EPiC Series in Computing

Volume 87, 2022, Pages 18–33

Proceedings of 10th International Workshop on Security Proofs for Embedded Systems

Simulation Based Evaluation of Bit-Interaction Side-Channel Leakage on RISC-V Processor

Tamon Asano[1] and Takeshi Sugawara[1]

The University of Electro-Communications, Chofu, Tokyo, Japan
{t.asano, sugawara}@uec.ac.jp

Abstract

Masking is a promising countermeasure against side-channel attack, and share slicing is its efficient software implementation that stores all the shares in a single register to exploit the parallelism of Boolean instructions. However, the security of share slicing relies on the assumption of bit-independent leakage from those instructions. Gao et al. recently discovered a violation causing a security degradation, called the bit-interaction leakage, by experimentally evaluating ARM processors. However, its causality remained open because of the blackbox inside the target processors. In this paper, we approach this problem with simulation-based side-channel leakage evaluation using a RISC-V processor. More specifically, we use Western Digital's open-source SweRV EH1 core as a target platform and measure its side-channel traces by running logic simulation and counting the number of signal transitions in the synthesized ALU netlist. We successfully replicate the bit-interaction leakage from a shifter using the simulated traces. By exploiting the flexibility of simulation-based analysis, we positively verify Gao et al.'s hypothesis on how the shifter causes the leakage. Moreover, we discover a new bit-interaction leakage from an arithmetic adder caused by carry propagation. Finally, we discuss hardware and software countermeasures against the bit-interaction leakage.

1 Introduction

There is a growing demand for extending the IT systems to the physical world by using network-enabled embedded devices, and there is a growing demand for efficient and secure implementation of cryptography. There are many use cases of operating such embedded devices in a hostile environment wherein the device owners are potential attackers. Given the physical access, such attackers obtain information leakage via physical side-channels such as execution time, power consumption, and electromagnetic radiation to attack cryptography. Such side-channel attack (SCA) is a realistic threat to secure embedded devices [21], and researchers have been studying new attacks and defenses for more than two decades since the Kocher et al.'s discovery of simple and differential power analyses [12, 14].

Masking is arguably the most well-studied countermeasure against SCA, which redundantly encodes a sensitive intermediate variable into a set of variables called *shares* [4]. In a $(d+1)$-share Boolean masking, for instance, we encode a target variable x into its shared representation

$\overline{x} = (x_0, x_1, \cdots, x_d)$ satisfying $x = \bigoplus_{i=0}^{d} x_i$. Then, we perform the target cryptographic operation while maintaining the shared representation, which efficiently randomize the computation and eliminate the correlation between the sensitive intermediate variable and side-channel leakage. A large computational overhead is the major drawback of masking; the computational cost grows quadratically with the number of shares $(d + 1)$, which is the security parameter. The overhead is particularly serious for resource-constrained devices, and efficient masking implementation is an important research subject.

Barthe et al.'s share slicing is a technique for efficiently implementing a Boolean masking in software [2]. The incompatibility between the bit-wise masking and the word-wise processing in CPUs is one reason of inefficiency, and share slicing addresses this issue by concatenating the shares into a word, e.g., $x_0||x_1||\cdots||x_d$, and storing it into a general-purpose register. By using the CPU's Boolean instructions with this register, we can efficiently process these bits in parallel, in the same way as the conventional bit-slicing technique [15, 3]. Since we put several shares in the same register, the interaction between the bits in the register immediately causes an exploitable side-channel leakage; the bit-wise independence is the prerequisite for the security of share slicing, which looks reasonable at first glance considering the CPU's instructions being bit-wise.

Gao et al. rigorously studied the prerequisite behind share slicing by experimentally evaluating ARM processors and discovered a violation, called the bit-interaction leakage, which causes a security degradation in a masking scheme [8]. They observed the bit-interaction leakage only with particular shift instructions, and hypothesized that a barrel-shifter circuit is the source of the leakage. However, the verification of the hypothesis remained open because the black-box nature of the target ARM processor prevented us from understanding its internals.

We study the bit-interaction leakage with simulation-based leakage evaluation using an open-source RISC-V processor. More specifically, we choose Western Digital's SweRV EH1 Core [5] as a target, and evaluate its side-channel leakage by counting the number of signal transitions (i.e., toggles) in a logic simulator using a synthesized netlist [22]. The target processor's source code in SystemVerilog is publicly available [5], which enable the research community to publicly discuss and modify the circuit structure. We study the causality of bit-interaction leakage by the following techniques enabled the above simulation environment. First, we pinpoint the leakage source by simulating side-channel leakage for each component. Second, we modify the target processor to observe how it changes the side-channel leakage.

1.1 Contributions

Our contributions are fourfold. We first make a simulation-based evaluation environment and replicate the bit-interaction leakage. Then, we pinpoint the source with finer-grained simulation, followed by the experiments verifying why these components cause bit-interaction leakages. Finally, we discuss software and hardware countermeasures. Below we briefly summarize the contributions and how they correspond to the paper organization.

Leakage Simulator based on Open-Source RISC-V Processor (Section 3) Our simulation environment enables us simulating side-channel leakage from an open-source RISC-V implementation (SweRV EH1) by counting the number of signal transitions in a logic simulator. We successfully replicate the bit-interaction leakage using simulated side-channel traces.

Causality of Bit-Interaction Leakage from Shifters (Section 3 and Section 4) We narrow down the leakage source into the shifter circuits in the arithmetic logic unit (ALU). We

further verify the Gao et al.'s hypothesis that signal transition in its control signal being the cause of leakage.

Yet Another Bit-Interaction Leakage from an Arithmetic Adder (Section 3 and Section 4) Our simulation identifies that an arithmetic adder in ALU also causes a bit-interaction leakage through carry-propagation paths.

Proposing Countermeasure Against Bit-Interaction Leakage (Section 5) We discuss possible countermeasures to eliminate the bit-interaction leakage. The target processor generates the bit-interaction leakage even with Boolean (cf. add/shift) instructions because of the particular ALU structure that runs all the combinatorial operations in parallel and selects one later. Addressing the problem, we discuss a new ALU circuit that stops the adder and shifter circuits during irrelevant instructions thereby eliminating the bit-interaction leakage while executing Boolean instructions. Moreover, we discuss a software countermeasure to eliminate the bit-interaction leakage from the shift instructions. The method dispatches several dummy instructions in advance, which prevents the signal transition and the bit-interaction leakage.

2 Preliminary

We briefly recall the bit-interaction leakage and the target processor. Masking is well-studied countermeasure against side-channel attack, and share-slicing is its efficient software implementation. The bit-interaction leakage is a certain type of side-channel leakage that can defeat share slicing. SweRV EH1 is an open-source implementation of the RISC-V instruction set.

2.1 Masking and Share-slicing

Masking is arguably the most well-studied countermeasure against SCA. It encodes a sensitive intermediate variable x into a set of shares namely $\overline{x}$ [4]. One common realization is Boolean (i.e., additive) masking, in which the sum of all the shares represents the original variable. More specifically, with $(d+1)$-share masking, we have

$$\overline{x} = (x_0, x_1, \cdots, x_d) \quad \text{s.t.} \quad x = \bigoplus_{i=0}^{d} x_i. \tag{1}$$

Realizing the entire cryptographic operation in the form of shares is the core idea of masking. Given a target function $f : x \mapsto y$, we can construct a map between the shares namely $\overline{f} : \overline{x} \mapsto \overline{y}$, which realizes the original function while preserving the shared representation of the input and output. We can achieve the construction of $\overline{f}$ by decomposing f with basic operations (such as AND and XOR etc.) and then replacing each operation with its corresponding masked gadget. The shared representation efficiently randomize the computation and eliminate the correlation between the sensitive intermediate variable and side-channel leakage.

A large computational overhead is the major drawback of masking. The number of shares $(d+1)$ corresponds to the security parameter d [11], the number of probes the adversary has access to, and we should choose sufficiently large d. Computational cost, memory space, and randomness grow with the number of shares $(d+1)$. In particular, the computational cost in circuit complexity grows quadratically with the number of shares because of the additional effort needed to realize a non-linear operation (i.e., AND) in the shared representation [11].

This is a serious issue especially for embedded devices with limited computational resources. Consequently, researchers have been studying efficient masking and its implementation.

Bit-slicing is a common technique for efficient software implementation, which decomposes the target algorithm into bit operations, and execute them in parallel by using Boolean instructions in a CPU [15, 3]. Efficient software implementation of masking using the bit-slicing technique is an active research field [10, 2]. In particular, share slicing is first introduced by Barthe et al. [2] and then elaborated by Gao et al. [8]. The particular way of slicing characterizes share-slicing; share-slicing packs all the shared bits of x into a word, e.g., $x_0||x_1||\cdots||x_d$, and stores into a general-purpose register.

2.2 Bit-Interaction Leakage and its Causality

Gao et al. rigorously studied the security of share-slicing and discovered *the bit-interaction leakage* which causes a security degradation. Since share-slicing puts several shares in the same register, the interaction between the bits in the register immediately causes an exploitable side-channel leakage. Gao et al. experimentally evaluated the ARM Cortex M0 and M3 processors and discovered that such a bit-interaction leakage exists and is measurable, and causes a security degradation in a masking scheme [8].

Based on the observation that bit-interaction leakage occurs only with certain shift instructions, Gao et al. hypothesized that a transient phenomenon in a barrel shifter causes the leakage. A barrel shifter, shown in Figure 1, is a circuit for realizing variable shifts and is composed of a series of 2^n-bit shifters and selectors. For simplicity, we focus on the first selector for the 32-bit input x and its 1-bit shift denoted by $x \ll 1$. When the control signal shamt[0] arrives later than x, there is a signal transition from x to $x \ll 1$ at the selector output, causing a side-channel leakage depends on two neighboring bits, i.e., a bit-interaction leakage. In summary, the signal transition in the `shamt` causes the bit-interaction leakage in the barrel shifter. Verification of the hypothesis remained open because the black-box nature of the target ARM processor prevented us from understanding its internals.

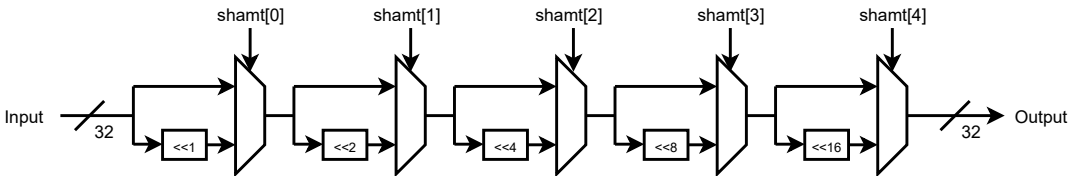

Figure 1: A typical 32-bit barrel shifter for left-logical shift composed of a series of constant shifts and selectors

2.3 RISC-V and SweRV EH1 Core

RISC-V [20] is an open instruction-set architecture, attracting much attentions from the semiconductor industry as a free and open alternative to ARM and other micro-architectures. There are several open-source implementations, including Rocket Chip [1], SonicBOOM [23], and SweRV EH1 Core [5], which makes RISC-V a desirable platform for studying side-channel leakage from processors.

SweRV EH1 Core is a RISC-V implementation developed by Western Digital, of which source code publicly available with Apache License 2.0 [5]. SweRV is written in synthesiz-

able SystemVerilog (cf. Rocket Chip written in Chisel), which makes its integration to the conventional digital-design flow easier.

We briefly review the details about the SweRV EH1's ALU in Figure 2, which is essential for understanding the bit-interaction leakage. There are three inputs: `A_IN` and `B_IN` are 32-bit input operands, and `ALU_codes` is a control signal deciding which instruction to execute. There are pipeline registers that update their values the beginning of a clock cycle. Upon the register's update, the new data in the registers propagate to the combinatorial logic represented as the boxes labeled with `Comb_bitwise`, `Comb_shifters`[1], and `Comb_adder`. These combinatorial circuits run in parallel independently to the instruction being executed. Finally, the AND-OR tree at the end, controlled by the `ALU_codes` signal selects a particular output depending on the instruction. We believe that this is a textbook implementation of ALU [19].

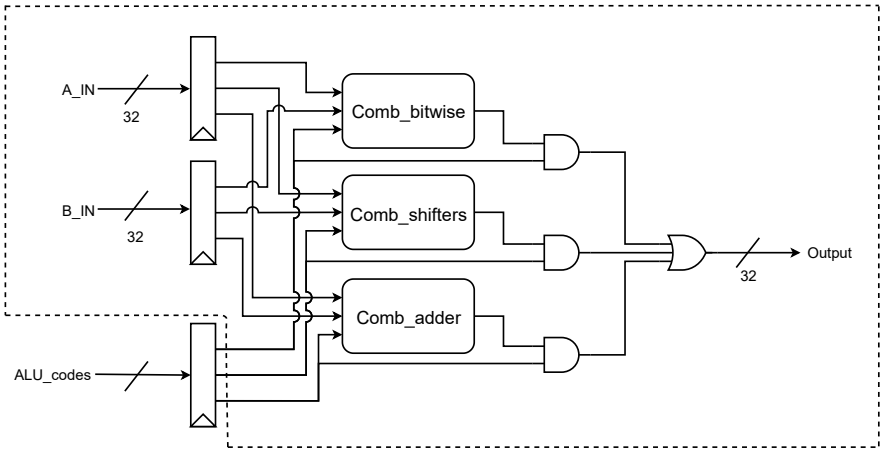

Figure 2: SweRV EH1's ALU that runs all the combinatorial circuits in parallel, and selects one output later. `Comb_bitwise` is the combinatorial circuit for Boolean instructions. `Comb_shifters` is the combinatorial circuit for various shift instructions. `Comb_adder` is the arithmetic adder. `ALU_codes` control outputs of each combinatorial circuit. The border indicates the range of the ALU module written in SystemVerilog.

3 Leakage Simulation Environment using Open-Source Processor

In this section, we describe our simulation environment and replicate the bit-interaction leakage using simulated side-channel traces. Finally, we pinpoint the sub-components inside ALU causing the bit-interaction leakage.

3.1 Simulation Method

Simulation based side-channel evaluation is indispensable for verifying the security before fabricating a chip, and there are several simulation-based evaluations of side-channel attack in

[1]The `Comb_shifters` submodule consists of the left-logical, right-logical, and right-arithmetic shifts expressed with SystemVerilog's operators, i.e., `<<`, `>>`, and `>>>`.

different levels e.g., functional, gate-level, SPICE, and electromagnetic simulations. Among them, we choose gate-level simulation [22] which can simulate the Gao et al.'s hypothesis on the bit-interaction leakage (see Section 2.2) at low computational cost.

In gate-level simulation, we simulates side-channel traces by the number of signal transitions in logic simulation using a netlist i.e., a circuit composed of standard cells usually generated by a logic synthesizer. It can simulate transient phenomena, also known as glitches, known to cause security degradation in masking countermeasures [17]. Gate-level simulation is cheaper than finer-grained alternatives such as SPICE simulation, and we can easily integrate it to the standard design flow because logic engineers who design cryptographic circuits are usually familiar with logic simulators. We can choose either pre- and post-P&R (place and route) simulation. The latter can provide more accurate and realistic delay information at the cost of additional effort of making P&R. In this paper, however, we choose pre-P&R simulation by prioritizing shorter turn-around time in experiments.

Limitation As a downside, however, gate-level simulation only considers gate-level phenomena. More specifically, it does not simulate the electrical coupling effect [6, 13] which is hypothesized as yet another cause of the bit-interaction leakage. The gate-level simulation is sufficient for detecting the bit-interaction leakages in the gate level, as we will see in the later sections, however, missing bit-interaction leakage in gate-level simulation does not promise its absence in a real chip. Verification of the bit-interaction leakage by the coupling effect is beyond the scope of this paper.

3.2 Simulation Environment

SweRV Configuration We instantiate the SweRV EH1 [5] processor with its default configuration parameters except for the following two options. First, we set the number of ALUs to two (cf. four) to simplify the analysis. Second, we enable the clock gating which we believe a natural choice considering an ASIC implementation.

Logic Synthesis We focus on the bit-interaction leakage from ALU and simulate the leakage exclusively from the ALU. We first generate a netlist by synthesizing the ALU's source code (`design/exu/exu_alu_ctl.sv` in the SweRV EH1's source code [5]) using NanGate 45-nm standard cell library [18] and Synopsys Design Compiler. For the simulation efficiency, we keep the non-ALU components intact at the RTL-level description. After the synthesis, we make functional verification by replacing the original ALU code with the synthesized netlist and running a logic simulation using Cadence Incisive to check if a test succeeds. During the simulation, we back-annotate the delay information in SDF and run the logic simulation at the 1-picosecond precision.

Extending Logic Simulator for Counting Toggles We simulate side-channel leakage from the ALU by counting the number of signal transitions at the standard cells composing the synthesized netlist. We realize it by using Verilog Procedural Interface (VPI) [7, 16], which enables us to assign a callback function called upon the target signal's change. More specifically, we design a callback function that counts the number of $0 \to 1$ output transitions (i.e., rising edges) in each clock cycle to simulate current consumption at a V_{DD} pin [22]. We assign the callback function to all the standard cells in the ALU's netlist, which enable us to measure the ALU's total toggle counts representing side-channel leakage. For narrowing down the leakage source in the later experiments, we separate the ALU into smaller submodules, corresponding

to `Comb_adder`, `Comb_shifters` and `Comb_bitwise` in Figure 2, without changing the original logical structure before the synthesis. Then, we count the number of toggles for each of the submodules representing `Comb_bitwise`, `Comb_shifters` and `Comb_adder`.

3.3 Experimental Validation

We verify our simulation environment by replicating the bit-interaction leakage using simulated side-channel traces.

Target Software Snippet Figure 3 shows the code snippet the processor executes during the simulation. Figure 3-(a) to -(c) correspond to three instructions we examine namely the logical-left shift (`slli`), xor (`xori`), and add (`addi`) instructions[2]. The target takes the sensitive value in the general-purpose register `t3`, processes it with the immediate 1, and stores the result to another register `t4`.

The remaining lines are experimental instrumentation. In particular, we use `nop` instructions in the 12th and 17th lines, which is the shortcut of `addi x0, x0, 0`, in order to initialize the ALU into a known state.

(a) `slli`

```
1  // t0: loop count
2  // t1: loop var i
3  // t2: array address
4  loop:
5      // t3 = *(array)
6      lw   t3, 0(t2)
7      // array++
8      addi t2, t2, 4
9      // i++
10     addi t1, t1, 1
11     // nops
12     addi x0, x0, 0
13     ...
14     // t4 = t3 << 1
15     slli t4, t3, 1
16     // nops
17     addi x0, x0, 0
18     ...
19     // back to loop
20     bne t0, t1, loop
```

(b) `xori`

```
1  // t0: loop count
2  // t1: loop var i
3  // t2: array address
4  loop:
5      // t3 = *(array)
6      lw   t3, 0(t2)
7      // array++
8      addi t2, t2, 4
9      // i++
10     addi t1, t1, 1
11     // nops
12     addi x0, x0, 0
13     ...
14     // t4 = t3 ^ 1
15     xori t4, t3, 1
16     // nops
17     addi x0, x0, 0
18     ...
19     // back to loop
20     bne t0, t1, loop
```

(c) `addi`

```
1  // t0: loop count
2  // t1: loop var i
3  // t2: array address
4  loop:
5      // t3 = *(array)
6      lw   t3, 0(t2)
7      // array++
8      addi t2, t2, 4
9      // i++
10     addi t1, t1, 1
11     // nops
12     addi x0, x0, 0
13     ...
14     // t4 = t3 + 1
15     addi t4, t3, 1
16     // nops
17     addi x0, x0, 0
18     ...
19     // back to loop
20     bne t0, t1, loop
```

Figure 3: The assembly code used for the simulation. The sub-figures (a)–(c) differ in the target instructions in the line #15. We load the test vectors in the general-purpose register `t3`.

Test Vector for Leakage Assessment We conduct the test vector leakage assessment (TVLA) [9] to evaluate the bit-interaction leakage. By following the Gao et al.'s work [8], we use a pair of test vectors namely

[2]We use the immediate (I-type) instructions for simplicity. The leakage from the register (R-type) instructions should be the same because they use the same ALU.

$$\begin{cases} \mathsf{TV}_{\mathsf{random}} := \{\underbrace{r_{31}||r_{30}||\cdots||r_4||r_3||r_2||r_1||r_0}_{32bit} \quad | \quad r_i \in \{0,1\}\} \\ \mathsf{TV}_{\mathsf{share0s}} := \{\underbrace{s_{15}||s_{14}||\cdots||s_4||s_3||s_2||s_1||s_0}_{32bit} \quad | \quad s_i \in \{00,11\}\} \end{cases} \tag{2}$$

We feed these values to the target instructions through the general-purpose register `t3`. $\mathsf{TV}_{\mathsf{random}}$ consists of 32-bit random values. Meanwhile, the 32-bit values in $\mathsf{TV}_{\mathsf{share0}}$ are composed of either 00 or 11 representing all zeroes in share slicing with a 2-share masking. The expected Hamming weights of both $\mathsf{TV}_{\mathsf{random}}$ and $\mathsf{TV}_{\mathsf{share0}}$ is 16, and their side-channel leakage should be indistinguishable as far as the bits in the register are independent. Finally, we compare the two sets of traces, corresponding to $\mathsf{TV}_{\mathsf{random}}$ and $\mathsf{TV}_{\mathsf{share0}}$, with Welch's t-test. Rejection in the t-test implies the presence of the bit-interaction leakage.

We use 20,000 test vectors for each of $\mathsf{TV}_{\mathsf{random}}$ and $\mathsf{TV}_{\mathsf{share0}}$. We repeat the measurements for the three target instructions, i.e., `slli`, `xori`, and `addi`. For the later experiments, we measure the toggles in different resolutions: `Comb_shifters`, `Comb_adder`, `Comb_bitwise`, and the entire ALU.

3.4 Replicating the Bit-Interaction Leakage

Figure 4-(a) to -(c) show the results with the `slli`, `xori`, and `addi` instructions. In each graph, the vertical and horizontal axes represent the t-statistic and the number of side-channel traces, respectively. Each subfigure has 3–4 traces corresponding to the resolution of the measurement[3]. The red horizontal lines indicate ±4.5, which are the typical thresholds for rejecting the null hypothesis.

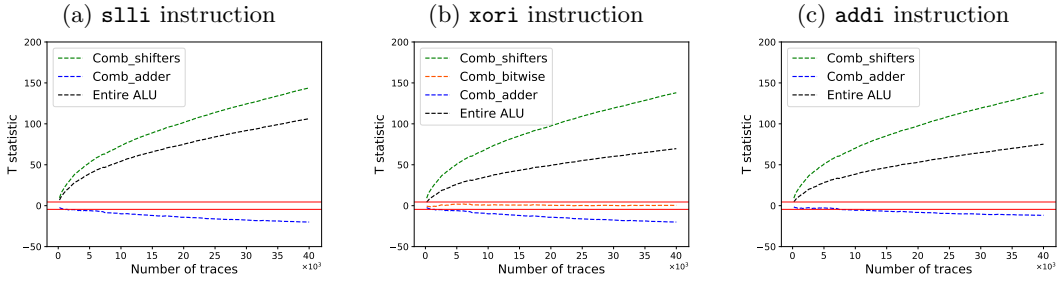

Figure 4: Bit-interaction leakage from the SweRV EH1 core's ALU. The sub-figures (a)–(c) correspond to the `slli`, `xori`, and `addi` instructions. Each graph shows the t-statistic and the number of side-channel traces. Each graph contains 3 or 4 traces corresponding to the side-channel traces from `Comb_shifters`, `Comb_adder`, `Comb_bitwise`, and the entire ALU.

We observe t-statistics exceeding the thresholds with all the instructions, successfully replicating the bit-interaction leakage. In particular, we always observe the bit-interaction leakage from `Comb_shifters` and `Comb_adder`. Meanwhile, the t-statistic fit within the $(-4.5, 4.5)$ range while measuring `Comb_bitwise` with the `xori` instruction. We can interpret the results as follows: `Comb_shifters` and `Comb_adder` cause the bit-interaction leakage. `Comb_bitwise` is

[3]We omit the `Comb_bitwise` in Figure 4-(a) and (c) because we observe no switching activity in the target component, and thus the corresponding t-statistics are meaningless.

leak free, but the entire ALU shows the bit-interaction leakage even with the `xori` instruction because the ALU circuit in Figure 2 activates `Comb_shifters` and `Comb_adder`.

4 Causality of Bit-Interaction Leakage

In this section, we study the mechanism behind the bit-interaction leakage from `Comb_shifters` and `Comb_adder` observed in the previous experiment.

4.1 Barrel shifter

The bit-interaction leakage from `Comb_shifters` confirms with the Gao et al.'s hypothesis that a barrel shifter causes the bit-interaction leakage. As discussed in Section 2.2, Gao et al. further hypothesised that signal transition in the selector signal is the problem, which we will verify in the next experiment.

In order to verify the hypothesis, we modify the shifter in Figure 1 to stop the signal transition in the control signal. More specifically, we insert buffers in front of the shifter input so that the `shamt` signal always arrives earlier, which efficiently removes the phenomenon Gao et al. hypothesized as the cause.

In order to simplify the experiment, we focus on an independent barrel shifter instead of the EH1 core. More specifically, we implement a 32-bit logical-left shifter using the `<<` operator[4] in the same way as the SweRV EH1 core. We also implement a modified shifter which is the same way except for inserting buffers in front of the input signal. Then, we synthesize the corresponding netlists and simulate them in the same ways as the previous experiment. The amount of left shift is always set as 1 to represents the target in the previous experiment, i.e., `slli t4, t3, 1`. Before each measurement, we initialize all the inputs with zeros to set the circuit into a known state.

For this experiment, we evaluate the bit-interaction leakage between any pair of the bits in the 32-bit input, in contrast to the previous experiments which considered the interaction between adjacent bits only. In other words, we conduct the following measurements to find the specific pair of bits which causes the bit-interaction leakage. We realize the evaluation by measuring the target shifter with random 32-bit values and then split them into two groups afterwards. More specifically, we denote $\mathcal{S} \subset \{0,1\}^{32}$ be the input set. For a 32-bit shifter input $r \in \mathcal{S}$, we denote its i-th bit as $[r]_i$. For evaluating the bit-interaction leakage between i-th and j-th bits, we split $\mathcal{S}$ into the following two groups:

$$\begin{cases} \mathcal{S}_1^{(i,j)} := \{r \in \mathcal{S} \quad | \quad [r]_i \oplus [r]_j = 1\} \\ \mathcal{S}_0^{(i,j)} := \{r \in \mathcal{S} \quad | \quad [r]_i \oplus [r]_j = 0\} \end{cases} \tag{3}$$

Finally, we run a t-test comparing the side-channel traces corresponding to $\mathcal{S}_1^{(i,j)}$ and $\mathcal{S}_0^{(i,j)}$. Rejection in the test implies a bit-interaction leakage between the i-th and j-th bits. We measure 50,000 traces, i.e., $|\mathcal{S}| = 50,000$, and conduct t-tests for any pair of i and j.

Figure 5-(a) and -(b) show the experimental results for the two shifters before and after inserting the input buffer. Each sub-figure shows t-statistic for any (i, j) combination with a heatmap. More specifically, the horizontal and vertical axes represent the indices of the target bits j and i, and the color represents the t-statistic for the given pair. We show the results only for $i < j$ because of the symmetry.

[4]We also conduct same experiment with `>>` and `>>>` (arithmetic right shift) operator and confirm that the results are similar to the result of `<<` operator.

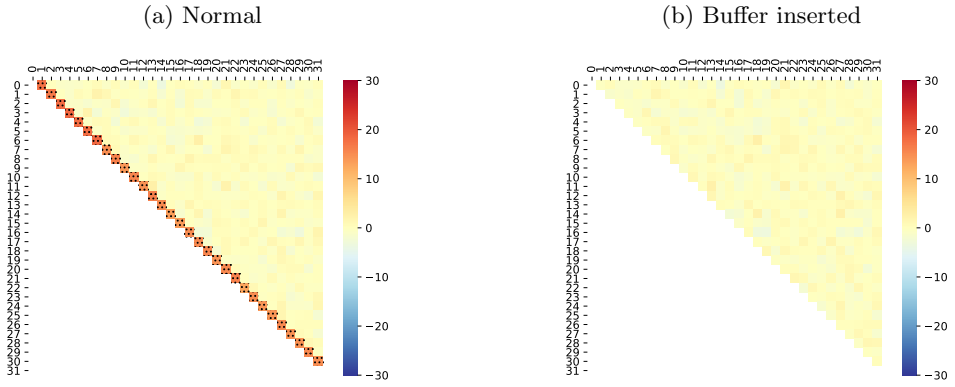

Figure 5: The heatmaps showing the bit-interaction leakage between i-th and j-th bits for 1-bit left shift in barrel shifters. The sub-figure (a) and (b) correspond to the two shifters before and after inserting the input buffer. The horizontal and vertical axes represent the indices of the target bits j and i, and the color represents the t-statistic. The t-statistics of dotted cells are outside the range of (-4.5, 4.5).

With the shifter without modification, we observe t-statistics beyond the threshold with $j = i + 1$, showing a slash line in Figure 5-(a). This result indicates that the bit-interaction leakage occurs only between the neighboring bits because the shift amount is 1. This bit-interaction leakage disappears in Figure 5-(b), confirming that the insertion of input buffer eliminate the bit-interaction leakage. As a result, we can conclude that Gao et al.'s hypothesis is correct, and signal transition in `shamt` signals causes the bit-interaction leakage in the barrel shifter.

4.2 Adder

In section 3, we observed the bit-interaction leakage source from the arithmetic adder `Comb_adder`, which has not been known in the previous study [8]. We study its causality in the same way as the previous section and show that carry propagation is the cause of the problem.

4.2.1 Experiment #1

First, we check the interaction between bits by repeating the previous experiment with an arithmetic adder. More specifically, we implement and synthesize a 32-bit adder using the `+` operator in Verilog, which results in netlist of a simple ripple carry adder [5](See Figure 6). We feed the random 32-bit input $r \in \mathcal{S}$ to the adder's first operand, while the second operand is always 1, i.e., the adder always increment the first operand. Before each add operation, we let the adder perform $0 + 0$ to initialize it into a known state.

We analyze the simulated traces in the same as Section 4.1 to evaluate the interaction between i-th and j-th bits in the first operand $r \in \mathcal{S}$. Figure 7 shows that the t-statistics for

[5]We confirm that the arithmetic adder of the SweRV EH1's ALU is also written by `+` operator and synthesized into a simple ripple carry adder.

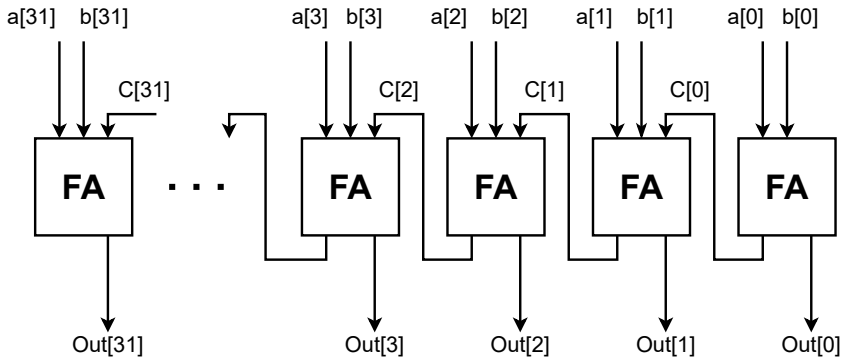

Figure 6: A 32-bit ripple carry adder. `FA` is a full-adder component. `a` and `b` is input, and `Out` is output. `C` is carry.

different (i, j) pairs in heatmap. Unlike the shifter, only lower bits exhibit the bit-interaction leakage. More specifically, we observe t-statistics exceeding the -4.5 threshold for

$$(i, j) \in \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\}. \tag{4}$$

Moreover, the t-statistic becomes smaller in the higher bits. The above result strongly suggests that carry propagation is the cause of the bit-interaction leakage; the smaller probability of the carry propagation reaching to the higher bits explains the less interaction in the higher bits.

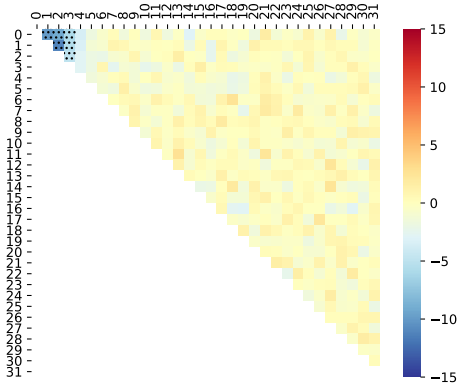

Figure 7: The heatmaps showing the bit-interaction leakage between i-th and j-th bits while performing +1 in the ripple-carry adder

4.2.2 Experiment #2

For further verifying the hypothesis with the second experiment using carefully designed test vectors that stops the carry propagation by fixing the LSB to 0. We feed these test vectors as the first operand of the adder while fixing the second operand to 1. More specifically, we use

the following test vectors:

$$\begin{cases} \mathsf{TV}^0_{\mathsf{carry1}} := \{r \in \{0,1\}^{32} \mid [r]_2 \oplus [r]_1 = 0 \text{ and } [r]_0 = 1\} \\ \mathsf{TV}^1_{\mathsf{carry1}} := \{r \in \{0,1\}^{32} \mid [r]_2 \oplus [r]_1 = 1 \text{ and } [r]_0 = 1\} \end{cases} \tag{5}$$

$$\begin{cases} \mathsf{TV}^0_{\mathsf{carry0}} := \{r \in \{0,1\}^{32} \mid [r]_2 \oplus [r]_1 = 0 \text{ and } [r]_0 = 0\} \\ \mathsf{TV}^1_{\mathsf{carry0}} := \{r \in \{0,1\}^{32} \mid [r]_2 \oplus [r]_1 = 1 \text{ and } [r]_0 = 0\} \end{cases} \tag{6}$$

For analysis, we compare the pairs ($\mathsf{TV}^0_{\mathsf{carry1}}$, $\mathsf{TV}^1_{\mathsf{carry1}}$) and ($\mathsf{TV}^0_{\mathsf{carry0}}$, $\mathsf{TV}^1_{\mathsf{carry0}}$), respectively. These pairs are designed to detect the bit-interaction leakage between the 1st and 2nd bits denoted as $[r]_1$ and $[r]_2$ wherein $r \in \{0,1\}^{32}$ is an element of the test vectors. Meanwhile, ($\mathsf{TV}^0_{\mathsf{carry1}}$, $\mathsf{TV}^1_{\mathsf{carry1}}$) and ($\mathsf{TV}^0_{\mathsf{carry0}}$, $\mathsf{TV}^1_{\mathsf{carry0}}$) are different in LSB, i.e., $[r]_0$. The LSB of ($\mathsf{TV}^0_{\mathsf{carry1}}$, $\mathsf{TV}^1_{\mathsf{carry1}}$) is fixed to 1, and thus the carry always propagates to the $[r]_1$. Meanwhile, with ($\mathsf{TV}^0_{\mathsf{carry0}}$, $\mathsf{TV}^1_{\mathsf{carry0}}$), $[r]_0$ is fixed to zero and thus carry propagation never reaches to higher bits. We prepare 50,000 test vectors of each group and measure toggles when adding 1.

Figure 8 shows the results of t-tests. The result comparing ($\mathsf{TV}^0_{\mathsf{carry1}}$, $\mathsf{TV}^1_{\mathsf{carry1}}$) is the baseline and in which t-statistics exceed the [-4.5, 4.5] range, confirming the presence of the bit-interaction leakage as predicted by the previous result in Figure 7. In contrast, the bit-interaction leakage disappears with the result comparing ($\mathsf{TV}^0_{\mathsf{carry0}}$, $\mathsf{TV}^1_{\mathsf{carry0}}$). These results verify that the bit-interaction leakage by adder is caused by propagating carry.

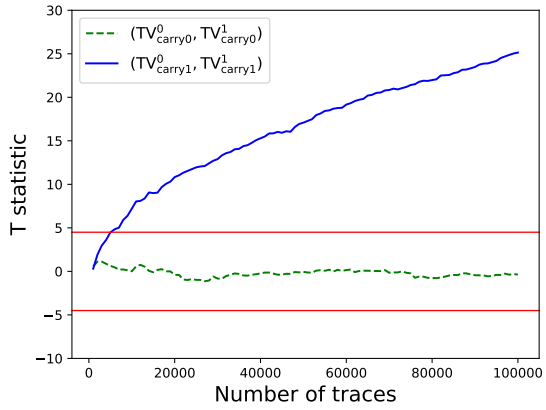

Figure 8: The bit-interaction leakage betwee the 1st and 2nd bits in the ripple-carry adder. The traces ($\mathsf{TV}^0_{\mathsf{carry1}}$, $\mathsf{TV}^1_{\mathsf{carry1}}$) and ($\mathsf{TV}^0_{\mathsf{carry0}}$, $\mathsf{TV}^1_{\mathsf{carry0}}$) represent to the results with and without the carry input from the 0th bit.

5 Discussion

Our results show that the bit-interaction leakage due to shifter and/or adder is present even in Boolean instructions. Considering the above two cases, we discuss countermeasures in two levels: (i) hardware countermeasure to eliminate the bit-interaction leakage from the adder and shifter circuits in executing Boolean instructions by stopping the adder and shifter circuits and (ii) software countermeasure to thwart the bit-interaction leakage from the shifter.

5.1 Hardware Countermeasure

We first discuss a hardware countermeasure to stop the adder and shifter circuits during irrelevant instructions thereby eliminating their bit-interaction leakage while executing Boolean instructions.

Figure 9-(a) and -(b) show the ALU circuit before and after the modification. Figure 9-(a) is the original ALU, similar to the one in Figure 2, which runs all the combinatorial circuits `Comb_bitwise`, `Comb_shifters`, and `Comb_adder` in parallel, and selects one output later. This parallel execution is harmless in logical level, but causes the bit-interaction leakage even with Boolean instructions. We can address the problem by stopping `Comb_shifters` and `Comb_adder` while executing Boolean instructions and can easily realize this by inserting registers in front of the combinatorial circuits as shown in Figure 9-(b).

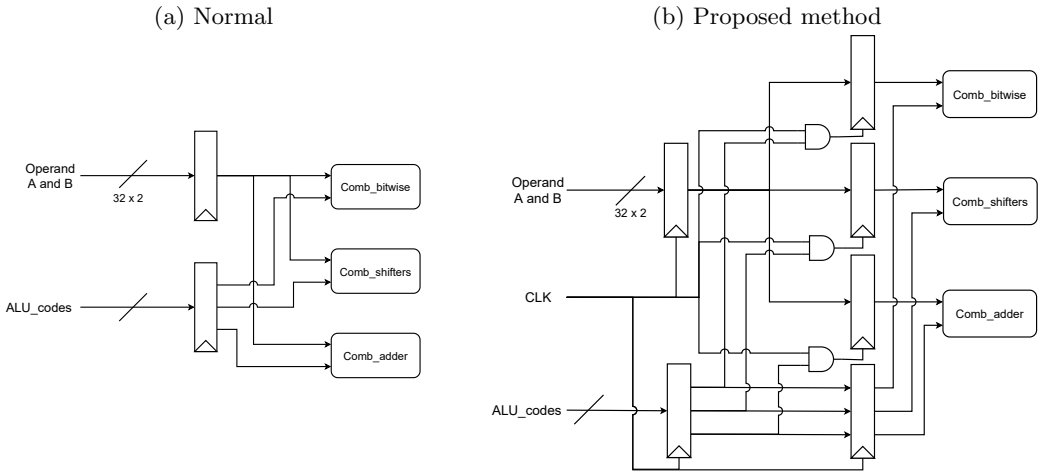

Figure 9: Hardware modification for isolating the adder/shifter leakage during Boolean instructions. (a) the original ALU before modification. (b) the modified ALU having registers in front of the combinatorial circuits

The main area overhead comes from the additional six 32-bit registers, which is negligible compared to the circuit area of the entire CPU. In addition, it increases the number of pipeline stages which can cause a performance overhead in the cycle per instruction. We can further optimize the ALU in Figure 9-(b) using negative-edge registers to keep the number of pipeline stages, but that is beyond the scope of this paper.

With the modified ALU in Figure 9-(b), we can eliminate the bit-interaction leakage from Boolean instructions because the ALU no longer feeds the sensitive operands to `Comb_shifters` and `Comb_adder`. This countermeasure is effective for Boolean instructions only, and the bit-interaction leakage is still present in the shift and add instructions. We address the leakage from the shift instructions in the next section, meanwhile we leave the bit-interaction leakage from add instructions unprotected because they are unused in share slicing.

5.2 Software Countermeasure

We discuss software countermeasure for eliminating the bit-interaction leakage from a barrel shifter caused by signal transition hypothesized by Gao et al.

As we verified in Section 4.1, the bit-interaction leakage is caused by the change in the control signal after the shifter input has arrived. We can avoid such transition in the control signal by dispatching several dummy instructions in advance thereby so that the control signal is properly precharged. This countermeasure prevents the bit-interaction leakage from the shift instructions. In particular, with the ARM Cortex M0/M3 processors that has the bit-interaction leakage on with the shift instructions [8], we can eliminate the bit-interaction leakage only with this software countermeasure.

Figure 10 shows an example code snippet with the countermeasure wherein the instructions shown in blue are the additional dummy instructions and the one in red is the target instruction handling sensitive data. More specifically, we replace the `nop` instruction (a shortcut of `addi x0, x0, 0`) in Figure 3-(a) with `slli x0, x0, 1`. Since the operand `x0` is always initialized with zero, this `slli x0, x0, 1` instruction makes no change in the general-purpose registers, and we can use it as yet another `nop` instruction. Meanwhile, the third operand `1` represent the amount of shifts which is directly connected to the barrel shifter's control signal (`shamt` in Figure 1). At the time of the target instruction at the 14th line, no signal transition occurs at the barrel shifter's control signal because it is already set by the preceding dummy instructions.

```
1  // t0: loop count
2  // t1: loop var i
3  // t2: array address
4  loop:
5      lw   t3, 0(t2)       // t3 = *(array)
6      addi t2, t2, 4       // array++
7      addi t1, t1, 1       // i++
8      addi x0, x0, 0       // nops
9      ...
10     slli x0, x0, 1       // alternative nops, input 1 in B operand
11     slli x0, x0, 1
12     slli x0, x0, 1
13     slli x0, x0, 1
14     slli t4, t3, 1       // target, t4 = t3 << 1
15     addi x0, x0, 0       // nops
16     ...
17     bne t0, t1, loop     // back to loop
```

Figure 10: An example assembly code sequence for the software countermeasure. The red instruction is the target instruction handling sensitive data. The blue instructions are the dummy instructions for countermeasure.

We can realize any shift with the constant offset by choosing the dummy instructions with the same shift amount as the target instruction. The number of dummy instructions determines the overhead. In this particular example, this countermeasure multiplies the cost of the shift instruction by $\times 4$ [6]. The overhead depends on the ratio of shift instructions occupied in the entire implementation. The evaluation through a case study of concrete masking implementation is beyond the scope of this paper, and opened for future research.

[6] We place several `slli x0, x0, 1` instructions to saturate the two ALUs in the EH1's dual-issue architecture. We will need additional care for the CPU with the out-of-order execution; it is not the case with the SweRV EH1 core.

6 Conclusion

We conducted a simulation-based side-channel evaluation of the SweRV EH1 core, open-source RISC-V implementation, and replicate the bit-interaction leakage in its ALU. We found that a shifter and an arithmetic adder are the sources of the leakage, and showed how these components cause the bit-interaction leakage. We also observed that the bit-interaction leakage is present even with Boolean instructions because of the particular ALU structure running all the combinatorial circuits in parallel. We finally discussed the hardware and software countermeasures for addressing the bit-interaction leakage: (i) an ALU modification to stop the bit-interaction leakage during the instructions irrelevant to the shifter and the adder, e.g., Boolean instructions, and (ii) a software countermeasure for eliminating the bit-interaction leakage from a shifter by using dummy instructions that pre-charges the ALU in advance.

There are several open questions. The other source of bit-interaction leakage is possible because we focused on ALU only and the TVLA-based methodology can cause false negatives. We plan to verify the proposed countermeasures through simulation and/or real experiments.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable and helpful comments. The study is supported by JSPS KAKENHI Grant Number JP18H05289. The CAD tools used in the paper are supported by VLSI Design and Education Center (VDEC), the University of Tokyo with the collaboration with CADENCE Corporation and SYNOPSYS Corporation.

References

[1] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[2] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In *Advances in Cryptology - EUROCRYPT 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 535–566, 2017.

[3] Ryad Benadjila, Jian Guo, Victor Lomné, and Thomas Peyrin. Implementing lightweight block ciphers on x86 architectures. In *Selected Areas in Cryptography - SAC 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 324–351. Springer, 2013.

[4] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Advances in Cryptology - CRYPTO '99, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[5] CHIPS Alliance. EH1 SweRV RISC-V Core™1.7 from Western Digital. https://github.com/chipsalliance/Cores-SweRV.

[6] Thomas De Cnudde, Begül Bilgin, Benedikt Gierlichs, Ventzislav Nikov, Svetla Nikova, and Vincent Rijmen. Does coupling affect the security of masked implementations? In *Constructive Side-Channel Analysis and Secure Design COSADE 2017, Revised Selected Papers*, volume 10348 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2017.

[7] C. Dawson, S.K. Pattanam, and D. Roberts. The verilog procedural interface for the verilog hardware description language. In *Proceedings. IEEE International Verilog HDL Conference*, pages 17–23, 1996.

[8] Si Gao, Ben Marshall, Dan Page, and Elisabeth Oswald. Share-slicing: Friend or foe? *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 152–174, 2020.

[9] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[10] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 567–597, 2017.

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

[12] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology - CRYPTO '99, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[13] Itamar Levi, Davide Bellizia, and François-Xavier Standaert. Reducing a masked implementation's effective security order with setup manipulations and an explanation based on externally-amplified couplings. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):293–317, 2019.

[14] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[15] Seiichi Matsuda and Shiho Moriai. Lightweight cryptography for the cloud: Exploit the power of bitslice implementation. In *Cryptographic Hardware and Embedded Systems - CHES 2012, Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 2012.

[16] S. Meyer. Verilog plus c language modeling with pli 2.0: The next generation simulation language. In *Proceedings International Verilog HDL Conference and VHDL International Users Forum*, pages 98–105, 1998.

[17] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. Glitch-resistant masking revisited or why proofs in the robust probing model are needed. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):256–292, 2019.

[18] NanGate. Nangate 45nm open cell library. https://si2.org/open-cell-library.

[19] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., 5th edition, 2013.

[20] RISC-V International. https://riscv.org.

[21] Thomas Roche, Victor Lomné, Camille Mutschler, and Laurent Imbert. A side journey to titan. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 231–248. USENIX Association, August 2021.

[22] Kris Tiri and Ingrid Verbauwhede. Simulation models for side-channel information leaks. In *Proceedings of the 42nd Design Automation Conference, DAC 2005*, pages 228–233. ACM, 2005.

[23] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. May 2020.