (= :val (sfor :read reg_i))))
        #("rfc" :reg val)))))
```

A.5 CS

```lisp
(defleakage SemiTrivialComputationSimplification []
  (on [(expr (op v1 v2))]
    (when (is-any-of op
      ST-ADD :if (or (= v1 0) (= v2 0))
      ST-SUB :if (or (= v1 0) (= v2 v2))
      ST-MUL :if (or (= v1 0) (= v2 0))
      ST-DIV :if (or (= v1 0) (= v2 2))
      ST-AND :if (or (= v1 0) (= v2 0))
      ST-XOR :if (or (= v1 0) (= v2 0)); XXX should
        this also have (= v1 v2)?
      )
    )
    #("cs" op v1 v2)))
```

A.6 CST

```lisp
(defleakage TrivialComputationSimplification []
  (on [(expr (op v1 v2))]
    (when (is-any-of op
      T-OR :if (or (= v1 ALL1) (= v2
        )
      )
    )
    #("cs" op v1 v2)))
```

References


[72] Pawel Wieczorkiewicz. 2022. The AMD Branch (Mis)predictor Part 2: Where No CPU has Gone Before (CVE-2021-26341). https://grsecurity.net/amd_branch_mispredictor_part_2_where_no_cpu_has_gone before.


[77] Pawel Wieczorkiewicz. 2022. The AMD Branch (Mis)predictor Part 2: Where No CPU has Gone Before (CVE-2021-26341). https://grsecurity.net/amd_branch_mispredictor_part_2_where_no_cpu_has_gone before.


A.7 CN
(setv NARROW_CS_LIMIT (<< 1 32))
(defleakage NarrowComputationSimplification []
(on [expr (op v1 v2)]
  (when (and (in op ST-MUL)
    (< v1 NARROW_CS_LIMIT)
    (< v2 NARROW_CS_LIMIT))
    #("cs" op))))
A.8 OP
(setv OP_CTX_SIZE 200)
(defleakage OperandPacking [ctx (deque)]
(on [expr (op v1 v2)]
  (when (and (< v1 16) (< v2 16))
    (while (and ctx (>= (- stick (. ctx [0][0]))
      OP_CTX_SIZE))
      (.popleft ctx))
    (for [[i [tick_i op_i]] enumerate ctx]]
      (when (< op_i op)
        (del (. ctx [i]))
        (return #("op" op_i i)))
    (else
      (.append ctx #(& stick op)))))))
A.9 CRA
(setv ComputationReuseWithAddresses [ctx
  (OrderedDict)]
ctx-addrs (OrderedDict)
ctx-loads (OrderedDict)
(on [expr (op #* vs)]
  (when (in op CACHEING_OPS)
    (if (in vs (. get ctx bpc #()))
      (update ctx &pc vs)))
    [(addr base + index * scale + off)
      (if (in #(base index scale off) (. get ctx-addrs &pc
        #())))
        (update ctx-addrs &pc [base index scale off]))]
    (if (in addr (. get ctx-loads &pc #()))
      (update ctx-loads &pc addr)))))
A.10 CC – FPC
(setv FPCACHECHAIN [iter
  (Queue)]
  (on [load [addr].sz]
    (let [block-addr (<< (> addr CACHELINE_BITS)
      #CACHELINE_BITS]
      block-data (mem.read-bytes block-addr
        #CACHELINE_SIZE)
      #("cc" (fpc-size block-data))]
    ([store [addr].sz := val]
      (let [block-addr (<< (> addr CACHELINE_BITS)
        #CACHELINE_BITS]
      block-data (mem.read-bytes block-addr
        #CACHELINE_SIZE)
      offset (%
      write-into block-data offset sz val)
      #("cc" (fpc-size block-data)))))))
A.11 CC – BDI
(setv BDICacheCompression []
  (on [load [addr].sz]
    (let [block-addr (<< (> addr CACHELINE_BITS)
      #CACHELINE_BITS]
      block-data (mem.read-bytes block-addr
        #CACHELINE_SIZE)
      #("cc" (bdic-size block-data))]
    (store [addr] := val)
    (let [block-addr (<< (> addr CACHELINE_BITS)
      #CACHELINE_BITS]
      block-data (mem.read-bytes block-addr
        #CACHELINE_SIZE)
      offset (%
      write-into block-data offset sz val)
      #("cc" (bdic-size block-data)))))))
A.12 PFNL
(setv POINTER_SIZE 8)
(setv CACHELINE_BITS 6)
(setv PAGE_BITS 12)
(defleakage NextLinePrefetch []
  (on [load [addr].sz]
    (let [cache-index (>> addr CACHELINE_BITS)]
      #("pf" (+ cache-index 1))))))
A.13 PFS
(setv PF_HITS 3)
(defn diff [ns]
  (setv [ns1 ns2] (tee ns))
  (next ns2)
  (ifor [m] (zip ns1 ns2) (~ m n))
  (define direction-of? [page-hits]
    (when (< (len page-hits) page-hits.maxlen)
      (return 0))
    (setv diff (diff page-hits))
    (cond (forall n diff (> n 0)) 1
      (forall n diff (< n 0)) -1
      :true 0)
    (define StreamPrefetch [all-page-hits (dict #%
      (on [load [addr].sz]
        (let [cache-index (>> addr CACHELINE_BITS)
          #CACHELINE_BITS]
          page-index (>> addr PAGE_BITS)
          page-hits (get all-page-hits page-index)
            (when (not-in cache-index page-hits)
              (append page-hits cache-index))
            stream-dir (direction-of? page-hits)
            next-cache-index (stream-dir cache-index)]
        (when (and stream-dir
          (=> next-cache-index (~ PAGE_BITS
          #CACHELINE_BITS)) page-index)])
        #("pf" next-cache-index)))))
A.14 PFDD
(setv MIPF_SIZE 20)
(setv MIPF_PREFETCH 5 ; we prefetch 5 elements
(defleakage MIPFPrefetch [initialized (set]
  accesses (deque :maxlen MIPF_SIZE)
  marks (deque :maxlen PF_HITS]
  (on-start [model input]
    (.update initialized (sfor addr
      input mem-initialized (~ model.STACK addr)))
    (on [store [addr].sz := val]
B ADDITIONAL PREDICTION CLAUSES

Here, we provide the LmSpec modelling of the missing prediction clauses from §6.1.2.

B.1 RSB⊥

(setv RSB_SIZE 16)
(defpredictor RSBCircular [stack (* [0] RSB_SIZE) idx [0]]
  ;; RSB that drops oldest entry on overflow
  ;; and halts on underflow
  (on [(jump addr : n)]
    (cond
      (insn.group CS_GRP_CALL)
      (.append stack (+ &pc &insn.size))
      (insn.group CS_GRP_RET)
      (if stack [(.-pop stack)] [HALT]))))