IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2022, No. 4, pp. 751–773. DOI:10.46586/tches.v2022.i4.751-773

SoC Root Canal!

Root Cause Analysis of Power Side-Channel Leakage in System-on-Chip Designs

Pantea Kiaei[1] and Patrick Schaumont[1]

Worcester Polytechnic Institute, Worcester, MA 01609, USA, {pkiaei,pschaumont}@wpi.edu

Abstract. Finding the root cause of power-based side-channel leakage becomes harder when multiple layers of design abstraction are involved. While side-channel leakage originates in processor hardware, the dangerous consequences may only become apparent in the cryptographic software that runs on the processor. This contribution presents RootCanal, a methodology to explain the origin of side-channel leakage in a software program in terms of the underlying micro-architecture and system architecture. We simulate the hardware power consumption at the gate level and perform a non-specific test to identify the logic gates that contribute most side-channel leakage. Then, we back-annotate those findings to the related activities in the software. The resulting analysis can automatically point out non-trivial causes of side-channel leakages. To illustrate RootCanal's capabilities, we discuss a collection of case studies.

Keywords: side-channel analysis · design-time methodology · micro-architecture · root-cause analysis

1 Introduction

A smart card application is a firmware program running on a microcontroller with cryptographic accelerators. In such a system, the analysis of power-based side-channel leakage spans multiple layers of design abstraction, including the hardware and the software. Both hardware and software play a role in analyzing, understanding, and mitigating this power-based side-channel leakage. While the smart card software manipulates the secrets of the application, it is the smart card hardware that lets those secrets escape through physical side-channel leakage.

The *root-cause analysis* of side-channel leakage refers to the set of design activities that help a designer understand the origin of side-channel leakage in terms of each of the relevant design abstraction levels in hardware or software. Root-cause analysis is challenging because of two reasons. First, the interface between hardware and software, the instruction-set architecture (ISA), hides important implementation details such as the micro-architecture state, the memory hierarchy, and the system-level interconnect. As a result, it is notoriously difficult to explain all the side-channel leakage from the software alone. Second, the complexity of modern embedded systems is enormous. A single chip may contain hundreds of thousands of standard cells and hard macros, that jointly implement a processor, memory, peripherals, and cryptographic hardware accelerators. *Any* of these standard cells is a potential contributor to side-channel leakage.

In this contribution, we propose RootCanal, a methodology to identify the origin of side-channel leakage from a white-box implementation of an embedded System-on-Chip design that contains hardware and software. In a pre-silicon white-box design, every detail is known – typically with gate-level accuracy – but no physical realization is available.

Licensed under Creative Commons License CC-BY 4.0.
Received: 2022-04-15 Accepted: 2022-06-15 Published: 2022-08-31

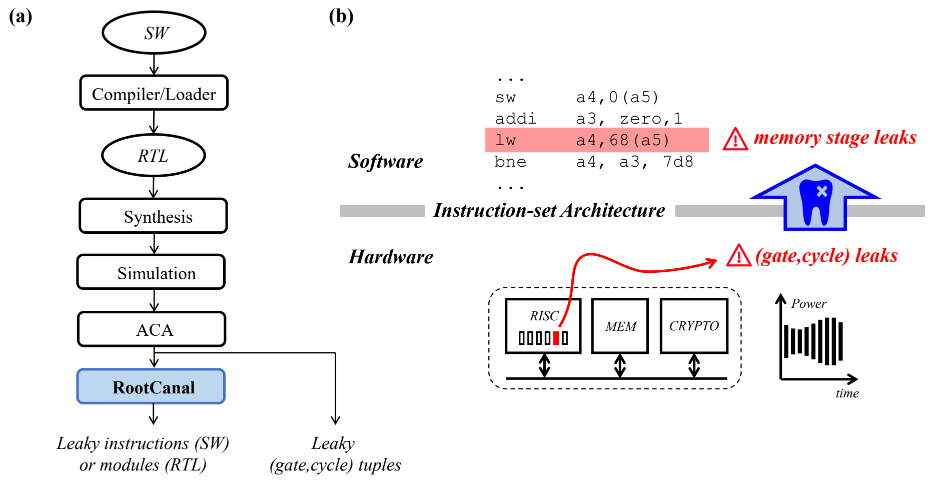


Figure 1: RootCanal is a pre-silicon side-channel leakage assessment technique to back-annotate leaky (gate,cycle) tuples in an SoC design to high-level software or hardware information. (a) RootCanal design flow integration (b) Example application.

The methodology aims to assist the hardware designer during the pre-silicon design phase to build insight into the implementation factors that will cause side-channel leakage. RootCanal replaces or extends the traditional side-channel leakage assessment on an FPGA prototype with simulation-based design automation of the actual design target. The main advantage of using the white-box implementation is that there is no ambiguity on the cause of power variations in the hardware, as we simulate a high-resolution power simulation at gate-level accuracy for the overall design.

The RootCanal method in a nutshell Figure 1 captures the main purpose of RootCanal. A hardware designer wishes to build insight into the factors of hardware and software that cause side-channel leakage in an SoC design that may include a processor, memory, and hardware accelerators. RootCanal starts from the design source code (HDL and software) and proceeds in two steps. First, using a non-specific leakage detection method on the synthesized gate-level netlist, RootCanal determines a list of *(gate, cycle)* tuples during which the design shows side-channel leakage. Second, RootCanal maps the list of leaky gate tuples into high-level design information as follows. A leaky tuple within a processor maps into the corresponding instruction of the embedded software that causes the leakage. A leaky tuple outside a processor maps to the module and HDL source code location of the RTL construction that causes the leakage. From this high-level design information, the designer learns about the root cause of the leakage in terms of the design's source code.

Benefits from RootCanal RootCanal tests the SoC side-channel leakage before tape-out, resulting in high prototyping cost savings. The use of gate-level power simulation offers high accuracy and low ambiguity on the amount and origin of the power consumption variations (with some limitations as discussed below). RootCanal uses a non-specific testing method on the simulated power traces and evaluates a broad spectrum of power-based side-channel leakage covering the SoC hardware, the micro-architecture, and the software. RootCanal can be used in practical applications such as testing if the integration of hardware modules in an SoC may cause side-channel leakage and testing if a software-based countermeasure works on the SoC hardware. The main contribution of RootCanal is to locate the source of side-channel leakage of a design in terms of the source code of the design[1]. Indeed, there is a significant semantic gap between a gate-level netlist and its high-level specification from RTL of software [ASA+21].

[1]We refer to the designer as a *hardware* designer to emphasize that we are dealing with a pre-silicon scenario. But this designer may very well be writing test software for the SoC processor too.

Limitations of using a Power Simulation RootCanal supports a pre-silicon scenario using simulated power traces from a gate-level model of the SoC. This requirement brings two caveats. First, power simulation tools are orders of magnitude slower when compared to measurements from a physical prototype. Although a power simulation delivers a noiseless trace, which significantly reduces the number of traces needed for a reliable statistical test of side-channel leakage, the time spent by RootCanal on power simulation of an SoC is still dominant compared to logic synthesis and logic simulation time. However, our results show that current commercial tooling for power simulation is sufficiently powerful to handle complete SoC analysis of power-based side-channel leakage. Second, no simulation-based method can guarantee that the implementation will be completely free from side-channel leakage since no simulation reflects the actual physical detail of the implementation with total accuracy. RootCanal uses post-synthesis gate-level power simulation, which captures technology-specific static, dynamic and internal gate-level power as well as sub-cycle timing effects such as glitches. However, post-synthesis power simulation does not capture capacitive coupling effects, and it does not capture off-chip factors such as coupling effects from a chip package or a PCB. The accuracy limitations of post-synthesis power simulation imply that false negatives in leakage detection are possible. However, the simulation-based method suffers no false positives: a side-channel leak identified by RootCanal in simulation will also occur in the physical implementation.

Related Work The modeling of side-channel leakage has received significant interest in recent years. Buhan *et al.* partition the world of side-channel assessment tooling according to the availability of silicon [BBYS21]. Pre-silicon tooling estimates side-channel leakage through simulation, while post-silicon tooling builds side-channel leakage models from measurements. Both types of tools serve different purposes. Pre-silicon techniques are helpful while validating the side-channel leakage of new hardware designs. Post-silicon techniques can handle the analysis of commercial-off-the-shelf (COTS) chips for which no design internals are publicly available, which may help software side-channel verification.

RootCanal belongs to the category of pre-silicon tools, which have traditionally focused on hardware-level simulations. Le Corre *et al.* use an HDL simulation of a Cortex-M3 core to build a leakage model [CGD18]. Their *MAPS* simulator captures the impact of micro-architecture level pipeline registers on the side-channel leakage from software. While specific to ARM Cortex-M3, this effort was among the first to show the utility of processor-aware modeling tools to predict the side-channel leakage properties of software. In the PARAM design, Arsath *et al.* use simulation for prediction of side-channel leakage in a processor design. They rank each processor module according to the level of contributed side-channel leakage. Several other authors have proposed techniques to locate the source of side-channel leakage in a hardware design, such as RTL-PSC [HPN+19], Architecture Correlation Analysis [YKES20], and KARNA [SVRK19]. These tools rank the activities of hardware elements according to a localized leakage score.

Several groups propose formal verification for the verification of masking-based countermeasures [BGI+18, BBC+19, KSM20]. These pre-silicon tools vary in their support for underlying leakage models, but all use formal techniques to sidestep simulation and instead use symbolic techniques to demonstrate statistical independence of probability distributions. Gigerl *et al.* applied formal verification of masked hardware implementations to analyze micro-architectures that execute masked software implementations [GHP+21, GPM21]. This work highlights the need and benefits of including the hardware details in the verification process of masked software.

In the post-silicon tooling, grey-box models capture the micro-architecture sources of side-channel leakage in significant detail [MOW17, SSB+19, MPW22, BIBB21]. Gao *et al.* proposed a novel test of completeness to measure the quality of side-channel leakage models of processors in the grey-box case [GO21]. We see the post-silicon work as complementary to RootCanal that handles pre-silicon analysis.

Outline In the next section, we review preliminary relevant concepts that support RootCanal. Section 3 presents the methodology with specific attention to the process of translating leaky gates in hardware to leaky instructions in software. Section 4 applies RootCanal to four different case studies. We conclude the paper in Section 5.

2 Preliminaries

We address three preliminary concepts in support of RootCanal. First, the root cause analysis of side-channel leakage must eventually point out an element in the software or hardware specification of the SoC. We will define the abstraction levels of concern in the design hierarchy, paying attention to the tension between an SoC's specification in source code and its realization in logic gates and instruction opcodes. Second, we will describe the properties of the gate-level power simulation used by RootCanal and we highlight its benefits and limitations. Third, we will review Architecture Correlation Analysis (ACA) [YKES20], which is used by RootCanal to identify the source of side-channel leakage in hardware. We will also describe an extension of ACA so that it can test non-specific leakage.

Design Hierarchy RootCanal's input is a System-on-Chip design in a synthesizable Hardware Description Language (HDL) and embedded software running on the SoC processor. We define a software specification as the assembly-level source code. RootCanal flags side-channel leakage at the granularity of an individual logic gate and then propagates this up the hierarchy to a level accessible to a designer. RootCanal will resolve ambiguities such as overlapped instruction execution in software and multiple-instantiated modules in the hardware design hierarchy. RootCanal also deals with discrepancies between the static source code manipulated by the designer, and the runtime view on the system simulated and analyzed by the methodology.

Power Simulation A RootCanal user will study the side-channel leakage over a given system-level simulation interval, such as the rounds of a cipher. In the RootCanal prototype, we use Cadence Joules as a power simulator. The system simulation interval is partitioned into multiple *frames* so that RootCanal obtains equally-spaced power samples over the system simulation interval. The Joules power simulator analyzes the gate-level activity of the design over each frame in the system simulation interval to determine the average power consumption of each gate within each frame. The technology-dependent gate-level power model of Joules captures switching power, internal power, and leakage power. In our experiments, we used a frame interval smaller than or equal to the clock period of the design under test.

The Joules power simulator takes every event within a frame into account to determine the per-frame power estimate. For example, even at a low sample rate of one frame per clock cycle, the power estimate for the frame still includes the power consumption caused by glitches, a known cause of side-channel leakage [MPG05]. The Joules power simulator models the capacitive loading of gates with wire-load models during the initial high-level design. Joules also uses capacitive loading estimated from the actual routing when the design layout is available. A limitation of the Joules power simulator is that it ignores cross-coupling capacitance, which may be responsible for masking order reduction on masked designs [CHS09].

Architecture Correlation Analysis Architecture Correlation Analysis (ACA) is a technique to rank the gates in a hardware design according to their contribution to the side-channel leakage. The gates are ranked based on a specific-leakage test using a leakage model, or a non-specific leakage test [KYL+22]. RootCanal builds on top of non-specific ACA. For a non-specific test, the stimuli are taken from two groups, leading to two groups of power traces. The evaluation works in two steps.

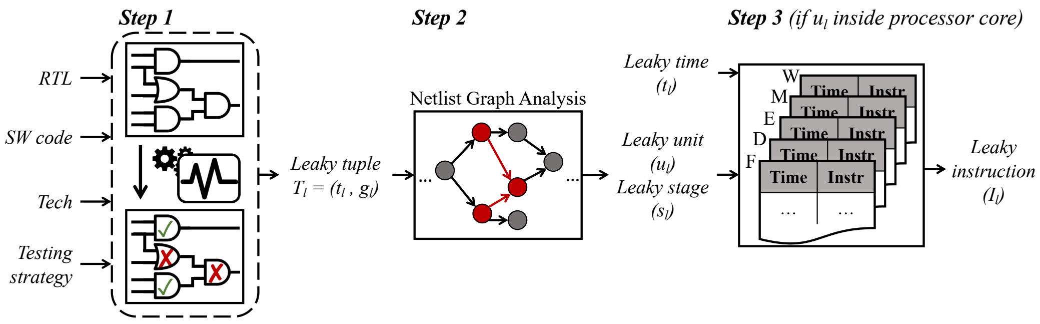


Figure 2: Overall RootCanal flow

1. Non-specific ACA compares the two groups of power traces with a Welch's t-test and flags the collection of frames over which a design leaks ($|t| > 4.5$) as the *Leakage Time Interval* (LTI).
2. Non-specific ACA computes a toggle trace for each gate during the Leakage Time Interval. A toggle trace encodes a gate's output transitions over the LTI, where +1 indicates the presence of at least one transition and -1 indicates the absence. Non-specific ACA correlates the toggle trace with the stimulus group identifier, which is -1 for a stimulus from the first group and +1 for a stimulus from the second group. This group correlation thus expresses how consistently the activity of a gate predicts the group during the leakage time interval. Unlike [KYL+22], the ranking used by RootCanal does not apply gate weighing factors; we found their impact on ranking accuracy to be minor compared to the overhead of computing them.

The leakage ranking of the gates is established by sorting the gates according to their group correlation.

A designer using RootCanal will adjust the test stimuli according to the side-channel leakage property under evaluation. The two groups of stimuli create an internal statistical bias in the gate-level design that is subsequently detected by RootCanal. Table 1 shows the stimuli used for the experimental work of the paper. A *Value Leakage* test evaluates if a sensitive *input* value will appear as side-channel leakage in SoC hardware or software, and is evaluated as a fixed-versus-fixed value test. A *Node Bias* test evaluates the state of an *internal* circuit node by partitioning random input stimuli in two groups according to the internal node. This test helps to evaluate a hiding countermeasure. Finally, in a *Masking* test, a designer tests the correct implementation of a masking scheme using a fixed-vs-random test for input values. The current RootCanal prototype only considers first-order leakage. Extending RootCanal to higher-order leakage testing will require extending non-specific ACA with higher-order testing criteria [SM15].

Table 1: The non-specific tests used for RootCanal compare power traces from Group 1 against Group 2. NAMES in capitals denote inputs. The Node Bias test uses Random INPUT in both groups.

Test	Group 1	Group 2	Purpose
Value Leakage	Fixed `VALUE1`	Fixed `VALUE2`	Testing of VALUE Leakage
Node Bias	`internal_bit=0`	`internal_bit=1`	Testing of Hiding
Masking	Random `INPUT`	Fixed `INPUT1, INPUT2, ..`	Testing of Masking

3 Methodology

RootCanal finds the source (in hardware and software) of power side-channel leakage from an SoC by a three-step process. Figure 2 shows the overall structure of the RootCanal methodology. Root cause analysis starts by performing a gate-level side-channel leakage assessment on the design to obtain a set of leakage tuples $T_l = (t_l, g_l)$, with t_l and g_l indicating the time and gate that cause side-channel leakage (**Step 1** in Figure 2).

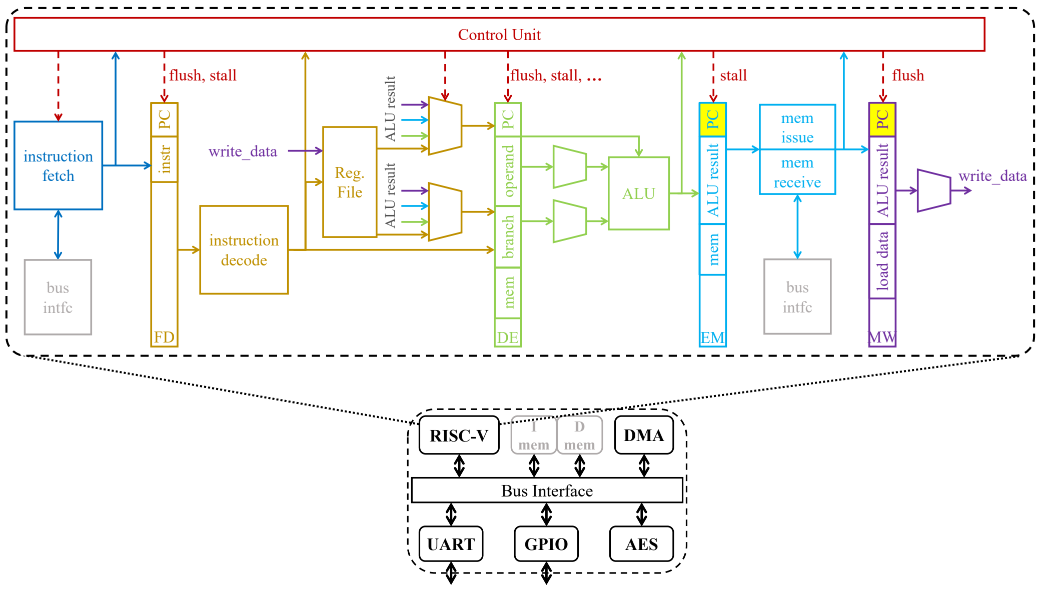


Figure 3: Block diagram of RISCV-SoC and its five-stage RISC-V processor. Resources from different pipeline stages are shown in different colors in the processor core. The gray blocks in the SoC (instruction and data memories) are modeled in the testbench (not synthesized).

Knowing the leaky gate (g_l) without its relation to the RTL design does not provide the secure hardware designer with useful information about the design. Therefore, we introduce a Netlist Graph Analysis (NGA) methodology to find the unit in the design to which the leaky gate g_l belongs, i.e., a leaky unit u_l. In case the processor core in the SoC is pipelined, NGA also reports the pipeline stage to which g_l belongs, i.e., leaky stage s_l (**Step 2** in Figure 2)

Furthermore, when the leakage stems from inside the processor core, it is helpful to know which instruction (or interaction of a group of instructions) has caused the observed leakage. To find the instructions causing a specific leakage tuple $T_l = (t_l, g_l)$, we log the instructions per clock cycle (per pipeline stage) and find the instruction running in the processor (in the leaky stage s_l) during the leaky time t_l (**Step 3** in Figure 2).

In the rest of this section, we elaborate each step of RootCanal. We first describe how the leaky tuples are detected. We further explain how these tuples are translated into the leaky unit of the circuit and instructions that cause the leakage. Throughout this section we use RISCV-SoC (Figure 3) as the running example. We refer to the pipeline stages as F, D, E, M, and W which respectively stand for fetch, decode, execute, memory, and write-back. We refer to the pipeline stage registers in two-letter words showing the pipeline stages surrounding the register (e.g. EM is the pipeline register between stages E and M as shown in Figure 3)

3.1 Step 1: Finding Leaky Time-Gate Tuples

In the first step (Figure 4), RootCanal uses ACA to find the leakage tuples. First, the software source code is compiled and loaded on the synthesized netlist. Next, we prepare two groups of stimuli depending on the planned type of non-specific test (Table 1). We then simulate the netlist's switching activity using the Cadence Xcelium simulator and save the result in value change dump (VCD) format. Using a power simulator, Cadence Joules, we then collect the power traces for all chosen stimuli. ACA then analyzes the power traces and the VCD files to determine the leaky time-gate tuples.

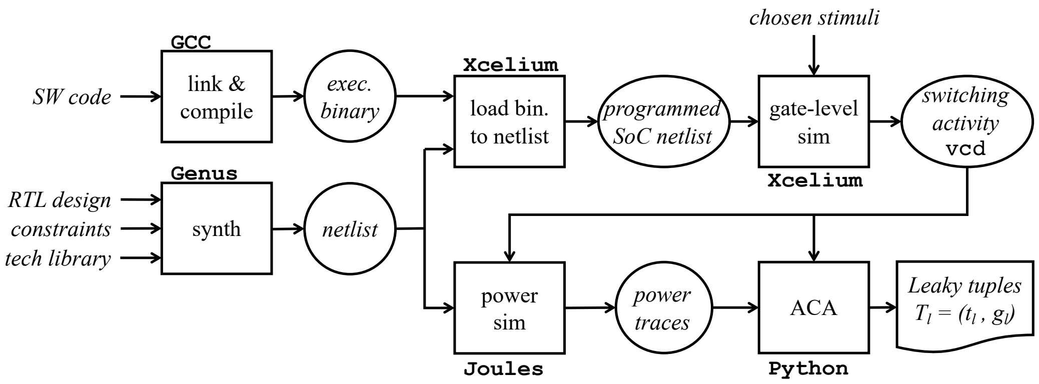
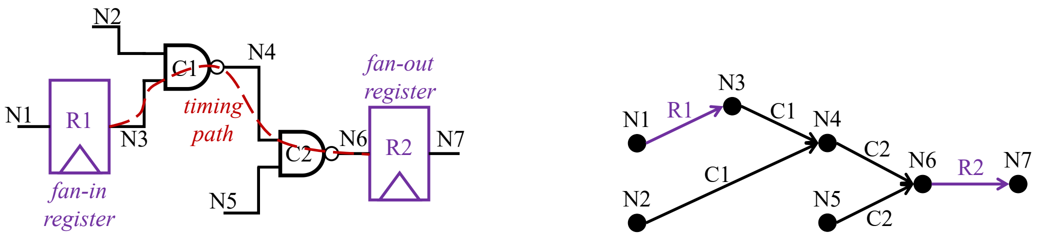


Figure 4: Layout of step 1 in RootCanal



(a) Example circuit **(b)** Corresponding netlist graph

Figure 5: An example for timing path, fan-in register, fan-out register, and gate-level netlist graph.

3.2 Step 2: Finding Leaky Units

To do a root-cause analysis of the leakage, we need to trace back the location of each leaky gate in the design. Commercial synthesis tools and open-source synthesis tools, such as Yosys, support tracking of each gate in the synthesized netlist to their RTL source file. In our tool-chain, for example, we use Cadence Genus for gate-level synthesis, which supports the synthesis attribute `hdl_track_filename_row_col` to trace the location of each gate in the synthesized netlist back to the RTL source file.

However, there are two problems with the source code tracking in synthesis tools. First, the reported RTL location can be incorrect for highly-optimized circuits. For example, in our experiments with RISCV-SoC (Figure 3), we observed many gates in the netlist being attributed mistakenly to the ALU in the processor core. Second, the tools only record the source RTL file name (and line number) of the lowest hierarchy level. Therefore, gates belonging to different instances of the same module appear to come from the same RTL source. For example, the RISC-V core in Figure 3 uses a common pipeline register module (defined in `pipeline_register.v`) in every pipeline stage; therefore, the tool traces the synthesized gates from pipeline registers in different stages to the same `pipeline_register.v` RTL file.

The first problem can be reduced or possibly overcome by preserving the hierarchy of some modules in the design. However, for our purpose, this solution is suboptimal. It interferes with the default synthesis flow and prevents the highest level of optimization of the circuit (incurring higher delay and area). Instead, we introduce a Netlist Graph Analysis (NGA) methodology to detect the source of a synthesized gate in a design, which overcomes both of the mentioned problems.

3.2.1 Definitions

We use the following definitions to describe NGA.

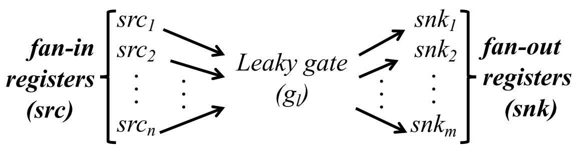


Figure 6: NGA uses fan-in and fan-out registers for each gate to determine its approximate location in the design.

Timing path. A path, in the gate-level netlist, starting from the output of a sequential cell and ending at the input of a sequential cell. The red dashed line in Figure 5a shows a timing path consisting of the nets $\{N3, N4, N6\}$.

Fan-in register. Register R is a fan-in register to logic cell C if there is a path in the gate-level netlist from the output of R to the input of C. In Figure 5a, $R1$ is a fan-in register for cells $C1$ and $C2$.

Fan-out register. Register R is a fan-out register to logic cell C if there is a path in the gate-level netlist from the output of C to the input of R. In Figure 5a, $R2$ is a fan-out register for cells $C1$ and $C2$.

Gate-level netlist graph. A directed graph representation of the gate-level netlist in which nodes represent the nets of the netlist and the edges connect all cell inputs (barring the input clock) to the cell's output. Figure 5b shows the netlist graph for the circuit in Figure 5a.

3.2.2 Netlist Graph Analysis

NGA finds the design unit of a gate in the netlist by locating the gate's fan-in registers and fan-out registers and then inferring the gate's design unit from the registers' design units. Synthesis tools optimize the combinatorial logic between registers to increase the maximum frequency of the design. This optimization may omit/combine much of the combinatorial logic in the design. However, the sequential cells remain in the final netlist, and they offer a stable connection between the RTL and gate-level netlist. Our graph analysis technique relies on the register instance names in the synthesized netlist. These names reveal the register's location in the design (including module instance, hierarchy, and original RTL name). Some synthesis tools preserve the register names by default, while other synthesis tools require a synthesis switch. To locate the fan-in and fan-out registers for gates in a circuit, we add a step at the end of synthesis to report the starting and ending point of timing paths containing each gate in the netlist.

For each given leaky gate g_l, NGA reports its corresponding leaky unit u_l (or leaky stage s_l) by using a set of design-specific rules. NGA views an SoC as a set of IPs connected to each other through a bus (B): $SoC = \{IP_1, ..., IP_n, B\}$. Each IP in the SoC is outlined by its registers: $IP_i = \{r_{i,1}, ..., r_{i,m}\}$. The unit u_l for each leaky gate g_l is detected by comparing the set of fan-in and fan-out registers for g_l with the registers in the IPs. In Figure 6, we label fan-in registers as src_i and fan-out registers as snk_i. The set of rules and the steps involved in NGA varies based on whether the processor core in the SoC is multi-cycle or pipelined.

Multi-cycle processor core. A multi-cycle processor core executes only a single instruction at a time. Therefore, simply knowing the leaky gate g_l is from inside the processor core is enough to locate the instruction causing the leakage at time t_l. To find the design unit u_l to which g_l belongs, we treat a multi-cycle processor core like any other IP in the SoC. Algorithm 1 determines the design unit for a given leaky gate g_l, given the set of its fan-in and fan-out registers (src and snk respectively). If all of g_l's srcs and snks are from the same IP, g_l resides in that IP. Otherwise, it belongs to the bus interface.

Pipelined processor core. A pipelined processor supports overlapped execution of instructions, with each instruction allocated to a different pipeline stage at a given clock cycle. Therefore, we need to know to which stage of the pipeline g_l belongs in order to

Algorithm 1 Single-Phase NGA for an SoC with Multi-Cycle Processor

```
Input: $src$, $snk$                                   ▷ $src = fanin(g_l)$, $snk = fanout(g_l)$
Output: design unit to which $g_l$ belongs
  procedure SINGLE__PHASE__NGA($src$, $snk$)
      if ($src \subseteq IP_k$ ) and ($snk \subseteq IP_k$) then
          return $IP_k$
      else
          return $B$                                  ▷ $B$: bus interface
      end if
  end procedure
```

Algorithm 2 Phase A of NGA for pipelined processor

```
Input: $src$, $snk$                                   ▷ $src = fanin(g_l)$, $snk = fanout(g_l)$
Output: design unit to which $g_l$ belongs
  procedure NGA__PHASE__A($src$, $snk$)
      if ($src \subseteq IP_k$ ) and ($snk \subseteq IP_k$ ) then
          return $IP_k$
      else if ($\{src \cup snk\} \subseteq \cup_{k=1}^{n} IP_k$) then
          return $B$ (IP to IP)                       ▷ $B$: bus interface
      else if (($\exists i\ src_i \in CPU$) and ($\exists i\ src_i \in \cup_{k=1}^{n} IP_k$))
          or (($\exists i\ snk_i \in CPU$) and ($\exists i\ snk_i \in \cup_{k=1}^{n} IP_k$)) then
          return $B$ (interconnect between IP and processor)
      else
          return $\varnothing$
      end if
  end procedure
```

associate a leaky gate with a leaky instruction. In an SoC with a pipelined processor, we differentiate between the processor core and the other IPs: $SoC = \{IP_1, ..., IP_n, B, CPU\}$. A three-phase NGA procedure is able to detect the unit u_l for a leaky gate g_l: (Phase A) Detect whether g_l is inside the processor; (Phase B) Detect the pipeline stage for g_l; and (Phase C) Ensure g_l is not from the processor's control unit.

NGA Phase A. We first identify using Algorithm 2 whether g_l belongs to IPs other than the processor. If all the srcs and snks for g_l are from the same IP, g_l belongs to that IP. If all srcs and snks are from different IPs not including the CPU, g_l belongs to the bus interface between IPs. If either of the srcs or snks is from the CPU, while the other set is from the IPs, g_l is from the bus interface between the CPU and IPs. If g_l is not identified as belonging to IPs (Algorithm 2 returns $\varnothing$), we use phase B and phase C of NGA to detect the pipeline stage s_l for a leaky gate g_l.

NGA Phase B. NGA finds the pipeline stage to which g_l belongs as follows. First, find the fan-out register in the path from g_l to the committing stage of the pipeline (write-back in Figure 3). Next, detect possible combination of different pipeline stages from g_l's fan-in and fan-out registers. Algorithm 3 shows the procedure for phase B of NGA. For this phase, we build the graph from the gate-level netlist and find the shortest path from the output of g_l to the output of the MW pipeline register using Dijkstra's algorithm. Taking the shortest path, prevents finding the path going through the control unit. We then find the first pipeline or general-purpose register (r_0) along this shortest path. If r_0 is a general purpose register (from the register file), g_l belongs to the write-back stage. If r_0 is a pipeline register, the stage right before r_0 is the stage where g_l resides, unless g_l is from the control unit.

Algorithm 3 Phase B of NGA for pipelined processor

```
Input: $netlist\_graph$, $g_l$
Output: pipeline stage to which $g_l$ belongs
  procedure NGA_PHASE_B($netlist\_graph, g_l$)
    $path\_wb \leftarrow$ DIJKSTRA_PATH($netlist\_graph, g_l, pipereg_{MW}$)
    $r_0 \leftarrow$ FIRST_REG($path\_wb$)
    if $r_0 \in pipereg_{k+1}$ then          ▷ $pipereg_{k+1}$: pipeline register after $stage_k$
      return $stage_k$
    else if $r_0 \in GPR$ then          ▷ $GPR$: set of general purpose registers in reg. file
      return $W$          ▷ $W$: write-back stage
    end if
  end procedure
  procedure FIRST_REG($path$)
    for $edge \in path$ do
      if $edge$ is register then
        return $edge$
      end if
    end for
  end procedure
```

Algorithm 4 Phase C of NGA for pipelined processor

```
Input: $stage_k$, $src$          ▷ $stage_k$ = NGA_PHASE_B($netlist\_graph, g_l$), $src = fanin(g_l)$
Output: pipeline stage or control unit to which $g_l$ belongs
  procedure NGA_PHASE_C($stage_k$, $src$)
    if $src \subseteq \{\cup_{k=1}^{n} IP_k \cup pipereg_k\}$ then          ▷ $pipereg_k$: pipeline register before $stage_k$
      return $stage_k$
    else if $(\exists i, j\ src_i \neq src_j)$ and $(stage_k = D)$ then
      return $C$ (can be related to operand forwarding)
    else if $(\exists i, j\ src_i \neq src_j)$ then
      return $C$          ▷ $C$: control unit
    end if
  end procedure
```

NGA Phase C. Phase C of NGA detects whether g_l is from the control unit. The control unit has inputs/outputs from/to all pipeline stages and breaks the independence between stages (Figure 3). We follow Algorithm 4 and use the fan-in registers for g_l and the detected $stage_k$ from phase B to decide whether g_l resides in the control unit.

If all of g_l's fan-in registers align with the detected $stage_k$, we conclude g_l belongs to $stage_k$. Otherwise, if fan-in registers to g_l come from different units, g_l belongs to the control unit. For example, assume that a gate belongs to the operand forwarding logic. In that case, the leakage observed from the gate can originate from any stage (or combination of stages) that forwards data to the leaky gate (D,E,M,W).

3.3 Step 3: Finding Leaky Instructions

Figure 7 illustrates step 3 of RootCanal. To find the instruction that has caused a particular leakage, we use a log of instructions from the processor core simulation. In case of a multi-cycle core, we log the program counter (PC) at every clock cycle. In case of a pipelined processor, we log the PC for each stage separately. To support PC logging for the processor core in Figure 3, we instrument the EM and MW stage registers with the PC signal (highlighted in yellow in Figure 3) for an RTL simulation of the SoC running

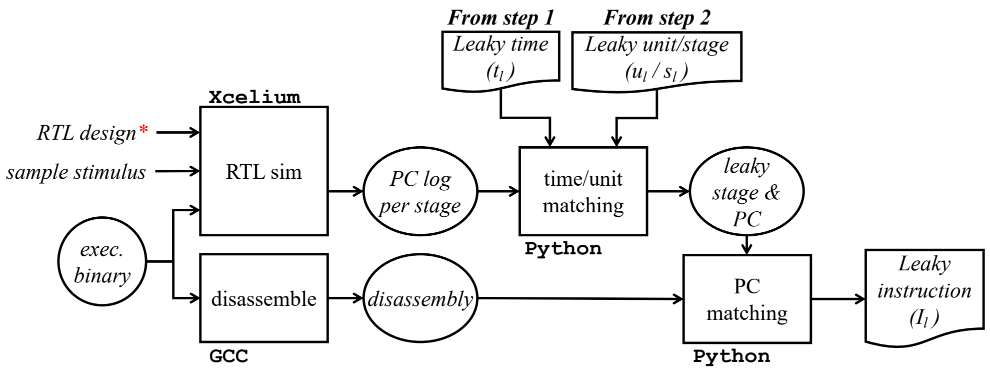


Figure 7: Layout of step 3 in RootCanal. The RTL design may need to be modified to pass on the PC signal to all pipeline registers. The executable binary is generated in the same way as in step 1.

the programmed software.

With a log of instructions executed by clock cycle, it is straightforward to map the leaky time t_l (from step 1) and unit/stage u_l/s_l (from step 2) and find the leaky PC. By disassembling the software binary file and looking up the leaky PC, the leaky instruction (I_l) is identified. When the detected u_l is outside the processor core, the instruction in the memory stage is flagged as the leaky instruction.

4 Experimental Results

This section demonstrates RootCanal's capabilities using practical examples of pre-silicon side-channel assessment. RootCanal uses simulated power traces at the gate level. Therefore, RootCanal supports the side-channel leakage assessment of SoCs, including evaluating (first-order) masking and hiding countermeasures at the level of software, RTL design, or gate level. We highlight the ability of RootCanal to back-annotate the source of leakage from the gate level to higher-level source code where a designer can interpret and act upon it. The following four examples illustrate RootCanal's capabilities in pre-silicon root-cause analysis of side-channel leakage. We start with analyzing the interaction between embedded software and a cryptographic hardware accelerator. Next, we demonstrate the impact of complementary data encoding on the hiding properties of software. Finally, we present two cases where RootCanal finds first-order flaws in masked software. Using RootCanal, we establish that these masking flaws originate not from programming errors but side-effects in the underlying hardware and compiler infrastructure.

The experiments study the power side-channel leakage in an SoC[2]. The SoC holds a RISC-V processor, an AES-128 hardware accelerator, and a collection of peripherals including DMA, UART and GPIO (Figure 3). The RISC-V processor has five pipeline stages: instruction fetch, instruction decode and register access, execution, memory access, and write-back. Using Cadence Genus, we synthesize this design with SkyWater 130nm standard cell library for 50MHz frequency. RootCanal can be used with any standard cell library for which the individual cell's power and timing characteristics are available (Liberty format). The SkyWater library is open source and therefore represents a low threshold for access. Table 2 shows the details for synthesis. The data and instruction memory blocks are modeled in the testbench and are not included in synthesis. The post-synthesis netlist is used for all of the following experiments.

[2]Source codes, design files, scripts, and results for all experiments are available at `https://github.com/Secure-Embedded-Systems/rootcanal-ches2022`.

Table 2: Synthesis details for RISCV-SoC using Cadence Genus

Standard Cell Library	Frequency	Sequential Cells	Logic Cells	Total Cells
SkyWater 130nm	50MHz	8155	20742	29872

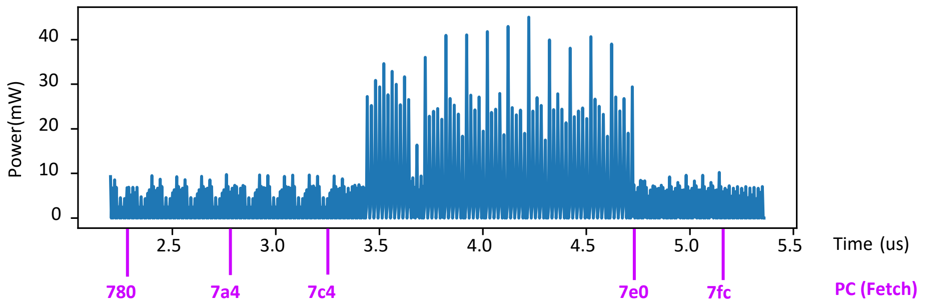


Figure 8: (Example 1) Average simulated power trace for SoC programming and running the AES hardware accelerator. The bottom X-scale links the power trace to Listing 1 through the value of the program counter (Fetch stage).

4.1 Example 1: Value-based Leakage in a System-on-Chip

The first example demonstrates the straightforward case of value-based leakage. The example shows how a designer can study the side-channel impact of a secret value moving around in the SoC architecture.

4.1.1 Setup

In this testbench, the RISC-V core reads a secret key and plaintext from memory, transfers these as 32-bit words to the AES accelerator, starts the accelerator, waits for its completion, and finally moves the ciphertext from the AES accelerator to memory. We simulate power consumption traces of this testbench given two groups of inputs. In the first group, we fix the key to all zeros while feeding 512 random plaintexts. In the second group, we set the key to all ones while providing 512 random plaintexts. By calculating Welch's t-test between the two groups of traces, we identify the leaky time samples caused by a difference in key-value.

Listing 1 shows the corresponding assembly snippet generated with the RISC-V GCC (10.2.0) compiler with `-O1` optimization flag. RootCanal obtains power traces for the 1024 test vectors using Cadence Joules (one frame per clock cycle). Figure 8 shows the average simulated power trace for the activity shown in Listing 1. RootCanal uses the power traces to determine the leaky instructions and the leaky hardware modules in the design by following Architecture Correlation Analysis and Netlist Graph Analysis.

4.1.2 Results

As expected for an unprotected implementation, there are numerous leaky time intervals with extremely high t-values as high as 5.10^9.

Leakage from Software Listing 1 summarizes the leakage sources identified by RootCanal from the perspective of the software, i.e., based on leakage from cells inside the processor core. The primary source of side-channel leakage stems from instructions that load the key from memory (instr. `780, 788, 790, 798`), namely in the memory stage by accessing RAM (`FDEMW`) and in the write-back stage by writing into the processor register file (`FDEMW`). Additional leakage stems from writing the key values to the AES hardware accelerator (instr. `784, 78c, 794, 79c`), namely from the decode (`FDEMW`), execute (`FDEMW`), and memory stages of these instructions. Indeed, each instruction reads a secret-key part from the register file in the decode stage, moves it through the execute stage to the memory stage, and finally writes it to the AES coprocessor register.

```
00000740 <main>:
 ...
 780:  lw    a3,0(a4)      # FDEMW # load key word 0 from RAM
 784:  sw    a3,4(a5)      # FDEMW # send key word 0 to AES
 788:  lw    a3,4(a4)      # FDEMW # load key word 1 from RAM
 78c:  sw    a3,8(a5)      # FDEMW # send key word 1 to AES
 790:  lw    a3,8(a4)      # FDEMW # load key word 2 from RAM
 794:  sw    a3,12(a5)     # FDEMW # send key word 2 to AES
 798:  lw    a4,12(a4)     # FDEMW # load key word 3 from RAM
 79c:  sw    a4,16(a5)     # FDEMW # send key word 3 to AES
 7a0:  addi  a4,zero,24    # FDEMW
 7a4:  lw    a3,0(a4)      # FDEMW # load pt word 0 from RAM
 7a8:  sw    a3,20(a5)     # FDEMW # send pt word 0 to AES
 7ac:  lw    a3,4(a4)      # FDEMW # load pt word 1 from RAM
 7b0:  sw    a3,24(a5)     # FDEMW # send pt word 1 to AES
 7b4:  lw    a3,8(a4)      # FDEMW # load pt word 2 from RAM
 7b8:  sw    a3,28(a5)     # FDEMW # send pt word 2 to AES
 7bc:  lw    a4,12(a4)     # FDEMW # load pt word 3 from RAM
 7c0:  sw    a4,32(a5)     # FDEMW # send pt word 3 to AES
 7c4:  addi  a4,zero,6     # FDEMW
 7c8:  sw    a4,0(a5)      # FDEMW # assert start signal
 7cc:  addi  a4,zero,4     # FDEMW
 7d0:  sw    a4,0(a5)      # FDEMW # deassert start signal
 7d4:  addi  a3,zero,1     # FDEMW
 7d8:  lw    a4,68(a5)     # FDEMW # read AES status signal
 7dc:  bne   a4,a3,7d8     # FDEMW # if AES not done loop back
 7e0:  lw    a4,52(a5)     # FDEMW # read ct word 0 from AES
 7e4:  sw    a4,40(zero)   # FDEMW # store ct word 0 to RAM
 7e8:  lw    a4,56(a5)     # FDEMW # read ct word 1 from AES
 7ec:  sw    a4,44(zero)   # FDEMW # store ct word 1 to RAM
 7f0:  lw    a4,60(a5)     # FDEMW # read ct word 2 from AES
 7f4:  sw    a4,48(zero)   # FDEMW # store ct word 2 to RAM
 7f8:  lw    a5,64(a5)     # FDEMW # read ct word 3 from AES
 7fc:  sw    a5,52(zero)   # FDEMW # store ct word 3 to RAM
...
```

Listing 1: (Example 1) Assembly code of the firmware for AES accelerator.
Blue letters indicate leaky pipeline stages.

RootCanal flags two additional leaky instructions immediately following the key loading (instr. `7a4, 7a8`). These leakages are from different pipeline stages but map to the same time sample. They are both caused by overwriting a storage buffer in the memory stage, which causes transitional leakage from the key-value (remaining from instr. `79c`) to the first plain-text word. The output of this buffer is forwarded to the decode stage (operand forwarding), resulting in the leakage in the decode stage of the next instruction `7a8`.

Leakage from Hardware In addition to the instructions listed above, RootCanal also flags the following hardware modules as leaky:

1. While reading the key from memory, gates from the bus structure show leakage.
2. When the processor writes the secret key to the AES accelerator, the bus interconnections and interfaces of the SoC modules attached to the bus (DMA, UART, and AES) generate side-channel leakage.
3. When AES is running, gates in the AES core create leakage.

This first example on analysis of known leakage serves as a sanity check for the methodology. Thanks to the automated back-annotation, RootCanal reduces the manual overhead in the analysis of side-channel leakage.

4.2 Example 2: Testing Bit-Sliced Data Encoding in Software Hiding

The second example demonstrates how RootCanal helps evaluate redundancy encoding schemes. Redundancy schemes are popular as fault-detection technique but the redundancy

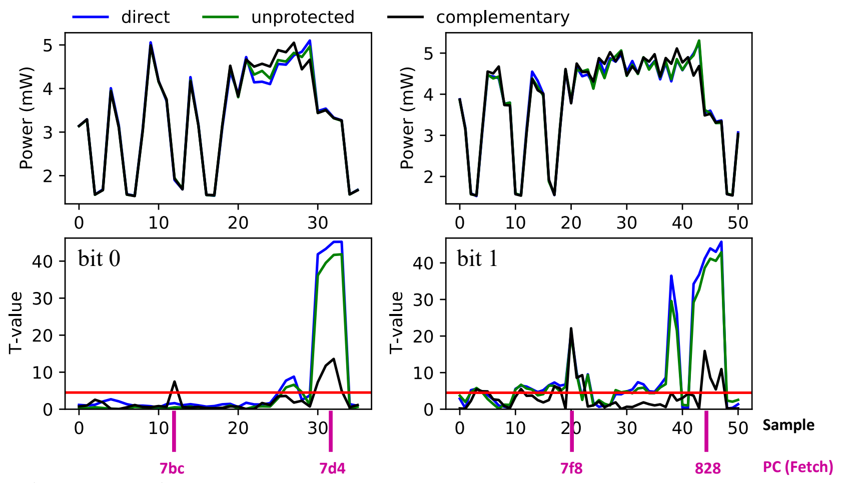


Figure 9: (Example 2) Average simulated power traces and TVLA results for redundant encoding schemes on bit-sliced PRESENT SBox

```
                          # Unprot. Red.    Red.
                          #         Direct  Complementary
                          # FDEMW   FDEMW   FDEMW
...
7bc: lw      a4,8(a1)     # FDEMW   FDEMW   FDEMW
7c0: xori    t5,a4,-1     # FDEMW   FDEMW   FDEMW
7c4: xori    t1,t4,-1     # FDEMW   FDEMW   FDEMW
7c8: and     t1,t1,a4     # FDEMW   FDEMW   FDEMW
7cc: xor     a4,a3,a5     # FDEMW   FDEMW   FDEMW
7d0: xori    t1,t1,-1     # FDEMW   FDEMW   FDEMW
7d4: xor     t1,t1,a4     # FDEMW   FDEMW   FDEMW
7d8: sw      t1,0(a0)     # FDEMW   FDEMW   FDEMW
...
```

Listing 2: (Example 2) Assembly code of the leaky parts of bit-sliced PRESENT SBox calculating bit 0 of output. Blue letters indicate leaky pipeline stages.

itself may be a source of side-channel leakage. Using RootCanal, a designer can compare alternate encodings.

4.2.1 Setup

We use RootCanal to compare the side-channel impact of two fault encoding schemes presented in SKIVA [KMD+21]. SKIVA proposed two bit-sliced redundancy schemes to detect faults. The redundant copy can be either a direct copy or an inverted version of the reference slice. The rationale for the latter scheme, complementary redundancy, is that it creates a hiding effect that may lead to lower side-channel leakage than direct redundancy. Bit-sliced software computations use bit-wise instructions. To compute complementary redundant bit-slices and perform fault checking, SKIVA introduces new instructions. We integrated these instructions into our RISC-V SoC.

We compare three bit-sliced implementations of the PRESENT SBox [BKL+07]. They use no redundancy (32 parallel runs), direct redundancy (16 parallel runs), and complementary redundancy (16 parallel runs), respectively. The input data is random but replicated appropriately according to each implementation's selected redundancy scheme (no redundancy, direct redundancy, and complementary redundancy). We feed the same 1024 random inputs to each scheme and simulate power traces. Figure 9 shows the average simulated traces over all inputs.

```
                         # Unprot. Red.     Red.
                         #         Direct   Complementary
                         # FDEMW   FDEMW    FDEMW
...
7f4:  lw    t1,12(a1) # FDEMW    FDEMW    FDEMW
7f8:  xori  t1,t1,-1  # FDEMW    FDEMW    FDEMW
7fc:  xori  t2,t2,-1  # FDEMW    FDEMW    FDEMW
800:  and   t1,t1,t2  # FDEMW    FDEMW    FDEMW
...
810:  and   a4,a4,t3  # FDEMW    FDEMW    FDEMW
814:  and   t0,a5,t0  # FDEMW    FDEMW    FDEMW
818:  xori  a4,a4,-1  # FDEMW    FDEMW    FDEMW
81c:  and   a4,a4,t0  # FDEMW    FDEMW    FDEMW
820:  xori  t1,t1,-1  # FDEMW    FDEMW    FDEMW
824:  xori  a4,a4,-1  # FDEMW    FDEMW    FDEMW
828:  and   a4,t1,a4  # FDEMW    FDEMW    FDEMW
82c:  sw    a4,4(a0)  # FDEMW    FDEMW    FDEMW
...
```

Listing 3: (Example 2) Assembly code of the leaky parts of bit-sliced PRESENT SBox calculating bit 1 of output. Blue letters indicate leaky pipeline stages.

4.2.2 Results

We use the node bias test on a single output bit of the PRESENT SBox, which will split the 1024 test traces into two groups of roughly equal size. The bit-sliced computation evaluates 32 or 16 SBoxes in parallel, depending on the redundancy level. Hence, rather than deciding the group on a single output bit, we use a majority vote over the corresponding output bit of all parallel SBoxes. We do the same experiment for bit 0 and bit 1 of the four-bit output of PRESENT SBox.

As shown in Figure 9, the maximum t-value (at sample number 33 and 47 for output bit 0 and 1 respectively) is the highest for direct redundancy and the lowest for complementary redundancy. Furthermore, the leaky frame count for output bit 0 (resp. output bit 1) is 6, 7, and 5 (resp. 29, 32, and 13) for unprotected, direct redundancy, and complementary redundancy schemes. Thus, the side-channel leakage level degrades when using direct redundancy and improves when using complementary redundancy.

Leakage from Software Listing 2 and Listing 3 show the parts of the assembly codes that cause leakage and the leaky stages detected for each instruction by RootCanal in each redundancy scheme for calculating bit 0 and bit 1 of the PRESENT SBox output. The direct redundancy scheme degrades the side-channel leakage over the unprotected scheme because more pipeline stages and instructions are affected. The complementary redundancy scheme reduces the side-channel leakage but does not eliminate it. As an imperfect hiding-based countermeasure, this is an expected result.

4.3 Example 3: Debugging Masking – across HW/SW Boundaries

The third example describes the analysis of a masking flaw across the boundaries of hardware and software. Using RootCanal we identified the cause and were able to formulate a potential solution.

4.3.1 Setup

We analyze an open-source byte-masked software implementation[3] of AES [YYP+18] that was previously shown to suffer from transition-based leakage [SSB+19, GMPO20]. We analyze the power side-channel leakage of the initial `AddRoundKey` and the first round `SubBytes`. Listing 4 shows the assembly code for the body of the loops in `addRoundKey_masked` and

[3]https://github.com/Secure-Embedded-Systems/Masked-AES-Implementation/tree/master/Byte-Masked-AES

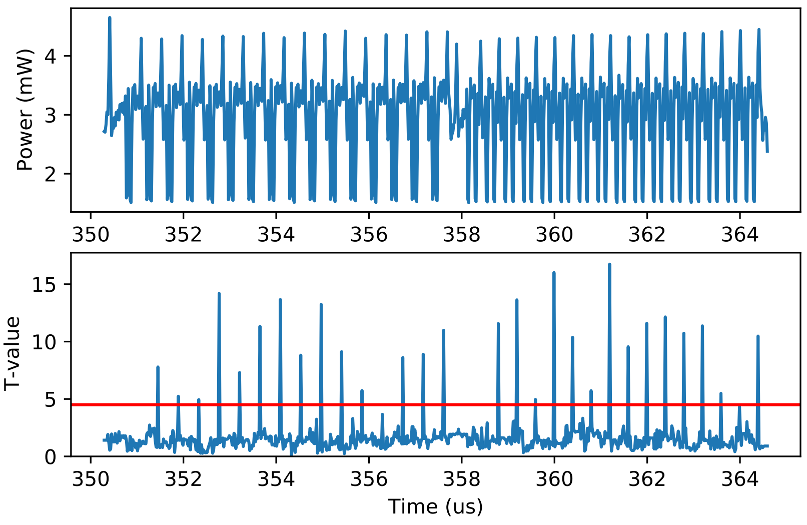


Figure 10: (Example 3) Average simulated power trace and TVLA result for the byte-masked AES example

`subBytes_masked` functions (RISC-V GCC (10.2.0) with `-O1` optimization flag). To evaluate this masked implementation, we use the non-specific fixed vs. random TVLA on 1024 random inputs and 1024 fixed inputs. The testbench reseeds PRNG with different and random initial states for each power simulation to ensure that different masks are used for different power traces.

4.3.2 Results

We perform the fixed vs. random TVLA test three times, each time with a new randomly chosen fixed value. Figure 10 shows the average power trace with the result of the TVLA test, taken as the maximum t-value for each sample among the three tests.

Leakage from Software RootCanal marks instruction `9dc` from the `addRoundKey_masked` function and instruction `ca0` from the `subBytes_masked` function as the origin of side-channel leakage. In both cases, leakage happens during the memory stage of the instruction, pointing at leakage from the bus connection to the RAM. The `sb` (store byte) instructions in consecutive iterations of both of the loops overwrite the previous state byte. For instance, in the first iteration of the loop, the `sb` instruction at `9dc` writes V_0 to RAM.

$$\begin{aligned}V_0 &= (state[0] \oplus Mask[6]) \oplus RoundKey\_masked[0][0] \\ &= (state[0] \oplus Mask[6]) \oplus (RoundKey[0][0] \oplus Mask[6] \oplus Mask[4]) \\ &= state[0] \oplus RoundKey[0][0] \oplus Mask[4]\end{aligned}$$

In the next iteration of the loop, the `sb` instruction at `9dc` writes V_1 to RAM.

$$\begin{aligned}V_1 &= (state[1] \oplus Mask[7]) \oplus RoundKey\_masked[0][1] \\ &= (state[1] \oplus Mask[7]) \oplus (RoundKey[0][1] \oplus Mask[7] \oplus Mask[4]) \\ &= state[1] \oplus RoundKey[0][1] \oplus Mask[4]\end{aligned}$$

The TVLA analysis hints at transitional leakage from this memory write operation. Indeed, this transitional leakage is proportional to the distance from V_0 to V_1 which is dependent on the plain (unmasked) values of two state and RoundKey bytes.

$$\begin{aligned}V_0 \oplus V_1 &= (state[0] \oplus RoundKey[0][0] \oplus Mask[4]) \\ &\oplus (state[1] \oplus RoundKey[0][1] \oplus Mask[4]) \\ &= state[0] \oplus RoundKey[0][0] \oplus state[1] \oplus RoundKey[0][1]\end{aligned}$$

```
000009b4 <addRoundKey_masked>:
 ...
 9cc:  lbu     a2,0(a4)    # FDEMW # load state byte
 9d0:  lbu     a5,0(a3)    # FDEMW # load RoundKey_masked byte
 9d4:  xor     a5,a5,a2    # FDEMW # state ^ RoundKey_masked
 9d8:  andi    a5,a5,255   # FDEMW # (state ^ RoundKey_masked) & 0xff
 9dc:  sb      a5,0(a4)    # FDEMW # update state
 9e0:  addi    a4,a4,1     # FDEMW
 ...

00000c84 <subBytes_masked>:
 ...
 c90:  lbu     a5,0(a0)    # FDEMW # load state
 c94:  andi    a5,a5,255   # FDEMW # state & 0xff
 c98:  add     a5,a4,a5    # FDEMW
 c9c:  lbu     a5,352(a5)  # FDEMW # load Sbox_masked
 ca0:  sb      a5,0(a0)    # FDEMW # update state
 ca4:  addi    a0,a0,1     # FDEMW
 ca8:  bne     a0,a3,c90   # FDEMW # if not done loop back
 ...
```

Listing 4: (Example 3) Assembly code of the byte-masked AES.
Blue letters indicate leaky pipeline stages.

In a similar fashion, instruction `ca0` in consecutive loop iterations causes transitional leakage. For instance, during the last iteration of the loop, two consecutive masked SBox values S_0 and S_1 are stored in memory, causing transitional leakage proportional to $S_0 \oplus S_1$.

$$\begin{aligned} S_0 &= sbox[state[14] \oplus RoundKey[0][14]] \oplus Mask[5] \\ S_1 &= sbox[state[15] \oplus RoundKey[0][15]] \oplus Mask[5] \\ S_0 \oplus S_1 &= sbox[state[14] \oplus RoundKey[0][14]] \oplus sbox[state[15] \oplus RoundKey[0][15]] \end{aligned}$$

Leakage from Hardware RootCanal can also explain *why* this transitional leakage resulting in unmasking occurs. To store data to the memory, the processor will first store the data in an interface register as part of the bus access protocol. The contents of this register will then move to the net connected to the data input port of the memory. Figure 11 shows a simplified diagram of the bus interface circuit. RootCanal flags the components shown in red as the root cause of side-channel leakage. These components include the `interface_d_mem_data_in_reg` register as well as the multiplexer (implemented as `sky130_fd_sc_hd__a22o_1` gate from SkyWater 130nm standard cell library). Leakage from this part of the circuit is present when instruction `9dc` (resp. `ca0`) is in the memory stage of the processor pipeline during the execution of `addRoundKey_masked` (resp. `subBytes_masked`) function.

To avoid the aforementioned leakages, the contents of the `interface_d_mem_data_in_reg` register should be cleared (zeroized or randomized [GMPP20]) between consecutive iterations of the loops in both functions. A simple software approach, for example, is to insert dummy store operations at the end of each iteration, sending random data to this register.

4.4 Example 4: Debugging Masking – When The Compiler Trips Up

The final example demonstrates a masking flaw introduced by compiler optimization. RootCanal identifies the location of the leakage in software, and through inspection we were able to explain its cause.

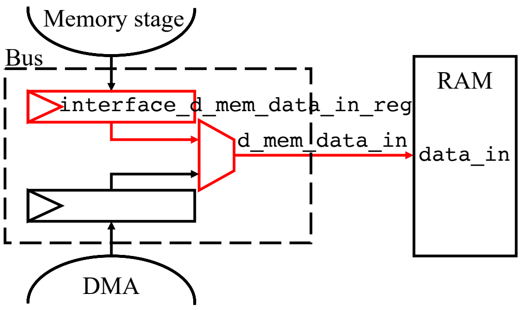


Figure 11: Leaking circuit in byte-masked software AES

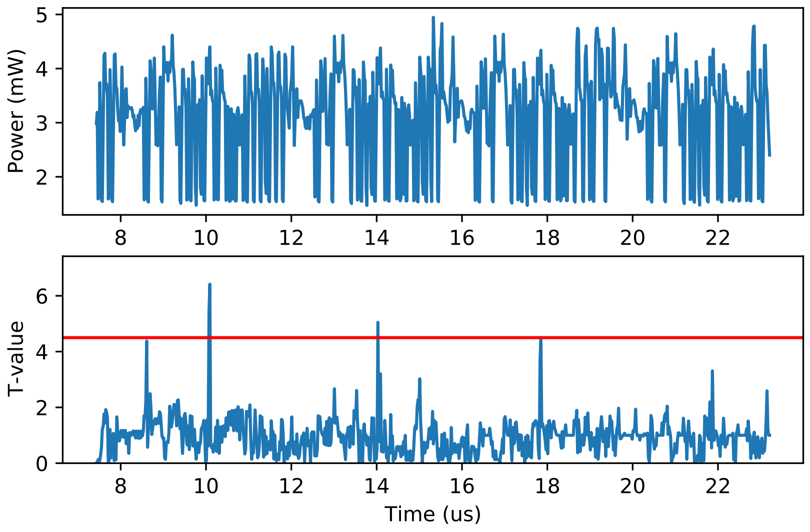


Figure 12: (Example 4) Average simulated power trace and TVLA result for bit-sliced masked PRESENT SBox

4.4.1 Setup

In this experiment, we analyze the power side-channel leakage of RISC-V SoC while running bit-sliced masked implementation of the SBox used in the PRESENT cipher. We generate the masked bit-sliced PRESENT SBox using the `usuba` compiler following the instructions from Tornado [BDM+20]. The non-linear operations in the generated code use the masked multiplication introduced by Ishai *et al.* [ISW03] (`isw_mult()` shown in Listing 5 and Listing 6 compiled by RISC-V GCC and `-O1` flag). As our leakage test, we use the non-specific fixed vs. random TVLA on 1024 random and 1024 fixed inputs.

4.4.2 Results

Figure 12 shows the average of the simulated power traces and the TVLA result.

Leakage from Software RootCanal flags instructions `160, 164, 168` from the `isw_mult` function as causes of leakage. Leakage from instruction `160` has a micro-architectural cause. The ALU result in the processor is switching from `op2[1] & op1[0]` (instr. `154`) to `op2[0] & op1[1]` (instr. `160`). Even though different registers are used in these two instructions (leakage not expected at ISA level), being two consecutive ALU instructions, their intermediate results collide in the pipeline registers.

```
 1 static void isw_mult(uint32_t *res,const uint32_t *op1,const uint32_t *op2)
       {
 2   int i,j;
 3   uint32_t rnd;
 4 
 5   for (i=0; i<MASKING_ORDER; i++) {
 6     res[i] = 0;
 7   }
 8 
 9   for (i=0; i<MASKING_ORDER; i++) {
10     res[i] ^= op1[i] & op2[i];
11 
12     for (j=i+1; j<MASKING_ORDER; j++) {
13       rnd = get_random();
14       res[i] ^= rnd;
15       res[j] ^= (rnd ^ (op1[i] & op2[j])) ^ (op1[j] & op2[i]);
16     }
17   }
18 }
```

Listing 5: (Example 4) ISW multiplication used in non-linear operations in the bit-sliced masked PRESENT SBox

```
 1  ...
 2  140:  sw    a5,0(s0)    # FDEMW # res[0] = (op1[0] & op2[0]) ^ rnd
 3  144:  lw    a5,4(s2)    # FDEMW # a5 = op2[1]
 4  148:  lw    a4,0(s1)    # FDEMW # a4 = op1[0]
 5  14c:  and   a5,a5,a4    # FDEMW # a5 = op2[1] & op1[0]
 6  150:  lw    a4,4(s0)    # FDEMW # a4 = res[1] (a4 = 0)
 7  154:  xor   a5,a5,a4    # FDEMW # a5 = (op2[1] & op1[0]) ^ 0
 8  158:  lw    a4,0(s2)    # FDEMW # a4 = op2[0]
 9  15c:  lw    a3,4(s1)    # FDEMW # a3 = op1[1]
10  160:  and   a4,a4,a3    # FDEMW # a4 = op2[0] & op1[1]
11  164:  xor   a5,a5,a4    # FDEMW # a5 = (op2[1] & op1[0]) ^ (op2[0] & op1[1])
12  168:  xor   a5,a5,a0    # FDEMW # a5 = a5 ^ rnd
13  16c:  sw    a5,4(s0)    # FDEMW # res[1] = a5
14  170:  lw    a4,4(s2)    # FDEMW # a4 = op2[1]
15  174:  lw    a3,4(s1)    # FDEMW # a3 = op1[1]
16  178:  and   a4,a4,a3    # FDEMW # a4 = op2[1] & op1[1]
17  17c:  xor   a5,a4,a5    # FDEMW
18  180:  sw    a5,4(s0)    # FDEMW # res[1] = a5
19  ...
20 
```

Listing 6: (Example 4) Assembly code of the bit-sliced masked PRESENT SBox. Blue letters indicate leaky pipeline stages.

The order of operations in the ISW multiplication gadget between the C code and the compiler-generated assembly reveals the reason for the observed leakage from instructions `164, 168`. Line 15 in the source code of `isw_mult` first refreshes the partial product (`op1[i] & op2[j]`) and only after this randomization, combines it with the other partial product (`op1[j] & op2[i]`). However, the compiler has changed the order of this combination as reordering consecutive `xor` operations is functionally correct due to `xor`'s associative property (line 11 in Listing 6). The multiplication operands now depend on different shares of the same variable, which creates side-channel leakage.

4.5 Analysis of Results

A major advantage of RootCanal is that it provides a systematic mechanism to present the outcome of side-channel assessment in a format that is easier to understand for a designer. In the pre-silicon white-box environment, the objective is not only to confirm the presence of side-channel leakage, but also to explain it. Table 3 illustrates the data reduction we achieved for each design example. The design complexity of our RISC-V SoC

Table 3: Summary of leakage observed in examples

Example	Leaky Gates	Leaky Frames	Leaky Instructions
Example 1	9665	558	10
Example 2 - unprot (bit 0 / bit 1)	978/814	6/29	5/10
Example 2 - direct (bit 0 / bit 1)	1733/3157	7/32	5/11
Example 2 - compl (bit 0 / bit 1)	1717/2712	5/13	3/5
Example 3	68	28	2
Example 4	2706	3	3

Table 4: Execution time of RootCanal steps for each example

Example	Synthesis	Simulation	Power Sim	ACA	Back Annotation	Total
Example 1	20m 42 s	2h 14m 31s	9h 10m 6s	14m 17s	22m 55s	12h 22m 31s
Example 2	20m 13 s	13h 25m 45s	19h 57m 17s	10m 24s	8m 46s	1d 10h 2m 25s
Example 3	20m 1s	21h 24m 27s	2d 7h 24m 57s	17m 45s	15s	3d 5h 27m 25s
Example 4	22m	4h 42m 09s	22h 35m 35s	6m 5s	3m 3s	1d 3h 48m 52s

*Xeon Gold 6248 CPU @ 2.50GHz, 384G Workstation

is 29,872 cells overall. In each of the examples, we are able to reduce a large collection of leaky gates to only a handful of processor RISC-V instructions. The leakage assessment results for the examples depend on every component of the technology stack, including compiler, micro-architecture, and standard-cell library. Hence, changing any component of the stack may affect the results. In all our design examples, we found that 1K test vectors is sufficient to produce clear conclusions on a non-specific test. The relatively low number of traces is explained by the noiseless simulation, and the limited design complexity (below 100K gates).

A pre-silicon technique brings up the important question of design tool performance. We measure the execution time for each step involved in RootCanal by example in Table 4. The synthesis step is common among the examples. The simulation step and power simulation step complexity depend on the size of the netlist and on the length of the testbench. The complexity of ACA depends on the size of the netlist, the number of samples in power traces, and the number of leaky samples. The complexity of the back-annotation step depends on the number of leaky gates and on the size of the netlist. There are multiple knobs available to reduce the power simulation time. First, the power simulation is embarrassingly parallel over the input vectors. Second, with additional designer input, the time window and the design size can be decreased to a specific region of input, at the risk of possibly missing a source of leakage by human error. Third, commercial power simulation tools are in our experience not yet optimized for side-channel assessment, leading to large stimuli and result file sizes. Hence, power simulation techniques could be tuned. Finally, one could reduce the accuracy of the simulation and use for example toggle counts instead of gate-level power models. This last option has limited advantage because it reduces the capability of RootCanal to identify leakage sources.

5 Conclusion

Design automation is a crucial ingredient to scaling up successful design techniques for a large community of designers. The advent of pre-silicon side-channel leakage assessment tools may mean significant cost savings for new designs. But these savings can only be realized when the output of the tools is accessible to the broader hardware design community. RootCanal demonstrates the feasibility of automatically determining the cause of side-channel leakage at an abstraction level accessible to a designer. Like a source-level software debugger that enables a programmer to debug software source code instead of machine instructions, RootCanal aims to be a source-level side-channel leakage debugger. Future improvements to RootCanal include improving the accuracy of power simulation with cross-coupling effects, extending the toolbox of the non-specific tests, and extending the methodology for super scalar architectures.

Acknowledgements This research was supported in part by the National Science Foundation (Award 1931639) and by the DARPA SCATE project.

References

[ASA$^+$21] Leonid Azriel, Julian Speith, Nils Albartus, Ran Ginosar, Avi Mendelson, and Christof Paar. A survey of algorithmic methods in IC reverse engineering. *J. Cryptogr. Eng.*, 11(3):299–315, 2021.

[BBC$^+$19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskverif: Automated verification of higher-order masking in presence of physical defaults. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019.

[BBYS21] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. Sok: Design tools for side-channel-aware implementions. *CoRR*, abs/2104.08593, 2021.

[BDM$^+$20] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. Tornado: Automatic generation of probing-secure masked bitsliced implementations. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 311–341. Springer, 2020.

[BGI$^+$18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 321–353. Springer, 2018.

[BIBB21] Omid Bazangani, Alexandre Iooss, Ileana Buhan, and Lejla Batina. ABBY: automating the creation of fine-grained leakage models. *IACR Cryptol. ePrint Arch.*, page 1569, 2021.

[BKL$^+$07] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. Present: An ultra-lightweight block cipher. In *International workshop on cryptographic hardware and embedded systems*, pages 450–466. Springer, 2007.

[CGD18] Yann Le Corre, Johann Großschädl, and Daniel Dinu. Micro-architectural power simulator for leakage assessment of cryptographic software on ARM cortex-m3 processors. In Junfeng Fan and Benedikt Gierlichs, editors, *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, volume 10815 of *Lecture Notes in Computer Science*, pages 82–98. Springer, 2018.

[CHS09] Zhimin Chen, Syed Haider, and Patrick Schaumont. Side-channel leakage in masked circuits caused by higher-order circuit effects. In Jong Hyuk Park, Hsiao-Hwa Chen, Mohammed Atiquzzaman, Changhoon Lee, Tai-Hoon Kim, and Sang-Soo Yeo, editors, *Advances in Information Security and Assurance, Third International Conference and Workshops, ISA 2009, Seoul, Korea, June 25-27, 2009. Proceedings*, volume 5576 of *Lecture Notes in Computer Science*, pages 327–336. Springer, 2009.

[GHP$^+$21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco:{Co-Design} and {Co-Verification} of masked software implementations on {CPUs}. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1469–1468, 2021.

[GMPO20] Si Gao, Ben Marshall, Dan Page, and Elisabeth Oswald. Share-slicing: Friend or foe? *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(1):152–174, 2020.

[GMPP20] Si Gao, Ben Marshall, Dan Page, and Thinh Pham. Fenl: an ise to mitigate analogue micro-architectural leakage. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 73–98, 2020.

[GO21] Si Gao and Elisabeth Oswald. A novel completeness test and its application to side channel attacks and simulators. *IACR Cryptol. ePrint Arch.*, page 756, 2021.

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. Secure and efficient software masking on superscalar pipelined processors. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 3–32. Springer, 2021.

[HPN$^+$19] Miao Tony He, Jungmin Park, Adib Nahiyan, Apostol Vassilev, Yier Jin, and Mark M. Tehranipoor. RTL-PSC: automated power side-channel leakage assessment at register-transfer level. In *37th IEEE VLSI Test Symposium, VTS 2019, Monterey, CA, USA, April 23-25, 2019*, pages 1–6. IEEE, 2019.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *Annual International Cryptology Conference*, pages 463–481. Springer, 2003.

[KMD$^+$21] Pantea Kiaei, Darius Mercadier, Pierre-Evariste Dagand, Karine Heydemann, and Patrick Schaumont. Custom instruction support for modular defense against side-channel and fault attacks. In Guido Marco Bertoni and Francesco Regazzoni, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 221–253, Cham, 2021. Springer International Publishing.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, 2020.

[KYL$^+$22] Pantea Kiaei, Yuan Yao, Zhenyuan Liu, Nicole Fern, Cees-Bart Breunesse, Jasper Van Woudenberg, Kate Gillis, Alex Dich, Peter Grossmann, and Patrick Schaumont. Gate-level side-channel leakage assessment with architecture correlation analysis, 2022. https://arxiv.org/abs/2204.11972.

[MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 199–216. USENIX Association, 2017.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-channel leakage of masked CMOS gates. In Alfred Menezes, editor, *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, volume 3376 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2005.

[MPW22] Ben Marshall, Dan Page, and James Webb. MIRACLE: micro-architectural leakage evaluation A study of micro-architectural power leakage across many devices. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):175–220, 2022.

[SM15] Tobias Schneider and Amir Moradi. Leakage assessment methodology. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, pages 495–513, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[SSB+19] Madura A Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. *arXiv preprint arXiv:1912.05183*, 2019.

[SVRK19] Patanjali SLPSK, Prasanna Karthik Vairam, Chester Rebeiro, and V. Kamakoti. Karna: A gate-sizing based security aware EDA flow for improved power side-channel attack protection. In David Z. Pan, editor, *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*, pages 1–8. ACM, 2019.

[YKES20] Yuan Yao, Tarun Kathuria, Baris Ege, and Patrick Schaumont. Architecture correlation analysis (ACA): identifying the source of side-channel leakage at gate-level. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020*, pages 188–196. IEEE, 2020.

[YYP+18] Yuan Yao, Mo Yang, Conor Patrick, Bilgiday Yuce, and Patrick Schaumont. Fault-assisted side-channel analysis of masked implementations. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 57–64. IEEE, 2018.