Cross-Layer Vulnerability Analysis of System-on-Chip against Physical Hardware Attacks

Pantea Kiaei

A Dissertation
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy
in
Electrical and Computer Engineering

July 2022

APPROVED:

_____

Professor Patrick Schaumont, Advisor, Worcester Polytechnic Institute

_____

Professor Lejla Batina, Committee Member, Radboud University

_____

Professor Berk Sunar, Committee Member, Worcester Polytechnic Institute

_____

Professor Shahin Tajik, Committee Member, Worcester Polytechnic Institute

This dissertation is dedicated to my parents.

Without their constant support, patience, and love the completion

of this work would not have been possible. They made me who I am.

At the same time, my thanks go to my sister who has always inspired me.

Abstract

Hardware attacks, such as power side-channel attacks, jeopardize the security of embedded systems at a low cost. Protecting embedded systems against such attacks entails protecting hardware blocks, software programs, and the integration of hardware and software. As the development of hardware and software is typically independent, novel cross-layer protection mechanisms are desirable. The development of hardware and software itself has its own complexities. A hardware designer or a software developer implements the hardware or software in a high-level language, which is later optimized to the final product using automated tools. These automated tools are chiefly concerned about optimizing a design for better performance and can omit an inserted countermeasure at higher design abstraction layers. Furthermore, it is essential to evaluate the vulnerability of a system before its deployment. Design-time leakage assessment techniques can find whether a design is vulnerable to side-channel attacks. At the time of writing this dissertation, design-time leakage assessment methodologies are not developed to automatically find the cause of an observed leakage in hardware and software.

This dissertation contains three main contributions. First, we present cross-layer countermeasures to protect embedded systems against power side-channel analysis and fault injection. Through hardware-software co-designed protection mechanisms we are able to ensure the implemented protections persist at the final product. Second, we introduce a methodology to enable faster design-time leakage assessment. We further test the validity of our design-time assessment compared to physical measurements by designing and fabricating a custom chip. Last, we establish a technique to automatically pinpoint the cause of a certain observed power side-channel leakage in hardware and software.

Acknowledgments

My sincere gratitude goes to my advisor, Professor Patrick Schaumont, for his constant trust and support during the five years of my Ph.D. studies. He has been a great example of integrity and dedication in research. I learned all I know about being a researcher from working in his lab and I will carry these lessons with me in my future career.

I also want to extend my heartfelt appreciation to my committee members Professor Lejla Batina, Professor Berk Sunar, and Professor Shahin Tajik for their thoughtful feedback and support in forming this dissertation.

It was my great honor to have the internship opportunities at USC Information Sciences Institute, Riscure, and Apple. My thanks go to Dr. Travis Haroldsen, Jasper van Woudenberg, Dr. Cees-Bart Breunesse, and Joseph Lu for teaching me useful skills and broadening my view in my career.

I'd like to thank the members of Secure Embedded Systems lab in Virginia Tech and Vernam group at WPI. It was my honor to call you my colleagues and look up to you. Last but not least, I want to thank my friends, near or far, who helped me greatly during the recent challenging years.

Contents

List of Tables

List of Figures

Glossary

CPA Correlation Power Analysis.

DMA Direct Memory Access.

DPA Differential Power Analysis.

DRC design rule checking.

EDA electronic design automation.

FI fault injection.

HA hardware attack.

HDL hadware description language.

IR intermediate representation.

ISA Instruction Set Architecture.

ISE Instruction Set Extension.

LVS layout versus schematic.

P&R place and route.

PCB printed circuit board.

PHA physical hardware attack.

RTL register-transfer level.

SCA side-channel analysis.

SoC System-on-Chip.

Chapter 1

Introduction

In today's connected world, many tasks are done by embedded systems. These systems are tightly constrained and often made for a single specific function. They can perform a diverse set of tasks as simple as controlling the temperature of a room to the much more complex task of driving vehicles. Even though some of these tasks can be done entirely by a hardware chip (i.e. hardware-only architecture), embedded systems contain a micro-controller to run software codes (i.e. hardware-software (HW-SW) architecture). Support for software in such systems brings more flexibility to the design and enables possible future upgrades.

The simplicity and constrained nature of embedded systems make them appealing targets for relatively low-cost attacks. For instance, an embedded system that runs a cryptographic algorithm in software can be the target of a hardware attack. The attacker can find the key of the cryptographic algorithm, knowing only pairs of input-output from the device, by injecting faults during the operation of the software or performing power side-channel analysis. Many of such systems rely on cryptography for security. However, even though cryptographic algorithms are mathematically secure against cryptanalysis, their implementations can still make them vulnerable to hardware attacks (HAs). An adversary with physical access to these vulnerable implementations, can carry out physical hardware attacks (PHAs) and break

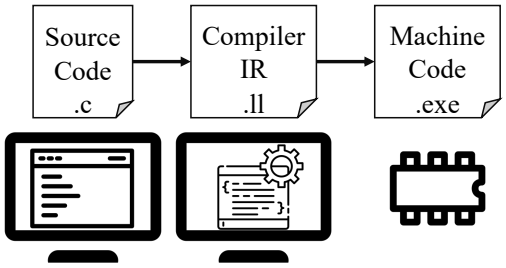


Figure 1.1: Design abstraction layers in software development

the cryptographic algorithms by finding their secret key. Even without physical access, an attacker can carry out remote attacks for the same purpose. These attacks enable adversaries to gain control over the systems (to run malicious code for example) or gain access to their internal secret data.

Embedded systems have a layered design. In hardware attacks on embedded systems, the hardware components are the direct target of the attack. However, the goal of the attack could be to control the execution of the software or gain information about data internal to the program. Therefore, in protecting HW-SW architectures against PHAs, there are two main players involved; The hardware and the software. On top of that, each of these components have their own abstraction layers during their development. The integration of hardware and software and the multitude of design abstraction layers, make the vulnerability assessment of such devices to PHAs challenging.

1.1 Software Design Abstraction Layers

Figure 1.1 shows the design abstraction levels in a software development project. A software programmer normally writes a code in a high-level language (C, Java, Python, etc.). This *source code* is the highest abstraction layer directly written by the programmer. The source code can use libraries by their APIs and hide their complexity. To run this software on a micro-controller, the source code should be converted to machine code to form the binary

file which will be stored in the program memory.

To generate the binary file, the source code will run through a cross-compiler for the target micro-controller architecture. The compiler consists of several passes for optimization and machine code generation. To make the compiler passes run more efficiently, the source code is first translated into a *compiler intermediate representation (IR)*. This new representation also makes the compiler optimization passes applicable to different source code languages. A compiler IR is a data structure that represents the source code and is designed to be conducive to optimization and translation.

The last level of abstraction in the design of a software program is the *machine code*. The machine code is stored in the program memory of the micro-controller. The binary code will be fetched and executed by the micro-controller. This is the least human-readable layer of software abstractions as it only consists of binary values. To make the contents of the machine code more readable, disassemblers generate the assembly code from the binary file.

Since the compiler passes mainly optimize the binary code for either speed or code size (at different levels), any implemented countermeasure against PHA in higher design layers can potentially be missing in the binary code. For example, the following code snippet shows a countermeasure inserted at the source code level. The goal of the `else` branch is to make the execution time constant regardless of the value of `try_pwd`. However, this branch causes a dead code and will be eliminated at the IR level by compiler optimization passes. Similarly, a countermeasure inserted at the IR level by optimization passes can be removed by code generation passes.

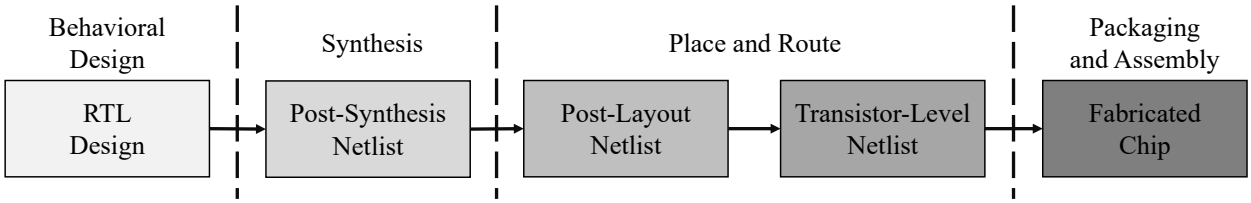


Figure 1.2: Design abstraction layers in digital hardware

```
char try_pwd[5];
char pwd[5] = "password";

unsigned correct_letters = 0;
unsigned dummy = 0;

for (unsigned i=0 ; i<5 ; i++)
    if (try_pwd[i]==pwd[i])
        correct_letters++;
    else
        dummy++;

if (correct_letters == 5)
    unlock_door();
```

It is therefore a challenge to ensure that the binary code indeed contains the required countermeasures. In this dissertation, we show examples of novel countermeasure insertion against PHA at different design abstraction layers and ensure they remain functional at the machine code level.

1.2 Hardware Design Abstraction Layers

Figure 1.2 shows the abstraction layers in a digital hardware design process. In digital hardware design cycle, once the system specifications are set, the behavioral model of the circuit is prepared in hadware description languages (HDLs). Verilog, SystemVerilog, and VHDL are the most commonly-used HDLs. This behavioral modeling is done in *register-transfer level (RTL)*.

The RTL is then run through synthesis electronic design automation (EDA) tool which generates the *gate-level netlist*. The synthesis tool optimize an RTL design to meet the area, power, and timing budgets while defining the design in terms of the given standard cell library. Gate-level synthesis can therefore introduce new buffers into the design and omit or combine some of the existing logic.

Next, the gate-level netlist will run through the P&R EDA tool to generate the layout which is sent for fabrication and consists of several steps; Preparing the floorplan, placement of cells and macros, synthesis of the clock tree, and routing the signals. After P&R, technology-specific design rule checking (DRC) and layout versus schematic (LVS) checks are run to make sure the implementation is ready for fabrication. The output of the P&R can be taken either as a netlist, *i.e.*, *post-layout netlist*, or at a lower abstraction level showing the transistors and the RLC information of their connections, *i.e.*, *transistor-level*.

All the previously mentioned abstraction layers (*i.e.*, pre-silicon) are for preparing a design for fabrication. During the fabrication process itself, process variation will cause different variations in the characteristics of transistors. Consequently the pre-silicon transistor-level attributes do not exactly match the *post-silicon* attributes. The fabricated chip will need a host board to run and communicate with the world. The integration of a chip into a board can itself make more complex discrepancies between the pre-silicon model of the circuit and the final running system.

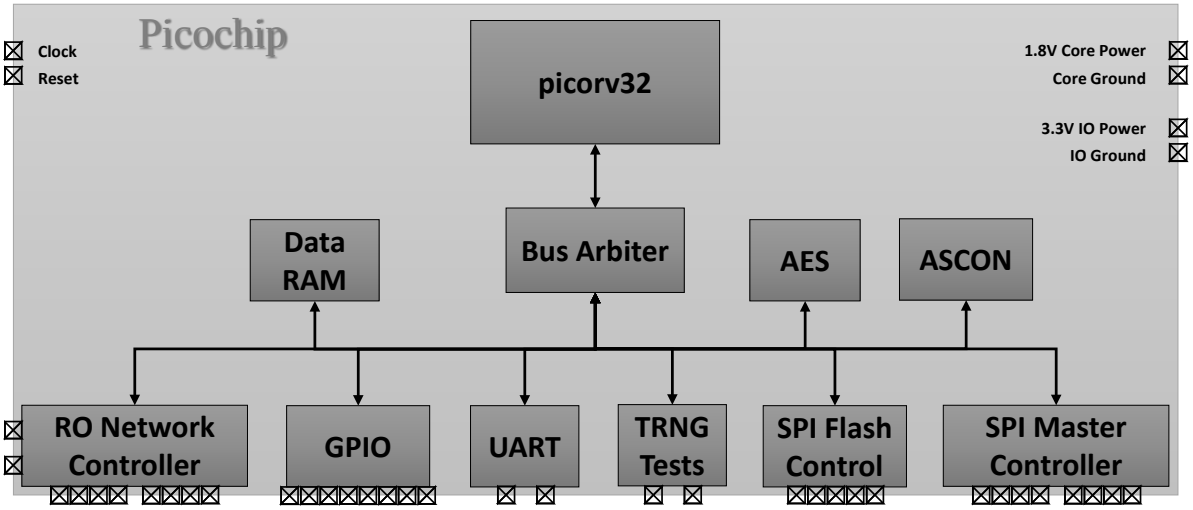


Figure 1.3: Block diagram of Picochip

Any countermeasure that is introduced to a hardware design, will inevitably go through the aforementioned abstraction layers. It is therefore possible to see an implemented countermeasure not being functional in the final taped out chip. This pronounces the importance of design-time analysis of vulnerabilities. Furthermore, higher abstraction layers limit the complexity of the modeled vulnerabilities. For example, at RTL level, the signal delays are not modeled. Therefore, any vulnerability caused by such delays will be missed in a leakage assessment carried out at the RTL level. In this dissertation, we present methods to find sources of vulnerability against PHAs in hardware and link them to their causes in software.

An example hardware design flow. As an example, we demonstrate the design procedure for Picochip in the following. Picochip was designed with the goal of providing a completely locally designed platform. The local design of this platform enabled experiments in comparing pre- and post-silicon attributes of power side-channel leakage assessment. In the first stage, the overall structure shown in Figure 1.3 was modeled in RTL format. Using Synopsys Design Compiler (Synopsys DC) the RTL files were synthesized into the gate-level netlist for TSMC 180nm standard cell library and frequency of 80MHz. The size of data

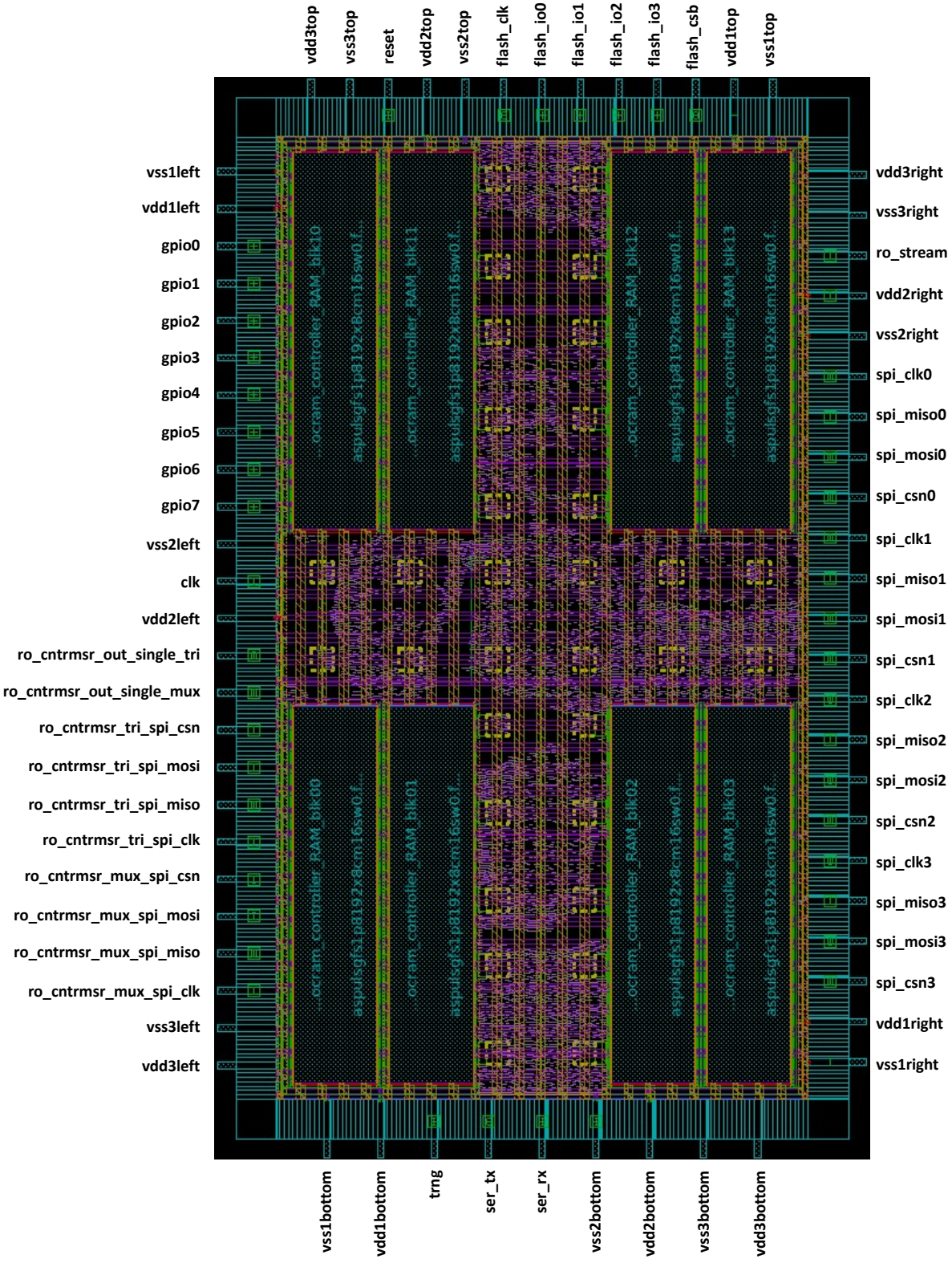
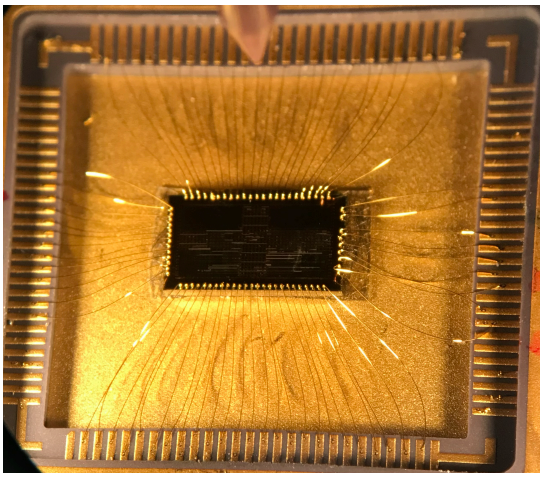
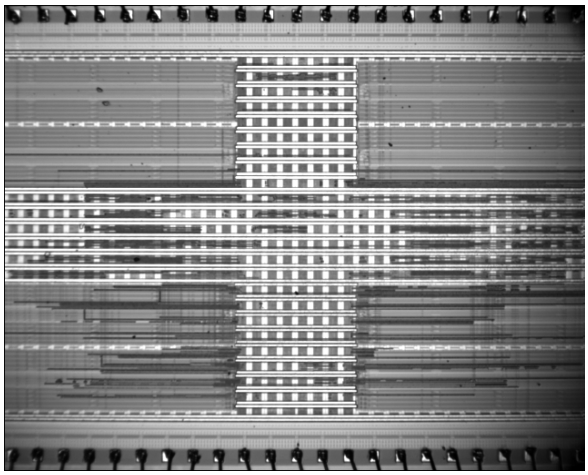


Figure 1.4: Layout of Picochip after P&R in ICC2



(a) Wire-bonding Picochip to a package



(b) Microscopic picture of a wire-bonded chip

Figure 1.5: Wire-bonded Picochip

RAM was 64kB. Next, the place and route was performed using Synopsys IC Compiler II (Synopsys ICC2). In this step, first, the memory macros, IO pads, bond pads, and logic cells were placed in a 3mm×5mm area. Next, the clock tree was generated to route the clock signal. Other signals were routed next and filler cells were inserted to meet the minimum metal density requirement of the foundry. The result of this step is shown in Figure 1.4. After running final verification steps, namely DRC and LVS using Mentor Graphics Calibre, the GDSII file was sent for fabrication. Once the chip was fabricated, we wire-bonded (Figure 1.5a) it to a package to be able to integrate it into a custom designed printed circuit board (PCB) called Saidoyoki board detailed in Chapter 6. Figure 1.5b shows a microscopic picture of wire-bonded Picochip.

1.3 Physical Attacks on HW-SW Architectures

PHAs are categorized into active and passive attacks [150]. In active attacks, also known as fault injection (FI), the adversary makes changes to the device under attack to either gain information about its internal state or disrupt its correct behavior. Faults can be injected

into devices using different techniques such as optical fault injection [147], electromagnetic fault injection (EM-FI), voltage and clock glitching. Optical fault injection injects fault by exposing silicon to high intensity light sources. This technique requires the chip to decapsulated from the package. On the other hand EM-FI can inject faults through the package, eliminating the need for chip decapsulation.

On the other hand, the adversary can carry out passive attacks, also known as side-channel analysis (SCA), without any changes to the device by merely observing and measuring its behavior given arbitrary known inputs. SCA can be performed by analyzing the power consumption [37], electromagnetic emanation [68], or timing information [111] of the device among other mediums of leakage.

When physical attacks are conducted on HW-SW architectures, even though the hardware component is the immediate cause or effect of the attack, the main target of the attack can be the software. For example, an attacker can temporarily tamper with the supplied voltage to the device (voltage glitching fault injection) to cause the execution of an instruction in the software code to be skipped. As another instance, an adversary can measure the power consumption of the device, given controlled inputs, to find the value of a secret data used in the software code. In a HW-SW system, the machine code of the software is fetched and executed on the group of transistors as building blocks of the hardware circuit. Consequently, physical attacks on HW-SW architectures deal with a combination of abstraction layers in both the software and the hardware components. In this dissertation, among different types of PHAs, our main focus is on power side-channel analysis.

1.4 Protecting Embedded Systems against PHAs

Power side-channel analysis attacks are able to find the secret data on a device because of the contribution of the secret data to the power consumption of a device. The countermeasures

against such attacks can be classified into two groups: hiding and masking.

Hiding In a hiding countermeasure, the goal is to make the power consumption of a device independent of the internal data by either randomizing or equalizing. Hiding countermeasure can be implemented in both time and amplitude dimensions [117]. In time domain, by shuffling the execution of operations or blocks the alignment of the power traces will be disturbed and therefore post-processing the power traces will become challenging. In amplitude domain, by reducing the signal to noise ratio (SNR) of the contribution of the secret data to the power consumption, finding the correct key will become increasingly more difficult for an adversary. A simple example of hiding countermeasure by equalizing the power consumption is to perform the inverse of every operation in parallel. For example, for a simple AND operation $C = A\&B$, the inverse operation will be the OR operation on inverted inputs $\bar{C} = \bar{A}|\bar{B}$. A more accurate way to equalize the power consumption is Wave Dynamic Differential Logic (WDDL) [62].

Masking Masking breaks the dependency of the power consumption of a device on the underlying secret data by randomizing the internal data. In Boolean masking for every input data bit, a random bit is taken from a uniform distribution and XOR'ed with the data bit to generate the masked shares. The operations are then all adjusted to work on the masked data and generate the masked output. For example to apply masking to a simple AND operation $C = A\&B$, first two random numbers are generated R_1 and R_2 and the input shares are calculated: $A_1 = R_1$, $A_2 = A \oplus R_1$, $B_1 = R_2$, $B_2 = B \oplus R_2$. There are multiple strategies to calculate the output shares. Trichina [156] presented one of such implementations as: $C_1 = (((R_3 \oplus (A_1\&B_2)) \oplus A_2\&B_1) \oplus A_2\&B_2) \oplus A_1\&B_1$, $C_2 = R_3$.

Several studies [60] have shown that glitches will break the security of normal masked implementation. Other approaches were introduced including Threshold Implementation

(TI) [127] and Domain-Oriented Masking (DOM) [83] that can remain secure in the presence of glitches.

1.5 State of the Art

Several previous work related to securing software against hardware attacks on embedded systems work on fixing vulnerabilities at the assembly code (i.e. source code) of the software program or the hardware components. This magnifies the importance of cross-layer approach to secure HW-SW architectures. Rosita [141] finds vulnerabilities to power side-channel analysis in HW-SW architectures through modelling the leakage at the assembly code level using an extended version of ELMO [124]. It then fixes the observed leakage by refreshing the masks. The underlying assumption in Rosita is that the software code is already implementing masking as a countermeasure but may not be perfectly secure. Rosita++ [140] extends Rosita for higher-order masked software programs.

COCO and PARAM address the power side-channel leakage observed in a HW-SW architecture at the hardware level. PARAM works at behavioral simulation at the RTL abstraction level while COCO proves the security of a masked design at post-synthesis gate-level. Since the leakage from HW-SW architectures stem from the hardware, these approaches are stronger in finding the cause of the leakage. More advanced architectures require new masking rules in software and changes to the hardware to ensure a secure implementation. These complexities have been studied for a superscalar implementation of RISC-V, i.e. SweRV core, [78] using REBECCA [30], a tool originally developed for formal verification of masked hardware.

In the previous body of work, gate-level power simulation is considered costly and hence researchers have developed tools and methodologies to model the power consumption of a device. ELMO [124] is an instruction accurate power emulator for ARM Cortex-M0 devices.

The power leakage is estimated in ELMO based on the interaction between consecutive instructions. ABBY [19] presents an automated methodology to generate leakage models based on training.

The aforementioned contributions (with the exception of PARAM) all focused on finding and fixing bugs in masked software implementations. Althoff et al. [4] on the other hand present architecture support for intermittent hiding countermeasure in their work called blinking. While PARAM is not limited to masked software, it applies the leakage assessment at the RTL level which constrains the observed leakage at the behavioral model.

1.6 Challenges in Protecting HW-SW Architectures

Vulnerabilities to PHAs imply the necessity to implement countermeasures against such attacks both in hardware and in software. A software developer will write the program source code in a high level language potentially including protections against hardware attacks. However, the software source code will go through many optimization and code generation passes in the compiler to generate the binary file. Similarly, for a hardware designer it is most straight-forward to implement countermeasures at the RTL level which will go through many automated steps during synthesis and P&R to prepare the GDSII file ready for fabrication.

Moving from one abstraction layer to the next is done with automated tools (compiler tool-chain for software and CAD tools for hardware). These tools are mainly developed with the optimization for area or code size and speed in mind. As many of the countermeasures against PHAs do not go hand in hand with the most optimized design, they might be modified or completely lost during these automated processes. This necessitates assessment of the vulnerabilities and inserted countermeasures until the lowest abstraction level possible.

Furthermore, ensuring the hardware design and the software design are protected separately does not guarantee that their integration in the complete HW-SW architecture will

be secure. It is essential to assess the vulnerabilities of a system prior to its employment. Early leakage assessments of a design can be done by prototyping or simulation. Prototyping a design on FPGAs can be done as soon as the RTL designs of the hardware components are ready. However, the physical signature of the FPGA implementation of a design can be different from its ASIC implementation simply because of their different building blocks. Simulation-based leakage assessment on the other hand can overcome this downside of prototyping while increasing the assessment time. Simulation-based leakage assessment has the extra merit that it enables finding the source of every observed leakage in the hardware components and software programs. The lack of hybrid tools and methodologies to assess vulnerabilities in hardware connected to software brings more complexity to this task. Relying solely on verified software masking does not guarantee a secure final HW-SW design. As mentioned by Becker et al. [24], many of the provable secure masked software end up having power side-channel leakage in a physical implementation. Traditionally, in embedded systems, software was envisioned to bring flexibility and hardware to bring security and performance to the system. This division of hardware and software roles has proven to be imprecise with the emergence of HAs on HW-SW architectures. In this dissertation, we address three challenges as shown in Figure 1.6.

Challenge 1. Preserving countermeasures across design abstraction layers. Due to the intrinsic layered structure of embedded systems, when implementing countermeasures, several abstraction layers should be considered. Any protection mechanism inserted at a higher abstraction layer must be preserved at the lowest abstraction layer (final system).

Challenge 2. Simulating and emulating power consumption. As stressed before, it is imperative to evaluate a HW-SW architecture holistically. Simulation-based approaches capture a close-to-complete hardware and software integration in such architectures. One

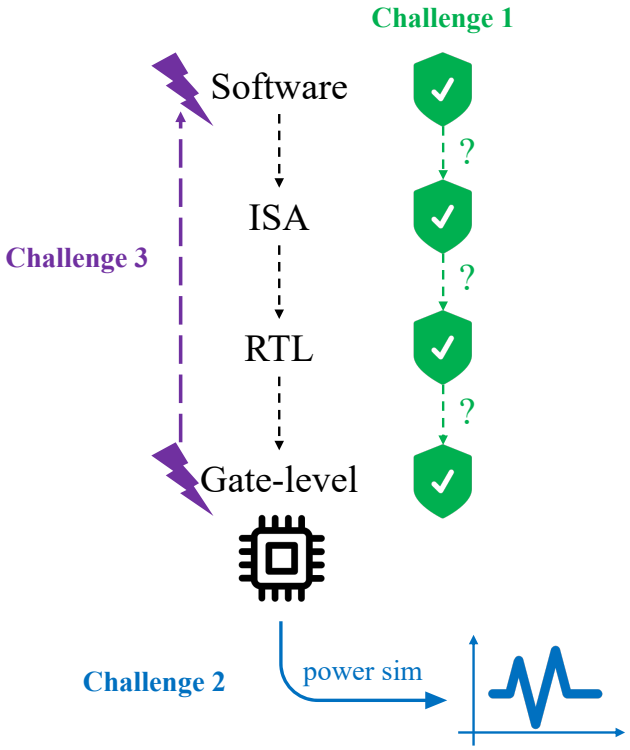


Figure 1.6: Addressed challenges in protecting embedded systems against hardware attacks.

of the challenges of simulation-based leakage assessment is power modeling. Emulating leakage is useful especially for commercial devices for which no hardware design source is available. In this dissertation, we study ways to reduce the power simulation time at the gate-level without affecting the accuracy of the simulated traces with respect to the power side-channel leakage. Our simulation setup shows using the existing CAD tools to simulate power traces for leakage assessment is practical. Using CAD tools has the advantage of using the information provided in the standard cell library which is the best of the available information to a designer at design time.

Challenge 3. Finding the cause of an observed vulnerability. Systematically finding the cause of an observed side-channel leakage, regardless of whether or not the implementation contains a countermeasure, is yet to be developed.

Proposal. In this work, we address the mentioned challenges as described in the following:

1. To show the importance of simultaneously addressing different layers of abstraction to protect HW-SW architectures, we employ hardware-software co-design methodologies for protecting micro-architectures against power side-channel attacks. This proposal addresses challenge 1.
2. We design and fabricate a chip to compare the simulation-based pre-silicon side-channel leakage assessment with the physical post-silicon measurements. We use CAD tools to simulate power consumption of a design and show techniques to reduce the power simulation time without losing accuracy of leakage assessment. This proposal addresses challenge 2.
3. We present simulation-time evaluation of side-channel leakage. Furthermore, we present techniques to pinpoint the cause of an observed leakage in both hardware (at the granularity of a gate) and software (at the granularity of the execution block of an instruction). This proposal addresses challenge 3.

1.7 Contribution and Outline

In this dissertation, we address protecting the software running on a HW-SW architecture against physical hardware attacks. Specifically, we have considered power side-channel attacks, timing side-channel attacks, and fault injection attacks. As mentioned previously, security assessment of HW-SW systems requires cutting through multiple abstraction layers as well as linking the hardware and software components. Throughout this dissertation, we address crossing abstraction layers in software and hardware.

Furthermore, previous works make implicit assumptions about simulated or emulated power traces being similar to post-silicon traces. In this dissertation we put this assumption to test; We design and fabricate the aforementioned Picochip in CMOS 180nm technology as

a platform for comparing pre-silicon simulation-based power leakage with post-silicon leakage. In the following we enumerate the main contributions and chapters of this dissertation.

Parallel Synchronous Programming (Chapter 2) - Challenge 1. In typical embedded applications, the precise execution time of the program does not matter, and it is sufficient to meet a real-time deadline. However, modern applications in information security have become much more time-sensitive, due to the risk of timing side-channel leakage. The timing of such programs needs to be data-independent and precise. We describe a parallel synchronous software model, which executes as N parallel threads on a processor with word-length N. Each thread is a single-bit synchronous machine with precise, contention-free timing, while each of the N threads still executes as an independent machine. The resulting software supports fine-grained parallel execution. In contrast to earlier work to obtain precise and repeatable timing in software, our solution does not require modifications to the processor architecture nor specialized instruction scheduling techniques. In addition, all threads run in parallel and without contention, which eliminates the problem of thread scheduling. We use hardware (HDL) semantics to describe a thread as a single-bit synchronous machine. Using logic synthesis and code generation, we derive a parallel synchronous implementation of this design. We illustrate the synchronous parallel programming model with practical examples from cryptography and other applications with precise timing requirements. Parallel synchronous programming gives control over the execution time, normally perceived as a hardware effect, to the software. This programming model serves as a hardware-software co-design even though it does not involve any changes to the hardware.

Rewrite to reinforce (Chapter 3) - Challenge 1. Fault injection attacks are hardware attacks that can cause errors in software. Countermeasures against fault attacks can be implemented in hardware, software, or a combination of both. Software countermeasures

implemented by hand do not scale, and are error prone. Tooling to insert countermeasures automatically can target different stages of the development life cycle.

We explore two possible approaches to inject countermeasures on the binary level, at the end of the development life cycle. The first is an LLVM compiler stage where we implement a countermeasure that automatically hardens conditional branches. The compiler stage is then applied on the LLVM intermediate representation of a binary lifted by Revng, and compiled back into a hardened binary using the compiler pass. The second approach injects countermeasures directly into the machine code of a binary using the Ddisasm rewriting framework, whose intermediate representation includes the original machine code. This work compares applying countermeasures at two different design abstraction layers of the software; Namely, binary code and the compiler intermediate representation. The comparison is made in terms of security and efficiency.

Domain-oriented masking instruction set architecture (Chapter 4) - Challenge 1. An important selling point for the RISC-V instruction set is the separation between ISA and the implementation of the ISA, leading to flexibility in the design. We argue that for secure implementations, this flexibility is often a vulnerability. With a hardware attacker, the side-effects of instruction execution cannot be ignored. As a result, a strict separation between the ISA interface and implementation is undesirable. We suggest that secure ISA may require additional implementation constraints. As an example, we describe an instruction-set for the development of power side-channel resistant software. This extended ISA serves as a hardware-software co-design approach for protecting a micro-architecture against power side-channel attacks.

Skiva-V (Chapter 5) - Challenge 1. The bitsliced programming model has shown to boost the throughput of software programs. However, on a standard architecture, it exerts

a high pressure on register access, causing memory spills and restraining the full potential of bitslicing. In this work, we present architecture support for bitslicing in a System-on-Chip. Our hardware extensions are of two types; internal to the processor core, in the form of custom instructions, and external to the processor, in the form of direct memory access module with support for data transposition. We present a comprehensive performance evaluation of the proposed enhancements in the context of several RISC-V ISA definitions (RV32I, RV64I, RV32B, RV64B). The proposed 14 new custom instructions use 1.5× fewer registers compared to the equivalent functionality expressed using RISC-V instructions. The integration of those custom instructions in a 5-stage pipelined RISC-V RV32I core incurs 10.21% and 12.72% overhead respectively in area and cell count using the SkyWater 130nm standard cell library. The proposed bitslice transposition unit with DMA provides a further speedup, changing the quadratic increase in execution time of data transposition to linear. Finally, we demonstrate a comprehensive performance evaluation using a set of benchmarks of lightweight and masked ciphers. The Skiva-V System-on-Chip (SoC) is a show-case for boosting the performance of protected software on an SoC by integrating secure hardware extensions inside and outside of the processor core.

Saidoyoki (Chapter 6) - Challenge 2. Predicting the level and exploitability of side-channel leakage from complex SoC design is a challenging task. We present Saidoyoki, a test platform that enables the assessment of side-channel leakage under two different settings. The first is pre-silicon side-channel leakage estimation in SoC, and it requires the use of fast side-channel leakage estimation from a high-level design description. The second is post-silicon side-channel leakage measurement and analysis in SoC, and it requires a hardware prototype that reflects the design description. By designing an in-house SoC and next building a side-channel leakage analysis environment around it, we are able to evaluate design-time (pre-silicon) side-channel leakage estimates as well as prototype (post-silicon)

side-channel leakage measurements. The Saidoyoki platform hosts two different SoC, one based on a 32-bit RISC-V processor and a second based on a SPARC V8 processor. In this contribution, we highlight our design decisions and design flow for side-channel leakage simulation and measurement, and we present preliminary results and analysis using the Saidoyoki platform. We highlight that, while the post-silicon setting provides more side-channel leakage detail than the pre-silicon setting, the latter provides significantly enhanced test resolution and root cause analysis support. We conclude that pre-silicon side-channel leakage assessment can be an important tool for the security analysis of modern Security SoC.

Leverage the Average (Chapter 7) - Challenge 2. Pre-silicon side-channel leakage assessment is a useful tool to identify hardware vulnerabilities at design time, but it requires many high-resolution power traces and increases the power simulation cost of the design. By downsampling and averaging these high-resolution traces, we show that the power simulation cost can be considerably reduced without significant loss of side-channel leakage assessment quality. We introduce a theoretical basis for our claims. Our results demonstrate up to 6.5-fold power-simulation speed improvement on a gate-level side-channel leakage assessment of a RISC-V SoC. Furthermore, we clarify the conditions under which the averaged sampling technique can be successfully used.

Non-specific Architecture Correlation Analysis (Chapter 8) - Challenge 3. While side-channel leakage is traditionally evaluated from a fabricated chip, it is more time-efficient and cost-effective to do so during the design phase of the chip. We present a methodology to rank the gates of a design according to their contribution to the side-channel leakage of the chip. The methodology relies on logic synthesis, logic simulation, gate-level power estimation, and gate leakage assessment to compute a ranking. The ranking metric

can be defined as a specific test by correlating gate-level activity with a leakage model, or else as a non-specific test by evaluating gate-level activity in response to distinct test vector groups. Our results show that only a minority of the gates in a design contribute most of the side-channel leakage. We demonstrate this property for several designs, including a hardware AES coprocessor and a cryptographic hardware/software interface in a five-stage pipelined RISC-V processor. This work enables a hardware designer to pinpoint the group of gates that are the cause of leakage.

RootCanal (Chapter 9) - Challenge 3. 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.

Chapter 2

Parallel Synchronous Programming

In this chapter, we present Parallel Synchronous Programming (PSP). PSP runs a state machine on top of a processor and serves as a programming model to provide predictable execution time. This work is published at IEEE Embedded Systems Letters [103]. Using the developed automated tool to generate PSP code (PSPCG), we applied this model of programming to a number of candidates in NIST's Light Weight Cryptography workshop. The results have been presented at NIST's LWC workshop [96].

2.1 Introduction

Producing software with precise, repeatable timing is a challenging task. First, the software application itself may have data-dependent processing complexity, such as with data-dependent loops. Second, the execution time of the application on the processor may be affected by the memory hierarchy and the run-time state of the processor. Third, the timing of the execution may be affected by resource contention when several parallel threads share the same processor resource. Among these three problems, the second and the third are most difficult because they are outside of the control of the programmer. In cryptographic applic-

ations, data-dependent timing variations may be exploited as timing side-channel leakage, either directly as an effect of data-dependent control flow, or indirectly as an effect of contention on shared processor resources. To avoid timing side-channels, we need data-independent timing.

In this contribution, we propose a programming model that yields these timing characteristics. We contrast our proposal with earlier work towards precise software timing for embedded applications, PRET [113, 176]. A fundamental idea of PRET is to use instruction scheduling to avoid resource contention in the processor in the pipeline. By spacing the instructions of timing-critical threads several cycles apart, stall-free execution is achieved in the pipeline. As a consequence, the timing of individual threads is repeatable regardless of the processor state. To ensure overall processor utilization, PRET combines multiple timing-critical threads with time-interleaving and a customized instruction scheduling technique [176].

Our insight in this chapter is that such time-sensitive threads can also be combined *spatially* within a processor word, instead of *temporally* using interleaved instruction streams. The advantage of spatially combining the threads (instead of using time-based interleaving) is that we don't need to adapt the processor for interleaved instruction execution. To implement a spatial arrangement of threads, we organize each thread as a single-bit program, and execute the overall application as a vectorized version of the single-bit program. We emphasize that the proposed model goes beyond software bit-slicing [29], which is strictly functional and ignores the control flow and the state within each slice.

To simplify the development of single-bit programs, we adopt a synchronous execution model. A single-bit program is captured as a synchronous Finite State Machine with Datapath (FSMD), and the execution of this program follows a sequential schedule of the bit-operations that define the FSMD. The vectorized form of the single-bit program is then achieved with bitwise instructions over the processor word. The vectorized form is a *parallel*

synchronous program. Since each thread has its own state, each thread executes as an independent FSMD. However, the instruction count for one iteration of the overall program is constant and repeatable, and therefore the execution time of these FSMD threads becomes repeatable too. A prototype implementation of a synthesis tool starts from a Verilog input specification and generates C code with inline assembly optimized for an embedded target. We demonstrate several useful examples of parallel synchronous programming (PSP).

The outline of this chapter is as follows. Section 2.2 develops the cardinal components of parallel synchronous programming. Section 2.3 discusses an example and proposes a code generation methodology. Section 2.4 describes experimental results. Section 2.5 concludes the contribution.

2.2 Preliminaries

We develop a software execution model that leads to repeatable, data-independent timing. We first define what is meant by repeatable and data-independent timing in software. We then describe software bitslicing, which can offer such timing characteristics for functions (*i.e.*, straight-line stateless programs). Next, we explain how to extend the semantics of software bitslicing from straight-line programs to synchronous FSMD. The result is a Parallel Synchronous Program (PSP).

2.2.1 Desired Timing Properties

Programs written as PSP aim for repeatable timing as well as data-independent timing. The former is useful in real-time embedded software design, while the latter is useful for secure systems design. We motivate and differentiate each property.

Edwards *et al.* make a distinction between repeatable timing and predictable timing [56]. Repeatable timing means that every correct execution of a program uses the same timing.

Repeatable timing is desired as a property of the program, not of the program running on a specific processor. Repeatable timing is needed in the context of real-time applications when timing jitter is a concern. For example, when a physical sensor must be read from software at a specific sample rate, then the software needs to have repeatable timing. Jitter is typically caused by resource contention and interrupts.

A second relevant domain for PSP is that of secure software. In recent years, a rich collection of attacks have been found to exploit the *implementation* characteristics of secure software rather than the program logic itself. The best known of these are side-channel attacks and micro-architectural attacks, which rely on precise execution time measurement [73]. To thwart these attacks, software with (secret-)data-independent timing is needed. This is hard because modern micro-architectures are rife with architectural contention and context-dependent timing. Even if there are no obvious dependencies in the program logic, there may still be hidden dependencies in the micro-architecture. The cryptographic community is well aware of the risk of timing-based side-channel leakage, leading to the design of so-called *constant-time* software that avoids data-dependencies in the program execution time [3]. The resulting programs are not literally constant-time, but rather they adopt data-independent control flow and memory access patterns.

We argue that software written as a Parallel Synchronous Program (PSP) provides repeatable timing as well as data-independent timing. PSP achieves these properties by combining two concepts: software bitslicing and synchronous FSMD. The following subsections introduce both.

2.2.2 Bitslicing

Software bitslicing was originally proposed for high throughput software implementations [29]. In this model of programming, a program is expanded into 1-bit (Boolean) operations as

```
while true do
    wait for clock_tick;
    outputs = eval(inputs, current_state);
    next_state = update(inputs, current_state);
    current_state = next_state;
end
```

Figure 2.1: Basic structure of a synchronous program

follows. A $k-$bit variable with bits $b_{k-1}...b_1b_0$ is distributed over k registers $R_{k-1}...R_1R_0$, such that register R_i holds bit b_i. An $N-$bit processor operates as an $N-$way SIMD processor, processing N instances of the $k-$bit variable in parallel, and storing these instances in k registers. Bitsliced programs are Boolean programs written with bit-wise logic operations. The rationale of bitslicing is that it guarantees full utilization of the processor word-length. The absence of control flow ensures that each iteration through a bitslice function uses the same amount of instructions. In addition, the absence of state (memory) in a bitslice function eliminates cache timing effects. For this reason, bitslicing is often applied in the context of developing programs that are *constant-time* (in the cryptographic sense). However, software bitslicing is insufficient as a general-purpose methodology for software. Because bitslice functions do not have control flow, control operations are typically emulated using non-bitsliced logic surrounding bitsliced expressions. This prevents individual slices from operating as independent threads of control. Bitslice programming essentially applies only to functions. The management of the program state resides outside of the bitslice logic.

2.2.3 Synchronous FSMD

We next describe how to introduce control flow and state into software bitslicing. A Boolean (1-bit) program does not offer the concept of an address space or control flow instructions. Therefore, we propose introducing the control flow through the intermediary of a synchronous FSMD. FSMDs are common in digital hardware design, and they are routinely applied in

register-transfer-level designs. An FSMD is a synchronous model of computation combining a datapath and a finite-state controller. Computations are done on the datapath under control of the FSM [136]. Each synchronous clock cycle, the FSM computes a single state transition and selects one or more operations in the datapath. The execution of datapath operations depends on the current state of the FSM, and the state transition conditions in the FSM depend on the current state of the datapath. Conditional control flow is expressed using dataflow-like semantics: the datapath will compute both the true and false case of the control condition, and the correct result will be selected using multiplexing.

To map a synchronous FSMD into software, we adopt a synchronous execution model as shown in Figure 2.1. Every loop in this program corresponds to a single clock cycle of the synchronous FSMD model. The software awaits the occurrence of a clock tick to read all inputs and evaluate all outputs concurrently [41]. The `eval()` function in Figure 2.1 computes the FSM next-state as well as the datapath next-state. The `update()` function adjusts the current state of the FSM and the datapath to the next. State update is handled synchronously: each state variable in the synchronous FSMD is split into two copies, the *current_state* and the *next_state*. This avoids race conditions, and ensures that the program will always compute the same result regardless of the scheduling of `eval()` and `update()`. The PSP is implemented as N parallel copies of the program of Figure 2.1, where every thread is tied to the same global clock tick, and each thread is a 1-bit program expressed as a synchronous FSMD.

2.3 Synthesis of Parallel Synchronous Software

We next demonstrate how to create PSP. We first describe the example PSP design of a parallel greatest common divisor program, and next discuss a design flow that synthesizes PSP software from a synchronous FSMD description.

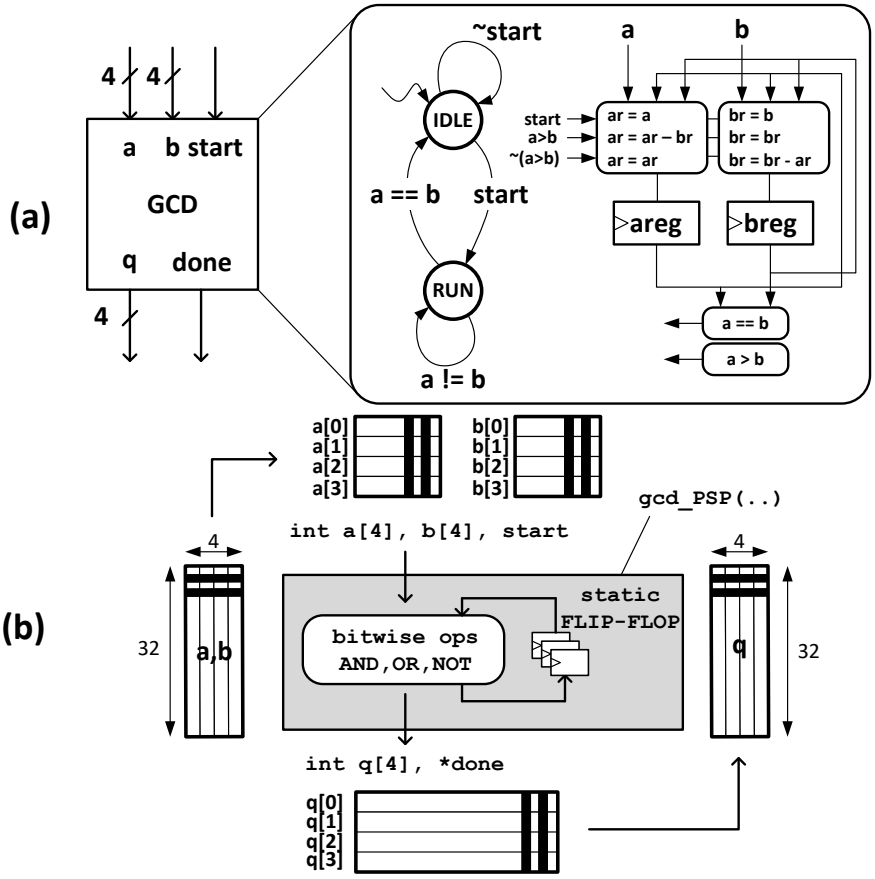


Figure 2.2: GCD (a) Interface and (b) FSMD model

Example Figure 2.2a shows the outline of a 4-bit GCD module. We express the functionality of the GCD algorithm as an FSMD model. After a `start` control pulse, the module reads two 4-bit inputs `a` and `b`, and repeatedly subtracts the smaller value from the larger value until they are equal. A `done` pulse is generated to indicate completion of the algorithm. A two-state control FSM drives the loading of two 4-bit registers `a` and `b` and their iterative computation.

A PSP version of the GCD algorithm for a 32-bit processor executes 32 parallel copies of the GCD. We create this software by converting the FSMD to a gate-level netlist using logic synthesis. We target a generic technology with a logically complete set of primitive functions (such as `AND`, `OR` and `NOT`) as well as a storage element such as a flip-flop (Figure 2.2b). The

outcome of the logic synthesis is a netlist in terms of logic elements. We then rewrite the netlist as a sequential function by leveling the netlist according to data dependencies from input to output. The logic cells are replaced by bit-wise operations, and the flip-flops are replaced by `static` (or global) variables. The resulting function declaration is as follows.

```
gcd_PSP(int a[4], int b[4], // data input
        int q[4],           // data output
        int start,          // control in
        int* done);         // status out
```

Each invocation of this function corresponds to a single synchronous iteration (one clock cycle of the synchronous FSMD). An important difference between the circuit of Figure 2.2a and the PSP function in Figure 2.2b is the degree of parallelism; The circuit in Figure 2.2 computes a single GCD whereas the `gcd_PSP` function is a software design that computes 32 *concurrent* GCD algorithms independently, each with their own `start` and `done` bits. The inputs and outputs of `gdc_PSP` are in bitsliced form. For example, `a[2]` contains the second bit of 32 different inputs. Hence, a call to `gcd_PSP` needs to transpose the input and output arguments.

An Automated Flow We implemented a software synthesis flow for PSP that starts from an FSMD description in a Verilog program. An open-source Verilog synthesis tool [163] converts the FSMD into a netlist in terms of generic target technology for Boolean logic and a state element. The target library for logic synthesis is adjusted in function of the targeted processor. Table 2.1 demonstrates a sample mapping for several embedded processors. The state elements (flip-flop) are mapped to `static` variables.

The netlist is then converted to software as follows. The netlist is topologically sorted, from the primary inputs and flip-flop outputs to the primary outputs and the flip-flop inputs.

Table 2.1: Instructions targeted by PSP synthesis

processor	suitable instructions for PSP
ARM Cortex-M4	AND, BIC, EOR, MOV, MVN, ORN, ORR
RISC-V	AND, OR, XOR
MSP430	AND, BIC, BIS, XOR
AVR	AND, COM, EOR, OR

Table 2.2: Evaluated encryption ciphers and comparison of performance of the PSP and normal implementations of them

	cipher properties				speed (cycles/byte)		
cipher	**block size**	**key size**	**rounds**	**type**	**PSP**	**normal**	**speedup**
SIMON	128	128	68	Feistel	744.48	1315.63	1.7×
PRESENT	64	80	31	SPN	399.61	1069.06	2.6×
LED	64	64	32	SPN	-	-	-
Midori	64	128	16	SPN	236.90	2233.38	9.4×

Next, each primitive gate is converted to a bitwise operation which is either emulated in C or else added through inline assembly. We rely on the C compiler to create a sequential schedule for the gate netlist that will minimize the register pressure on the processor. The following section applies the automated flow on several examples.

2.4 Experimental Results

We analyze our flow and the resulting performance using several examples. We target the 48 MHz ARM Cortex-M4F processor, which comes with the Texas Instruments MSP432P401R Launchpad and implements the ARMv7E-M architecture. Table 2.3 summarizes our results. The numbers reported on this table are compiled with size optimization (`-Os`).

The first two examples, GCD and PWM, illustrate the general-purpose nature of PSP as well as its real-time characteristics. For these examples, Table 2.3 lists the number of processor clock cycles per synchronous cycle. Computing 32 parallel GCD's thus takes 382

Table 2.3: Evaluation of parallel synchronous examples on 48MHz Cortex-M4F processor

	performance and cost			**instructions breakdown**									
example	**number of cycles (32 parallel runs)**	**throughput (Kbps)**	**code size (Kb)**	**AND**	**ORR**	**BIC**	**EOR**	**ORN**	**MVN**	**MOV**	**STR**	**LDR**	**overhead**
GCD	382	-	11.88	28	34	6	9	7	8	21	28	73	54.46%
PWM	239	-	11.82	29	20	11	6	2	8	0	23	39	44.93%
SIMON bit-parallel	381,175	515.79	23.40	907	470	180	367	27	4	18	1033	2002	60.96%
SIMON bit-serial	15,370,190	12.79	18.81	854	313	23	16	19	13	7	593	686	50.95%
PRESENT	102,301	960.93	17.79	226	282	60	119	70	32	24	454	861	62.92%
LED	139,949	702.43	20.29	379	301	80	395	60	60	138	556	1258	60.49%
Midori	60,646	1620.95	18.28	336	265	60	242	124	78	91	438	930	56.90%

clock cycles per synchronous cycle, *i.e.*, per iteration of the GCD while-loop.

The Pulse Width Modulator (PWM) generates pulses with a fixed period while having different duty cycles. The PSP version of this function in a 32-bit architecture can generate 32 pulses with varying cycles of duty at the same time. Our implementation demonstrates a PWM with 8-bit resolution. The synchronous cycle of our PWM uses 239 ARM cycles, which provides a minimum pulse width of $\frac{239}{48\text{MHz}} = 4.98\mu s$ and a period of $2^8 \times \frac{239}{48\text{MHz}} = 1,275\mu s$ or 784Hz.

The second group of examples are taken from cryptography [21, 32, 86, 13]. Their characteristics are summarized in Table 2.2. SIMON 128/128 is a block cipher with the Feistel structure and consists of 68 calls to the same round encryption routine. We used two different realizations of SIMON, the first one with a bit-parallel data-path and the second one with a bit-serial data-path [8]. In traditional hardware design, bit-serial methodologies are used to minimize area footprint at the expense of throughput. In the PSP execution model of software, we expect the lower gate-count of a bit-serial input specification to translate to fewer bit-wise operations in the program, and hence to a smaller code footprint. Further, we expect the bit-serial PSP design to have a lower throughput due to the lower computational effort done per synchronous clock cycle.

The first part of Table 2.3 shows that the models are small enough to fit on a simple embedded architecture. Furthermore, we observe, similar to their hardware designs, the bit-serial implementation of SIMON is 20% smaller than its bit-parallel counterpart in code size, whereas the bit-parallel version is 40× faster and has a higher throughput than the bit-serial version. The second part of Table 2.3 shows the overhead of data movements. The overhead values reported are calculated as *the number of move instructions (MOV, STR, LDR) divided by the total number of instructions.* Moving the data takes about 45-60% of the entire instructions, which is expected for a straight-line program. For comparison, the data-moving overhead for a regular (non-bitsliced) implementation of SIMON on NEON in

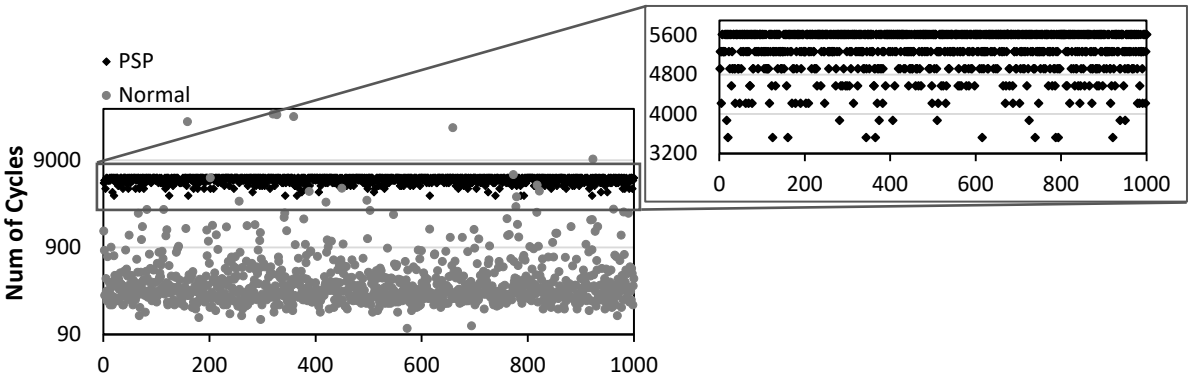


Figure 2.3: Runtime of normal and PSP implementations of the GCD algorithm on 1000 random inputs.

the SUPERCOP benchmark [1] is 34%.

We compare our PSP designs of cryptographic ciphers with their available normal implementations in Table 2.2. In the CRYPTREC lightweight project [44], SIMON-128/128 and Midori-64 ciphers are implemented in software for the RL78 16-bit microcontroller. The throughputs of the PSP implementation of these ciphers in this work are respectively almost 1.7× and 9.4× higher. PRESENT-80 is evaluated in the FELICS [54] project on ARM Cortex-M3. Even though the implementation of PRESENT-80 in FELICS uses precomputed keys, still the runtime of our PSP implementation of this cipher plus its key generation is approximately 2.6× smaller. Furthermore, to show the repeatable-timing property of PSP, we compare the runtime of the PSP and non-PSP implementations of GCD calculator for 1000 random inputs. As shown in Figure 2.3, the PSP implementation has a quantized runtime (with steps of length the runtime of one PSP function) whereas the runtime of the normal GCD function varies with an average of 580.475 and a standard deviation of 1969.29 clock cycles.

2.5 Conclusion

We presented parallel synchronous programming as a high-throughput, fixed-time model of programming, which is beneficial in safety-critical applications. We introduced an automated method for PSP code generation that can be implemented without any dependency on commercial tools. The PSP generation can be customized for the target processor to have a better performance by defining custom libraries. Finally, through examples and discussions, we demonstrated the potential of parallel synchronous software.

Chapter 3

Rewrite to Reinforce

In this chapter we present techniques to protect a binary code (without access to its source code) against fault injection attacks. This work was presented at Design Automation Conference (DAC) in 2021 [94].

3.1 Introduction

Nowadays, the Fault Injection (FI) hardware attacks are becoming more prevalent. Successful fault attacks lead to information leakage [131], [114] or privilege escalation. While fault injection is targeted at the hardware (e.g., clock glitching), consequences of the resulting faults may affect the software running on a processor. For example, ARM's secure boot can be affected by voltage glitching to enable an attacker to load controlled values in the program counter (PC) [154]. Furthermore, Vasselle et al. [158] show how laser injected faults can bypass the secure bootloader on an android smartphone.

To defend against these attacks, an extensive amount of countermeasures have been proposed that can be categorized in three groups [28], namely those that can be applied to the software source code, those that are implemented within the compiler tool-chain, and

those that are directly applied to the execution binary. The first two categories require access to the source code of the program which may not be practical in some scenarios. For instance, binary-level protection is useful for legacy binary code, or for third-party library code, or even for binary code for which the source code has been lost. In this chapter, we target the problem of applying FI countermeasures when we do not have access to the source code of the program. Knowing that applying countermeasures directly to the binary file is not easy, we demonstrate how static binary rewriting approaches help with instrumenting the program with our countermeasures. We apply and evaluate two static binary rewriting schemes; One reassembleable disassembly and the other complete translation. Using the reassembleable disassembly method, we demonstrate how we can apply simple fixes to the binary file with low overhead. Using the full-translation to LLVM-IR approach, we show how more complex countermeasures can be implemented exploiting the power of an intermediate representation (IR).

The rest of the chapter is organized as the following: In Section 3.2, we discuss the related work. In Section 3.3, we give a brief background on binary rewriting. In Section 3.4, we introduce our countermeasure insertion methodologies. In Section 3.5, we show the results of simple FI countermeasures implemented using our proposed approaches. Finally, in Section 3.6, we conclude the chapter.

3.2 Related Work

While the bulk of fault countermeasures is based in detecting faults in redundant design, researchers have made many different proposals regarding the format and abstraction level of these redundancies. Some of the related work start from the source code and add the countermeasure at that high level of abstraction. For instance, Lalande et al. [112] proposed a counter-based approach in which a counter is incremented and checked after a set number

of instructions to detect jump attacks. In other works, the countermeasures are added at compile time and require access to the source code. Barry et al. modified the LLVM back-end for insertion of countermeasures against instruction-skip fault attacks. For this purpose, they duplicated instructions based on whether the operation is idempotent [15]. Our focus is on scenarios where the source code of the program is unavailable and we need to protect the executable file against fault attacks. Given-Wilson et al. [79] propose a methodology to detect vulnerabilities of program binaries to fault injection. In their methodology, they annotate the source code with safety properties and detect vulnerabilities when these properties are shown to not be held using model-checking. Bréjon et al. [36] propose a framework consisting of symbolic execution, static analysis, and model-checking to find vulnerabilities in the binary file. Both of these works focus on finding vulnerabilities but once the vulnerabilities are found, the source code should be accessible to add corresponding countermeasures. De Keulenaer et al. [48] use the link-time optimizer tool Diablo [157] and look for patterns of instructions in the assembly-level IR that are known to be vulnerable to fault injection and replace them with hardened code. They show the approach of using the assembly-level IR results in a more compact hardened code, compared to the compiler-level IR, which is important for small embedded systems. In another work, O'sullivan et al. [130] propose a lifting and rewriting methodology based on the SecondWrite tool [5] for hardening a binary file against low-level software attacks such as buffer overflow attacks. Fault injection attacks are not considered in their work.

In this project, we propose two different approaches to find the vulnerabilities in the binary code against fault injection attacks and add corresponding countermeasures. As the first contribution of this work, we propose a simulation-driven countermeasure insertion. In this method, only the vulnerable parts of the binary file are patched and hence the overhead of the protected code is decreased compared to a full application of the countermeasure. As a second contribution, this work for the first time shows how lifting the binary to an IR can

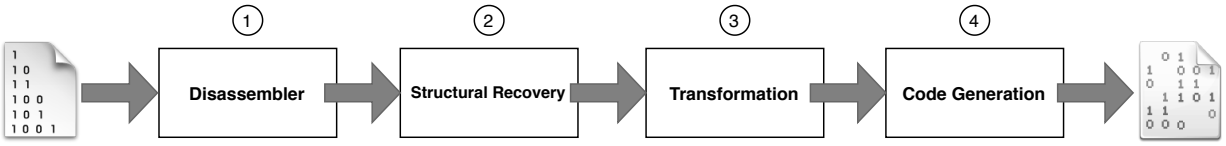


Figure 3.1: Coverage percentage achieved over for the cover sizes of different sizes

help in adding countermeasures against fault attacks. We use two different IR levels to this end and compare their resulting overhead.

3.3 Binary Rewriting

In this section, we will go through the publicly available binary rewriting solutions and compare their approach with regard to structural recovery, data type extraction, and limitations on supported architectures.

3.3.1 Definition

Binary rewriting is denoted as the process of modifying a compiled program in such a way that it remains executable and functional without having access to the source-code. There are two types of binary modifiers, static and dynamic. In the static approach, results of the modification will be stored on a persistent memory like disk for future execution. However, dynamic rewriting is applied during the program execution. In this work, we focus on static binary rewriting technique and compare the tools following this scheme.

Based on Figure 3.1, from a high-level view, in the first step (①), a binary rewriter receives a file in a binary stream format as input and passes it to the disassembler for decoding the instructions, and retrieving global variables and sections. Decoded instructions help step ② in building the control and data flow, recovering data types, and function boundaries to semantically enriching the context with metadata lost during the compilation. Transformation step (③) modifies the target binary in a way that mutated output is a

working executable.

3.3.2 Static Binary Rewriting

There are three known rewriting schemes. The oldest one is based on *detouring* at assembly level. Detouring works by hooking out the underlying instruction. There are two flavors of the detouring technique, *patch-based instrumentation* and *replica-based instrumentation*. *Patch-based instrumentation* replaces the instruction with an unconditional branch to a new section containing instrumentation, replaced instruction, and a control flow transfer back to the patch point. Detouring is a direct rewriting and is ISA dependant which makes the approach inconvenient. This approach introduces a high performance degradation given the two control transfers at patch points.

Replica-based instrumentation method inserts jump instructions to a replicated code section containing both a copy of the original code and the instrumentation. All memory references in this section are modified to maintain fewer control flow transfers between original and replicated section. While the performance of this approach is better compared to the *patch-based* instrumentation, the size of the resulting binary is noticeably increased.

Reassembleable disassembly works by recovering relocatable assembly code, the instrumentation of which could be in-lined and reassembled back to a working binary. This approach first introduced by UROBOROS [161] and then expanded by improving on top of the idea in Ramblr [160]. This approach enhances the performance since in-lined assembly avoids inserting control flow changing instructions at instrumentation points. As a result, performance penalty caused by jump instructions are alleviated in this technique.

Full-translation approach works upon translating a low-level machine code to a high-level intermediate representation (IR) using a compiler-based front-end for architecture independent binary rewriting. This process is called *lifting* the binary and assembling the IR back

to a working executable is denoted as *lowering*. The advantage of lifting the binary to a high-level IR are two fold. First, relying on IR makes the rewriting framework ISA-agnostic; Second, working on a high-level IR provides the ability to apply program analysis techniques like Value Set Analysis (VSA) [10] and optimization passes like Simple Expression Tracker (SET) and Offset Shifted Register Analysis (OSRA) [53]. On the other hand, complete translation suffers from changing the structural integrity such as cache locality and Control Flow Graph (CFG).

While each approach has its own drawbacks and benefits, we focused our evaluation on two recent research: Datalog Disassembly (Ddisasm) [66] for reassembling the disassembly and Rev.ng [53] as the candidate for full-translation.

3.3.3 Comparison of Binary Rewriters

In this section, we briefly describe the reasons behind choosing the above-mentioned two binary rewriters as our candidates. During the linking phase, linker replaces the symbolic labels with concrete memory addresses which results in losing the relocation information. Hence, to perform rewriting tasks, we need to recover the symbolic references from absolute addresses. A process which is called *symbolization*. Symbolization aims to distinguish whether an intermediate value belongs to a symbol or treat the value as a constant integer.

Comparing reassembling methods, Ramblr provided counter-examples in real-world binaries for which the UROBOROS symbol categorization fails. UROBOROS scans the data section linearly and considers any machine word-sized buffer whose integer representation falls in a memory region as a memory reference. This assumption with the compiler optimization introduces False-positive and False-negatives.

Ramblr improved the *content classification* by applying strong heuristics like *localized VSA* and *Intra-function data dependence analysis*. To improve the binary rewriting results,

Ramblr depends heavily on symbolic execution for accurate CFG recovery which slows down the rewriting process and brings up scalability issues.

Apart from Ramblr's heuristics, Ddisasm incorporated register value analysis as an alternative over traditional VSA. In addition, they introduced Data Access Pattern (DAP) analysis which is a def-use analysis combined with the results of register value analysis for a refined register value inference at any given data access point.

Rev.ng relies on full binary translation by lifting the binary to TCG (the IR used in QEMU [27]) and para-lifting TCG to LLVM-IR to benefit from more advanced transformation and analysis passes for CFG and function boundary recovery. While frameworks like angr [159] use lifting to apply more advanced binary analysis on top of the intermediate-level representation, they do not lower the resulting transformation back to the binary. Moreover, Rev.ng heavily relies on code pointers for identifying function entry points and leverages VSA for a more precise value boundary tracking.

As the rewriter tools to harden the binary code against fault injection, in this work, we chose Ddisasm and Rev.ng to show the difference between two different rewriting schemes for this purpose.

3.4 Countermeasure Insertion Methodology

For complex architectures, like x86-64, it is not straight forward to group the bits in the binary file to form full instructions. Neither is it easy to group the instructions to form basic blocks at this level of abstraction. Therefore, manipulating the binary file directly is not trivial. We propose a procedure in which we use an open source disassembler and binary manipulation tools as well as binary lifters to make the binary hardening process more manageable.

3.4.1 Rewriting the Binary

Using the disassemblers and working on the assembly code, compared to the binary file, can help in finding patterns of instructions and applying fixes locally. However, at the assembly code level, the register allocation and memory usage are fixed. Therefore, applying fixes at this level requires extra caution not to overwrite the allocated registers in use. A favorable property of this level of hierarchy is that we know which part of the assembly code corresponds to which part of the machine code exactly. We take advantage of this property in our proposed methodology and build an iterative process that, using simulation of fault effects, can locally apply countermeasures only to the parts that they are required.

While simple and small fixes can be applied at the assembly level, more complex fixes are not easily applicable. In this case, a higher level of abstraction that enables modification of the code and different types of analysis is useful. Since LLVM-IR is in Single Static Assignment (SSA) [135] format and supports different levels of hierarchy (namely module, function, basic-block, and instruction), we choose it as our high-level IR. Support of different levels of hierarchy in the IR makes it easier to perform static analyses on the program. Additionally, being a part of the LLVM tool-chain, has the advantage of being open-source, having a big number of active contributors, and a well-maintained documentation. Despite the aforementioned advantages, however, lifting the binary to such a high level of abstraction will eliminate the mapping between the abstraction levels. This results from the fact that the high level of abstraction lacks the low-level target-dependent information. Consequently, applying targeted and local fixes to the binary files at this high level of abstraction is not readily available.

In the following subsections, we discuss the mentioned approaches to countermeasure injection.

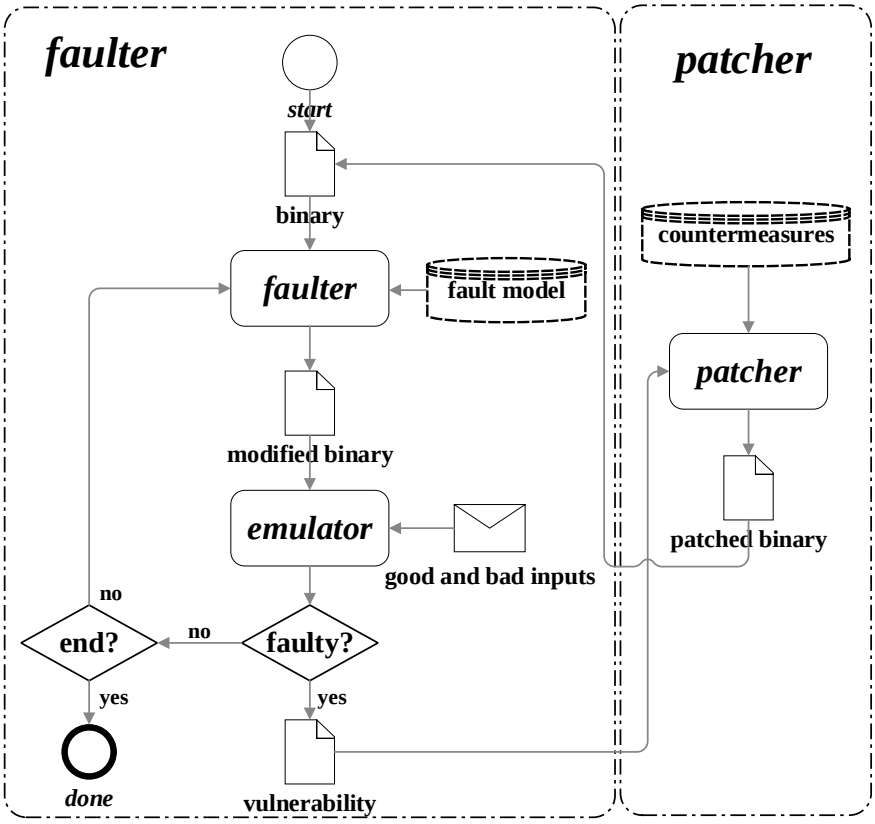


Figure 3.2: Flowchart of the *Faulter+Patcher* approach

3.4.2 Faulter+Patcher Approach

Our first approach injects countermeasures at the abstraction level of assembly code and is thus able to patch the binary file in a targeted manner. Figure 3.2 shows the overall scheme. In this approach, we have a fault simulation-driven, iterative method to mitigate fault injection vulnerabilities in the binary file. The system consists of two main components: a *faulter* and a *patcher*. The *faulter* is simulating faults under a certain fault model in a target binary and produces a list of vulnerabilities, meaning faults where unwanted behavior in the target binary is triggered. The *patcher* uses the list produced by the *faulter* to patch the binary. The *patcher* will patch each fault vulnerability as localized as possible, without affecting its surrounding code. The patched binary is then again run through the *faulter* and *patcher*. We repeat this process until no more faults are present or can be fixed. In the

following, the *faulter* and the *patcher* are discussed in detail.

Faulter

For our purpose, fault injection vulnerabilities are vulnerabilities where an attacker, i.e. an unauthorized user, is able to trigger a behavior in a target binary that should be reserved only for authorized users. For example, consider a pincode checker that receives an input pin and checks if the inserted value is correct. For a correct pin, the program will proceed to run some sensitive operations. An attacker does not know the correct pincode, but may be able to skip an instruction in the target binary such that the program will conclude the inserted pin is correct and therefore run the sensitive operations. These faults are labeled "successful faults". Faults that do not trigger the unwanted behavior or cause the program to crash are ignored.

We first choose a fault model that we want to protect our binary file against. Regardless of the fault model, and the number of faults injected per run, the *faulter* takes a target binary and two inputs; a "good" input and a "bad" input. For instance, in the pincheck example, the "good" input is the correct pincode and the "bad" input is any value other than the correct pincode. First, the "good" and the "bad" inputs are executed to see the difference of execution traces between them. When running the target binary with the "bad" input, we effectively can record a trace of all the instructions executed. For each offset in that trace, taking the "single bit flip model" as an example, we run the target binary normally up to that offset in the trace, flip a bit in the instruction at the trace offset, and then resume execution. The target binary either crashes, executes as an incorrect input, or behaves differently (as a correct input). If it behaves as a correct input, the trace offset and the fault that caused it, in this case a bit offset into an instruction at the trace offset, is recorded.

We implemented this simple *faulter* in Python using the Qiling binary emulator package. We fork each fault simulation to speed up the process. Our *faulter* supports x86-64 Linux

binaries only, but including support for other architectures supported by Unicorn should be straightforward.

Patcher

The list of "successfult faults" coming out of the *faulter* is addressed locally in the *patcher*. The *patcher* replaces the vulnerable patterns of instructions with known hardened patterns. For example, consider a run of the *faulter* under the "instruction skip" fault model that identified that at timestamp 40, the skipping of a `mov` instruction is a successful fault. A local countermeasure is to perform the `mov` twice, or add a compare instruction to verify the `mov` has been executed prior. Note that these countermeasures cause duplicate reads, as redundancy is key to mitigate fault injection attacks.

We implemented a proof of concept *patcher* based on GrammaTech's Ddisasm tool and their Python binary manipulation libraries. The Ddisasm tool performs pointer analysis on an executable and produces an IR in the form of GrammaTech Intermediate Representation for Binaries (GTIRB) [138] that can then be manipulated and recompiled into an executable by the Python GTIRB libraries.

Rinse and repeat

After running the *faulter* and the *patcher* once, we end up with a patched binary. Running the *faulter* on the patched binary may reveal that, since we added code and changed distances between instructions, we added new vulnerabilities. These new vulnerabilities then can be addressed by running the *patcher* iteratively until a fixed point is reached.

3.4.3 Hybrid Compiler-Binary Approach

In our second approach, we inject countermeasures at the abstraction level of compiler IR. To be able to implement more complex countermeasures, working at the level of assembly code is cumbersome if not impractical. For example, consider a countermeasure where extra registers are needed to hold some intermediate values. If the assembly code had been generated with a high level of optimization and no register is available in the unprotected program, extra steps should be taken to spill some data to the memory to make a few registers available and load them back to the same registers after the countermeasure. This requires knowledge of the state of the memory at different locations. However, there is no guarantee that these steps are possible therefore implementing some countermeasures on certain programs might not be feasible.

To overcome this problem, we propose a process as shown in the upper half of Figure 3.3. This process consists of three steps; First, we transform the program from binary to a compiler IR. Second, we implement the countermeasure on the IR. Last, we transform the protected IR back to the executable binary format, hence, achieve the goal of protecting the binary file. These steps are elaborated in the rest if this section.

Transforming to an intermediate representation

The goal of this step is to have a representation of the binary file which, while preserving the functionality, is easier to modify and supports a format in which different types of analysis can be performed. As discussed earlier, we choose LLVM-IR.

There are several open-source tools that are able to lift the binary file of different architectures to the architecture-independent form of LLVM-IR. In this work, we use Rev.ng. As mentioned in Section 3.3, Rev.ng is a binary analysis framework based on QEMU and LLVM. As part of this framework, it is possible to extract the LLVM-IR of a program from

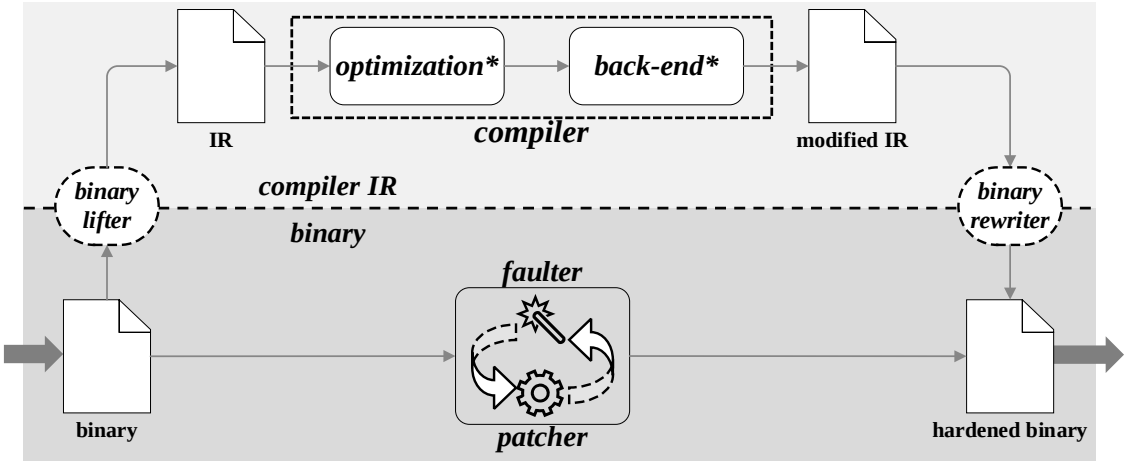


Figure 3.3: High-level overview of the *Faulter+Patcher* (lower half) and the *Hybrid* (upper half) approaches

its binary. The binary file can be for any of the x86, x86-64, ARM, MIPS, s390x, or AArch64 (WIP) architectures. In this project, without loss of generality, our focus is on the x86-64 architecture.

Implementing the countermeasure

After the lifting step is performed, we have the LLVM-IR representation of the algorithm. Therefore the countermeasure can be implemented in the form of a combination of optimization and/or back-end passes depending on the desired protection. If the protection algorithm does not have any dependencies specific to the target architecture, the entire countermeasure can be in the form of an optimization pass. Otherwise, back-end passes would be required.

Generating the protected executable

Finally, the protected IR needs to be compiled to an executable. In LLVM, the `llc` tool is responsible for translating the LLVM-IR to an architecture-specific executable file. Specific steps might need to be added in the form of back-end passes to make sure the implemented countermeasures are retained unchanged through this process. Once the hardened binary

file is generated, we use the same *faulter* system to detect remaining vulnerabilities.

3.4.4 Choosing the Right Method

The targeted insertion of countermeasures in the *Faulter+Patcher* scheme makes the overhead of the applied assurance smaller than a holistic approach. Furthermore, the mere act of lifting the binary to LLVM-IR and translating it back to binary in the *Hybrid* approach adds extra overhead to the program. This stems from the internal functions of the binary rewriting tools, which in our case is Rev.ng. On the other hand, applying countermeasures in the *Hybrid* approach is easily automated and is guaranteed to be feasible. The hierarchy levels supported by LLVM-IR as well as its SSA format eases many static analyses such as finding idempotent pieces, finding and replacing all the uses of a variable, and many more.

The aforementioned trade-off between these methods, makes each method suitable for a different use case scenario. In size-constrained applications, such as programs for small embedded systems, the *Faulter+Patcher* method is more favorable due to its smaller footprint. In scenarios where the code size is not of critical concern, the *Hybrid* approach provides a simpler and guaranteed assurance for inserting complex countermeasures.

3.5 Experimental Results

In this section, we show how our proposed approaches can apply countermeasures against a chosen fault model. We first show the local protections that we add by our *Faulter+Patcher* approach. We then demonstrate a holistic protection that can be used in our *Hybrid* approach. Finally, we show the results of the inserted countermeasures in our case studies.

Table 3.1: Local protection pattern for `mov` operations

Original	Protected

mov rax, [rbx+4]
happyflow: ...
	mov rax, [rbx+4]
	cmp rax, [rbx+4]
	je happyflow
	call faulthandler
	happyflow: ...

Table 3.2: Local protection pattern for `cmp` operations

Original	Protected

cmp rbx, [rcx+4]
fallthrough: ...
	lea rsp, [rsp-128]
	cmp rbx, [rcx+4]
	push rbx
	pushfq
	cmp rbx, [rcx+4]
	pushfq
	pop rbx
	cmp rbx, [rsp]
	je restore
	call faulthandler
	resotre:
	popfq
	pop rbx
	lea rsp, [rsp+128]
	fallthrough: ...

3.5.1 Local Protections

In the *Faulter+Patcher* approach, we are able to insert protected code patterns locally. The following is the description of these redundant computation-based protections.

Protecting `mov` Instruction

To protect the `mov` operation against fault attacks, after executing the `mov` operation, the result of the two memory locations are compared and in case of an inconsistency, a faulthandler is called (Table 3.1).

Table 3.3: Local protection for conditional jump operation

Original	Protected

j<cond>jumptarget
fallthrough: ...
jumptarget: ...
	j<cond>newjumptarget
	lea rsp, [rsp-128]
	push rcx
	pushfq
	set cl
	cmp cl, 0
	je newfallthroughjmp
	call faulthandler
	newfallthroughjmp:
	popfq
	pop rcx
	j<cond>fallthrough
	call faulthandler
	newjumptarget:
	lea rsp, [rsp-128]
	push rcx
	pushfq
	set cl
	cmp cl, 1
	je newjumptargetjmp
	call faulthandler
	newjumptargetjmp:
	popfq
	pop rcx
	j<cond>jumptarget
	call faulthandler
	fallthrough: ...
	jumptarget: ...

Protecting the `cmp` Instruction

We can protect a `cmp` instruction against fault attacks by executing the comparison twice and comparing their resulting flags (Table 3.2). To this end, we use the `pushfq` instruction in x86-64 ISA which requires a valid stack pointer (`rsp`). Due to Intel's red zone, we have to subtract 128 bytes from `rsp` to jump out of the red zone.

Protecting the `j<cond>` operation

By hardening the conditional jump operations, we detect glitches that change the jump condition. In the protected code shown in Table 3.3, we use the flags register and match this

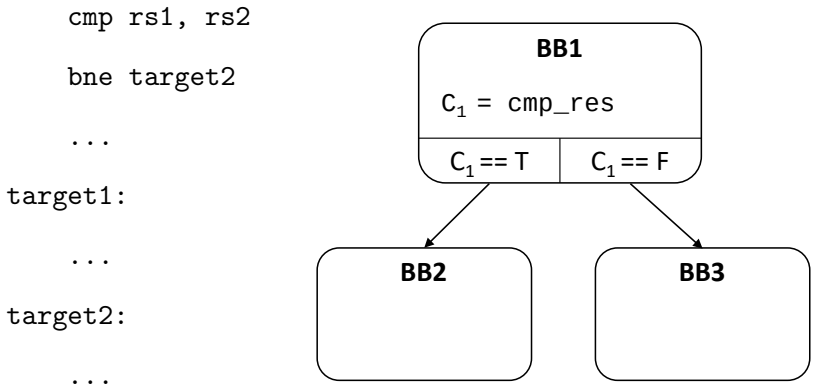


Figure 3.4: Assembly code and CFG of a simple branch instruction

to the expected flag in the jump-target and the fall-through of a branch.

3.5.2 Holistic Protection

In this section, we describe a conditional branch hardening method that we later use in our case studies for the *Hybrid* countermeasure implementation approach.

Imagine a simple program that receives a pin code and only if the pin code is correct, resumes the program to execute some operations. In the assembly code (equally the executable file) of this program, there will be a comparison instruction to compare the inserted pin code and its expected value, as well as a conditional branch that, based on the result of the comparison, jumps to a successor basic block. Figure 3.4 shows the assembly code and the control flow graph (CFG) of this branch operation.

In the case of Control Flow Integrity (CFI), going from BB1 to either of BB2 or BB3 does not raise any issues since they both are valid paths in the CFG. However, if an injected fault results in taking the wrong branch, it will be an unnoticed fault. In this conditional branch hardening method, our goal is to protect against this outcome of FI.

In this method, we assign a unique ID to each basic block (UID_{BB}) at compile time. We then use an algorithm, $h(UID_{src}, UID_{dst}, cmp\_res)$, which calculates a checksum at run-time based on the fixed UIDs assigned to the source and destination blocks at compile-time, i.e.

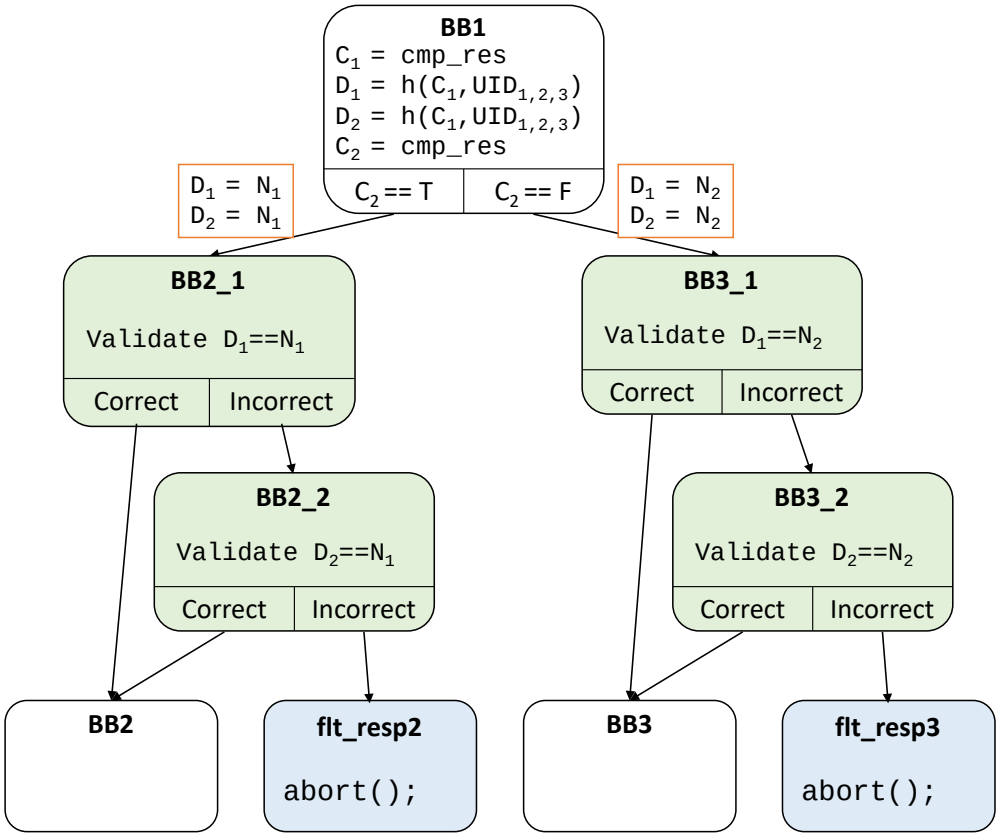


Figure 3.5: CFG of the example conditional branch hardening

UID_{src} and UID_{dst}, and the dynamically-evaluated compare result, *cmp_res*. The calculated checksum will be stored in a register and checked in the destination basic blocks. At the destination blocks, since the expected *cmp_res* for the taken edge is known, the expected value of h is known. Therefore checking the evaluated value only requires reading the register value and comparing it with the expected value. When the register does not contain the expected value, we jump to a fault-response basic block.

The simplicity level of the h function can be decided based on the required security properties of the program. As an example, we chose a simple option for h and implemented the countermeasure as an optimization pass in the LLVM tool-chain. In this example, the checksum is calculated as the XOR result of the UID of the taken destination block and that of the source block ($UID_{dst} \oplus UID_{src}$). The pseudo-code of the calculation procedure of this

checksum in LLVM is shown in Algorithm 1 where *cmp_res* is the result of the comparison for the conditional branch, and UID_{Tdst}, UID_{Fdst}, and UID_{src} are the UIDs assigned to the true destination (the destination taken when the comparison result is true), the false destination (the destination taken when the comparison result is false), and the source block respectively. The *mask* shown on line 4 will have the value of all ones if the comparison result is false and the value of all zeros if the comparison result is true. The checksum will be located in one register and each of the destination blocks will evaluate whether the value of the checksum is correct.

Furthermore, we made this evaluation duplicated; Figure 3.5 shows the CFG of this implementation protecting the conditional branch shown in Figure 3.4. We run the comparison instruction once ($\texttt{C}_1$), and calculate the checksum based on its result and keep it in a register ($\texttt{D}_1$). We perform this calculation another duplicated time and keep the result in a new register ($\texttt{D}_2$). We then perform the comparison again and run the branch based on the result of the second comparison ($\texttt{C}_2$). As shown by the orange boxes, the expected value of the checksum is different for the out-going edges of the source block ($\texttt{N}_1$ vs. $\texttt{N}_2$). In the destination blocks, we check both copies of the checksum stored in registers against the expected values in a nested fashion. The green blocks show the nested checksum validations and the blue blocks represent the fault-response. As a simple example, the fault-response can be aborting the execution (in the `flt_respx` basic blocks).

In this scenario, if the attacker tries to skip one of the comparison instructions or change it to compute the inverse output, the checksum validation process will be able to catch the injected error. If the adversary intends to bypass this process, they would need to inject the exact same fault in both comparison results. In Section 3.5.3, we show a simulated analysis of the effectiveness of this countermeasure.

The overall overhead of this countermeasure depends on the number of conditional branches that we want to protect and therefore is highly application-specific. Table 3.4

Algorithm 1 Simple example algorithm for h

Input: cmp_res, UID_{Tdst}, UID_{Fdst}, UID_{src}
Output: $checksum$
1: *Generate unique checksums for edges* :
2: $const_{Tdst} \leftarrow UID_{Tdst} \oplus UID_{src}$
3:
4: $const_{Fdst} \leftarrow UID_{Fdst} \oplus UID_{src}$
5: *Calculate checksum of the branch* :
6: $cmp\_ext \leftarrow zero\_extend(cmp\_res)$
7:
8: $mask \leftarrow cmp\_ext - 1$
9:
10: $checksum \leftarrow (\neg mask \wedge const_{Tdst}) \vee (mask \wedge const_{Fdst})$

Table 3.4: Qualitative overhead of the conditional branch hardening

Before Protection		**After Protection**	
LLVM-IR	**x86-64**	**LLVM-IR**	**x86-64**
1 cmp 1 br	1 cmp 1 jx (cond. jump)	1 cmp 2 zext 2 sub 6 xor 2 or 4 and 1 br 4 switch	2 cmp 6 mov 2 sub 6 xor 2 or 6 and 2 test 4 jx (cond. jump) 5 jmp (uncond. jump)

shows how many instructions are required to replace a simple conditional branch by this method. We implement the conditional branch hardening as an optimization pass in the LLVM compiler tool-chain. Therefore, its translation will differ for different target architectures based on how the instruction-lowering is done for that architecture in the LLVM back-end.

3.5.3 Case Studies

We choose two applications for the proof of concept of our proposed methods. The first application is a simple pin-check program that receives an input password and checks the correctness of the inserted password. The second application is a secure bootloader in which

the hash of the content of a memory location is calculated and compared with an expected hash value.

For each case study, we use the *faulter* described in Section 3.4.2 with the two fault models of "instruction skip" and "single bit flip" and verify that the code has vulnerabilities to these fault models. All of these vulnerabilities were caused by the conditional jumps (`mov`, `cmp`, and `jmp` instructions related to a jump operation) in the program. We then insert the code patterns described in Section 3.5.1 using the *Faulter+Patcher* approach for each vulnerable point and reevaluate the hardened binary file iteratively until confirmed that no more vulnerabilities exist. Furthermore, we apply the conditional branch hardening countermeasure from Section 3.5.2 using our *Hybrid* approach and verify that the vulnerabilities have been mitigated. In the case of the "instruction skip" fault model, we were able to resolve all the vulnerabilities using the mentioned countermeasures. In the case of the "single bit flip" fault model we were able to reduce the number of vulnerable points by 50% using both methodologies.

Table 3.5 shows the overhead caused by the inserted countermeasure in each approach. The overhead caused by the *Hybrid* approach is 2 to 5 times bigger than that of the iterative method. This is an expected outcome since the *Hybrid* approach applies the countermeasure to the entire program whereas the *Faulter+Patcher* approach only does so to the vulnerable points. Furthermore, duplicating every instruction, which is the go-to protection scheme against fault injection, implies at least 300% overhead in code size (since for each instruction, it will add another copy of the instruction and a comparison procedure between their results). Therefore, both of our methods perform better than a simple duplication scheme.

Table 3.5: Overhead of adding the protections

Case Study	**Overhead in code size (%)**	
	Faulter+Patcher	**Hybrid**
pincheck	17.61	85.88
secure bootloader	19.67	48.67

3.6 Conclusion

In this work, we proposed two approaches for hardening the binary file against fault attacks based on binary rewriting. In the first approach, we disassemble the binary file and insert countermeasures which enables us to insert local countermeasure and keep the structure of the original binary file. We propose an iterative and simulation-driven framework that only inserts countermeasures to the vulnerable parts of the program. In the second approach, we lift the binary file to LLVM-IR and insert countermeasures at a higher level of abstraction. This enables us to implement more complex countermeasures and perform static analyses on the program.

Chapter 4

Domain Oriented Masking Instruction Set Extension

In this chapter, we introduce an extension to RISC-V ISA to provide support for domain oriented masked software. This work was presented in the Workshop on Secure RISC-V Architecture Design (SECRISC-V) in 2020 [102].

4.1 Introduction

In recent years, side-channel analysis (SCA) attacks have gained significant notoriety in the field of computer security. In power SCA [110, 37], the attacker extracts a secret encryption key using only the power consumption of a device running the cipher. In timing SCA, the attacker exploits micro-architectural timing effects such as the last-level cache (LLC) access [84, 172, 116], speculative execution [109], and out-of-order execution [115] for the same purpose.

An important take-away from these side-channel attacks on standard processor architectures is how these attacks exploit resource implementation effects that are abstracted away

from the software programmer. Indeed, modern computer architectures exceed at layering and abstraction, and they hide the implementation details of hardware as much as possible. There are strong motivations towards such layering, such as the performance optimization, separation of the design concerns, and hiding of the design complexity. However, it is now clear that this practice also creates many new vulnerabilities. Therefore, one can argue that the root cause of such vulnerabilities is the ambiguity in the architecture specification. By only specifying an interface (such as the ISA), the implementation leaves room for optimizations that may result in security vulnerabilities. This is particularly true for the secure processor implementation. While the side-channel vulnerability of processor designs with respect to power and timing is understood, we rarely see efforts at design time to deal with the security implications of implementation effects.

This observation also holds for RISC-V [162]. The RISC-V Instruction Set Architecture (ISA) is prominently concerned with the definition of instruction functionalities and instruction types, mapping these instructions into opcodes, and so on. However, there is no discussion on how these instructions should be implemented for (side-channel) security-sensitive applications. On the other hand, since RISC-V is an open-source architecture, it is a good platform for trying out secure extensions to ultimately identify fitting secure extensions to include in the ISA. Gonzalez et al. [80] replicate Spectre attacks on the Berkeley Out-of-Order Machine (BOOM) [6] and then propose mitigation techniques for this line of attacks. Yu et al. [173] propose a data oblivious ISA extension which protects against timing SCA attacks.

In this contribution, we present a power SCA resistant ISA for RISC-V and discuss the important properties of the design. Our objective is to show that constraints in the ISA implementation can contribute to the practical side-channel security.

4.2 Related Work

In this section, we introduce preliminaries in side-channel leakage mitigation and related work in the design of instruction sets resistant against SCA.

Masking is a well-known countermeasure against power SCA. In this technique, the data is broken into uniformly distributed shares and all the operations are adjusted to work on the masked data. Masking breaks the relation between the power consumption and the (unmasked) data. Masked designs can only be broken using a side-channel attack that recombines the side-channel leakage of multiple shares. In higher-order masking, a single data item is split into more than two shares; and there is consensus that the higher the number of shares, the harder it is to exploit side-channel leakage.

Masking [117] has been employed to protect software against power SCA. Barthe et al. propose an algorithm for n^{th}-order masked implementations of multiplication providing security against power SCA of up to $(n-1)^{th}$-order [17]. For AES, Rivain et al. propose provably-secure higher-order masked algorithms [134]. However, later it was shown that the leakage model for this design is based on assumptions that are hard to achieve in practice. A careless implementer of the Rivain algorithm can still end up with a leaky design [11]. Therefore, even though in theory masking can be the ultimate solution for secure software design, when it comes to implementing these algorithms, the programmer must be well-aware of the processor implementation details to avoid unintentional leakage.

To alleviate the software designer's part in effectively applying masking to a program, Skiva [100] provides custom instructions that support masking as well as bitslicing and fault detection to provide a combination of countermeasures which can be combined in a modular fashion. However, to use Skiva, the program has to be bitsliced and the programmer should pay special attention in allocating registers for their variables.

Another effort, by De Mulder et al. [49], applies Threshold Implementation (TI) [127]

to the complete hardware implementation of a RISC-V design. They show through Test Vector Leakage Assessment (TVLA) technique [23] that their implementation provides the expected (first-order) security. Even though not discussed in the paper, the overhead of such protections is high. As an example, Nikova et al. [128] show that TI implementations of typical cryptographic functions would require a much higher number of shares to guarantee the required properties of TI.

Domain-Oriented Masking (DOM) [83] has shown to have a lower area overhead compared to TI as well as a lower need for randomness. To the best of our knowledge, DOM has not been applied to a processor before. In this work, we propose a DOM ISA for RISC-V which provides security against first-order power SCA. Our approach is a hybrid one in the sense that we do not apply DOM to the entire processor implementation, as was done with TI [49]. Instead, we propose an ISA extension which is masked and explain the implementation details. By this example, we illustrate how an ISA can adopt implementation constraints to provide better security guarantees.

4.3 Domain-Oriented Masking

DOM is a masking technique that provides security against power SCA in a hardware implementation. In DOM, like other masking schemes, the variables are broken into shares. The order of protection decides the number of shares the variables are broken into. Here, we discuss the first-order masking as it is the scheme we will apply to our ISA. The original algorithm then should be adjusted to work on the input shares and generate the output shares.

In first-order DOM, each variable is broken into two shares such that the xor result of the shares retrieves the original variable and the shares are uniformly distributed. Therefore, to generate the shares, we first generate a uniformly distributed random number, r, and

generate the shares of variable x as $A_x = x \oplus r$ and $B_x = r$. The exclusive or result of A_x and B_x retrieves the variable x. Meanwhile, since r comes from a uniform distribution, both shares are uniform.

As collision of the shares results in unmasking (and hence side-channel leakage), the main challenge of masking an overall program is to avoid collisions. DOM handles this by separating the shares as much as possible. For this purpose, DOM assigns separate domains to shares; A_x belongs to domain $\mathcal{A}$ and B_x to domain $\mathcal{B}$. Throughout an algorithm, share domains are kept completely separate. Only when absolutely necessary to be combined, shares are first remasked (refreshed) and only then they can be combined. In the following section, we discuss how we apply this technique to an ISA.

4.4 DOM ISA for RISC-V

When adding secure (side-channel resistant) instructions to an existing unprotected processor, care must be taken to not let shares collide unintentionally and cause side-channel leakage (power or EM). We apply the following two design principles.

1. Keeping the secure and the unprotected parts of the processor implementation separate from each other.
2. Protecting the secure part efficiently.

We address these two steps in the following subsections.

4.4.1 Separating protected execution from unprotected execution

Typical modern processor designs contain lots of redundant execution. For example, even operations that are not meant to be executed by an opcode are executed, and only at the end of the execute stage, the results of these operations are discarded (by not being stored). Since these processors are implemented in Complementary Metal–Oxide Semiconductor (CMOS)

technology, any logic operation on the chip contributes to the power consumption. When it comes to power SCA, we need strict control over the flow of information, including the flow of secret shares. For instance, it has been shown how the rotation instruction on a share-sliced design can cause unwanted leakage [69]. If the circuitry for the rotation instruction is available in the unprotected datapath, and the data is share-sliced, without the means of disabling the unprotected pipeline, the power consumption of this calculation will be present and will contribute to the power leakage correlated with the secret data even if the result of this instruction is not committed.

To implement a secure instruction set, we propose that a separate protected datapath is created in the processor exclusively to support those secure instructions. The protected datapath co-exists with the normal datapath, but is strictly separate from it, as shown in Figure 4.1. The instruction under execution should be evaluated to activate either the secure or the unprotected pipeline, therefore, the designer needs to build a circuit following the decode stage to determine whether the decoded instruction is a secure one or not. Based on the output of this circuit, either the secure pipeline or the normal pipeline is activated. In Figure 4.1, a comparison circuit is added in the execute stage (coming right after the decode stage) that disables/enables the corresponding parts of the datapath.

4.4.2 Protecting the secure instructions

In this work, we implement a small but universal set of instructions. We build a protected ALU to support them. We protect the instructions using the DOM technique. In this work we focus only on the ALU part of the CPU and assume all the other parts are already protected; the register file is duplicated such that a separate register file is used for each share domain, registers in the secure datapath which contain instruction operands are duplicated the same as the register file, and all the `load` and `store` operands are refreshed to avoid accidental

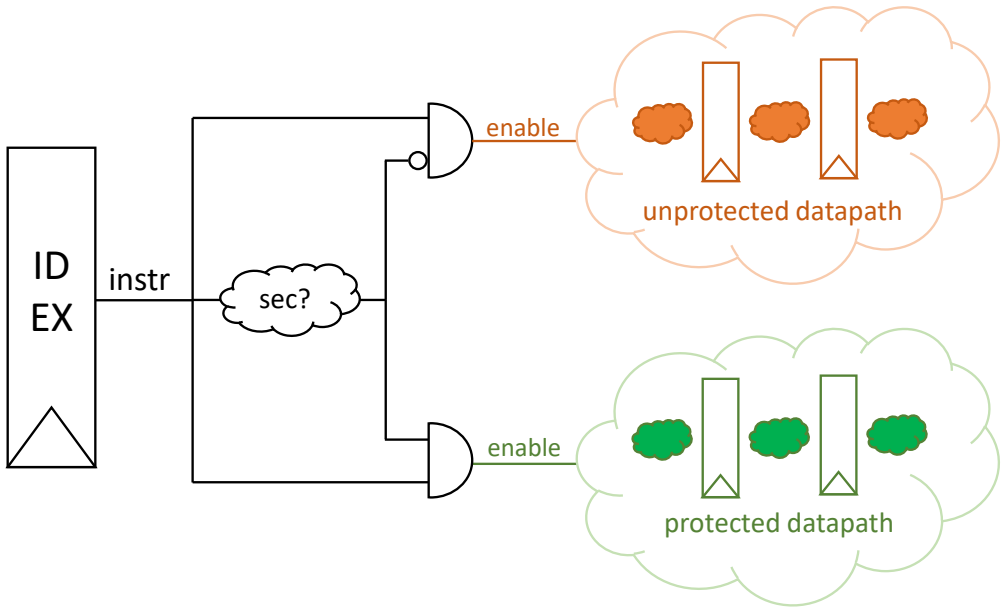


Figure 4.1: Separating the datapath for protected instructions from the unprotected datapath.

leakage through memory accesses. Additionally, for systems with data-cache support, the caches are separate for each share domain.

DOM works on the concept of share domains; each variable is divided into shares and the goal is to keep the shares of each domain separate from and independent of one another. In this work, we operate on two domains, i.e. domain $\mathcal{A}$ and domain $\mathcal{B}$. Therefore, each variable is broken into two shares to be protected against first-order power SCA according to the d-probing model [89]. Operations are divided into two categories; linear and non-linear. Linear operations preserve the uniformity of their inputs for their outputs, which is not the case for non-linear operations. As is mentioned by Nikova et al. [127], in linear operations, each share of the output only depends on one share of each input, therefore, in DOM implementation, there is no need for special attention as the separation is naturally provided. This is not the case for non-linear operations and special steps should be taken for their DOM implementation. We use the DOM-*dep* concept (viz. [83]) in which the inputs of an operation are not required to be independent of each other. Throughout this section, we

Table 4.1: DOM implementation of AND instruction.

instruction	$x \cdot y$	
domain	$\mathcal{A}$	$\mathcal{B}$
cycle 1 (Z_0 req'd)	$A_{t1} = A_x \cdot A_y$ $A_{t2} = B_y \oplus Z_0$ $A_{t3} = A_x \cdot Z_0$	$B_{t1} = B_x \cdot B_y$ $B_{t2} = A_y \oplus Z_0$ $B_{t3} = B_x \cdot Z_0$
cycle 2 (Z_1 req'd)	$A_q = A_{t1} \oplus A_x \cdot A_{t2} \oplus A_{t3} \oplus Z_1$	$B_q = B_{t1} \oplus B_x \cdot B_{t2} \oplus B_{t3} \oplus Z_1$

show the shares belonging to the domain $\mathcal{A}$ in blue, domain $\mathcal{B}$ in red, and neutral variables in green. The universal set of instructions that we choose are enumerated in the following.

NOT As a linear instruction, $q = \sim x$ is implemented as below and executed in one clock cycle:

$$A_q = \sim A_x, \quad B_q = B_x$$

XOR As another linear operation, $q = x \oplus y$ is implemented as follows and executed in one clock cycle:

$$A_q = A_x \oplus A_y, \quad B_q = B_x \oplus B_y.$$

AND *AND* is a non-linear operation. The DOM-*dep* implementation of $q = x \cdot y$ is:

$$A_q = A_x \cdot A_y \oplus A_x \cdot (B_y \oplus Z_0) \oplus A_x \cdot Z_0 \oplus Z_1,$$

$$B_q = B_x \cdot B_y \oplus B_x \cdot (A_y \oplus Z_0) \oplus B_x \cdot Z_0 \oplus Z_1;$$

where Z_0 and Z_1 are random bits (to see the justification of these algorithms refer to [83]). To avoid unintentional leakage through glitches, we need to insert registers in the middle of the calculation of these algorithms; this ensures the correct sequence of the operations (remasking first and calculating across domains next). In the realm of processor instructions, this results in a two-cycle instruction as shown in Table 4.1.

Table 4.2: DOM implementation of OR instruction.

instruction	$x + y$	
domain	$\mathcal{A}$	$\mathcal{B}$
cycle 1 (Z_0 req'd)	$A_{t1} = A_x \cdot A_y$ $A_{t2} = B_y \oplus Z_0$ $A_{t3} = A_x \cdot Z_0$	$B_{t1} = B_x \cdot B_y$ $B_{t2} = A_y \oplus Z_0$ $B_{t3} = B_x \cdot Z_0$
cycle 2 (Z_1 req'd)	$A_q = A_x \oplus A_y \oplus A_{t1} \oplus A_x \cdot A_{t2} \oplus A_{t3} \oplus Z_1$	$B_q = B_x \oplus B_y \oplus B_{t1} \oplus B_x \cdot B_{t2} \oplus B_{t3} \oplus Z_1$

OR We derive the DOM-*dep* implementation of *OR* in terms of *XOR* and *AND* as mentioned above, $q = x + y = (x \oplus y) \oplus (x \cdot y)$, which results to:

$$A_q = A_x \oplus A_y \oplus A_x \cdot A_y \oplus A_x \cdot (B_y \oplus Z_0) \oplus A_x \cdot Z_0 \oplus Z_1,$$

$$B_q = B_x \oplus B_y \oplus B_x \cdot B_y \oplus B_x \cdot (A_y \oplus Z_0) \oplus B_x \cdot Z_0 \oplus Z_1;$$

where Z_0 and Z_1 are random bits. Similar to AND, OR also takes two cycles to execute as shown in Table 4.2.

ADD From the implementation of *AND* and *OR*, we can conclude that the number of cycles for the execution of an instruction depends on the multiplicative complexity [35] of the instruction. Following the implementation of a Carry Look-Ahead Adder with inputs X, Y, and C (carry) where the concepts of *carry propagate* (P) and *carry generate* (G) are $P_i = X_i \oplus Y_i$, and $G_i = X_i \cdot Y_i$, and the sum (S) and carry-out (C) are calculated as $S_i = P_i \oplus C_i$ and $C_{i+1} = G_i + P_i \cdot C_i$, we find that the multiplicative complexity for the carry-out of an n-bit adder is $2n$, therefore, taking $4n$ clock cycles to run. Hence, using secure *ADD* instructions causes significant drops in the performance.

The alternative would be for the software programmer to make a binary (Boolean) implementation of the entire program, avoiding the usage of any *ADD* instruction. This will not necessarily have a better performance than using an *ADD* instruction and it should be decided for each application separately.

Bitslicing [29] is a type of programming common in secure software design where all the data in the program running on a w-bit wide architecture are transposed into w 1-bit values. For this type of programming, it could be helpful to have an instruction for a 1-bit adder (taking 4 clock cycles to run). The two shares of the carry-out can be stored in two special registers in the processor dedicated to the carry-outs of the ADD instruction. Hence, implementing ADD instruction requires two special registers, A_c and B_c, to be added to the processor. The ADD instruction then reads the contents of these registers as the carry-in at the first cycle of its execution and updates it with the result of carry-out at its last execution cycle. In this project, we opt for a 1-bit ADD instruction to calculate the sum (S) and carry-out (C_o) as $S = x \oplus y \oplus c_i$ and $C_o = (x \oplus y) \cdot c_i + x \cdot y$. The DOM implementation of S calculates

$$A_S = A_x \oplus A_y \oplus A_{c_i}, \quad B_S = B_x \oplus B_y \oplus B_{c_i};$$

both of which can be calculated in the same clock cycle. To show the DOM implementation of C_o, we define $z = x \oplus y$, $a = z \cdot c_i$, and $b = x \cdot y$. Therefore, we have $C_o = a + b$ and the DOM implementation of C_o calculates

$$A_a = (A_x \oplus A_y) \cdot A_{c_i} \oplus (A_x \oplus A_y) \cdot (B_{c_i} \oplus Z_0) \oplus (A_x \oplus A_y) \cdot Z_0 \oplus Z_1,$$

$$B_a = (B_x \oplus B_y) \cdot B_{c_i} \oplus (B_x \oplus B_y) \cdot (A_{c_i} \oplus Z_0) \oplus (B_x \oplus B_y) \cdot Z_0 \oplus Z_1,$$

$$A_b = A_x \cdot A_y \oplus A_x \cdot (B_y \oplus Z_2) \oplus A_x \cdot Z_2 \oplus Z_3,$$

$$B_b = B_x \cdot B_y \oplus B_x \cdot (A_y \oplus Z_2) \oplus B_x \cdot Z_2 \oplus Z_3,$$

$$A_{C_o} = A_a \oplus A_b \oplus A_a \cdot A_b \oplus A_a \cdot (B_b \oplus Z_4) \oplus A_a \cdot Z_4 \oplus Z_5,$$

$$B_{C_o} = B_a \oplus B_b \oplus B_a \cdot B_b \oplus B_a \cdot (A_b \oplus Z_4) \oplus B_a \cdot Z_4 \oplus Z_5.$$

The correct execution sequence of these operations is as shown in Table 4.3.

Table 4.3: DOM implementation of ADD instruction.

instruction	$(x \oplus y) \cdot c_i + x \cdot y$	
domain	$\mathcal{A}$	$\mathcal{B}$
cycle 1 (Z_0, Z_2 req'd)	$A_{t1} = A_x \oplus A_y$ $A_{t2} = B_{c_i} \oplus Z_0$ $A_{t3} = (A_x \oplus A_y) \cdot Z_0$ $A_{t4} = B_y \oplus Z_2$ $A_{t5} = A_x \cdot Z_2$	$B_{t1} = B_x \oplus B_y$ $B_{t2} = A_{c_i} \oplus Z_0$ $B_{t3} = (B_x \oplus B_y) \cdot Z_0$ $B_{t4} = A_y \oplus Z_2$ $B_{t5} = B_x \cdot Z_2$
cycle 2 (Z_1, Z_3 req'd)	$A_a = A_{t1} \cdot A_{c_i} \oplus A_{t1} \cdot A_{t2} \oplus A_{t3} \oplus Z_1$ $A_b = A_x \cdot A_y \oplus A_x \cdot A_{t4} \oplus A_{t5} \oplus Z_3$	$B_a = B_{t1} \cdot B_{c_i} \oplus B_{t1} \cdot B_{t2} \oplus B_{t3} \oplus Z_1$ $B_b = B_x \cdot B_y \oplus B_x \cdot B_{t4} \oplus B_{t5} \oplus Z_3$
cycle 3 (Z_4 req'd)	$A_{t6} = B_b \oplus Z_4$ $A_{t7} = A_a \cdot Z_4$	$B_{t6} = A_b \oplus Z_4$ $B_{t7} = B_a \cdot Z_4$
cycle 4 (Z_5 req'd)	$A_{C_o} = A_a \oplus A_b \oplus A_a \cdot A_b \oplus A_a \cdot A_{t6} \oplus A_{t7} \oplus Z_5$	$B_{C_o} = B_a \oplus B_b \oplus B_a \cdot B_b \oplus B_a \cdot B_{t6} \oplus B_{t7} \oplus Z_5$

Mapping to opcodes

All the presented instructions are of register-register type (*R-type*). To be compatible with the current and future states of RISC-V, we map these instructions to the *custom-0* opcode field (0001011) which will be avoided by the future standard extensions of the 32-bit format. Furthermore, separating the opcode of our secure extension from other instructions will make the secure comparator shown in Figure 4.1 simpler.

Recap

In this work, we studied the design principles for our ISA thwarting power based SCA. The proposed small ISA extension, as an example to show how ISAs can be extended to contain implementation details and requirements. For this case, where the attack model is power-based SCA, DOM ISA specifies the following:

1. constraints on the flow of information in the system,
2. break-down of operations into sub-operations with constraints on their execution order,
3. constraints on the required number of random bits in each execution clock cycle.

Following the proposed ISA, the designers know the security requirements in implementation; they know implementing this ISA requires the register file to be duplicated, they also know

they require a random number generator with the rate of two random bits per clock cycle (for the *ADD* instruction). This way, the gap between the ISA definition and its physical implementation is reduced.

4.5 Conclusion

We discussed how the current myriad of SCA attacks are caused in part by the ambiguity of the processors' design and how it can be beneficial to include secure design details in the ISA of processors to avoid SCA attacks after implementation. To give an example, we proposed an Instruction Set Extension for RISC-V which uses Domain-Oriented Masking to provide security against first-order power SCA. Our ISE contains implementation considerations that can bring closer the ISA and implementation of RISC-V.

Chapter 5

Skiva-V: Architecture Support for Bitslicing

In this chapter, we introduce an instruction set extension for both 32 and 64 bit RISC-V ISAs to support countermeasures against combined hardware attacks. We furthermore integrate a DMA module into Skiva-V SoC to make the data transposition and movements more efficient in bitsliced programs. This work is under review at IEEE Transactions on Emerging Topics in Computing [95].

5.1 Introduction

Bitslicing was first introduced as a programming model to boost the throughput of the software implementation of the Data Encryption Standard (DES) cryptographic algorithm [29]. Since then, researchers have explored applications that can benefit from this model of programming in security [132, 47, 45, 121] and dynamic word-length computation [164, 145] among others.

Bitslicing is a software technique, and as such it does not require any changes to the

underlying design of the processor. However, bitsliced programs bear significant memory spills due to their extensive amount of live registers [93]. Therefore, hardware support for bitslicing can lead to a significant increase in performance of various bitsliced applications.

Today, many digital circuits have a SoC architecture. In such systems, hardware support for bitslicing can be in the form of instruction extension in the processor implementation or it can be a hardware module accessible by the processor through a bus. Our goal in this work is to integrate both of these types of hardware support for bitslicing into an SoC. Even though our focus is mostly on security applications, non-security related applications of bitslicing can equally benefit from part of our proposed hardware extensions. For example, bitslicing can support custom-precision computations on data [164] and can speed up multiplications used in software-based neural networks [145]. All bitsliced applications (whether or not related to security) require frequent transposition of data. We present two methods to speed up this transposition: First, based on using custom instructions and second, based on using a dedicated hardware module.

As the open-source RISC-V Instruction Set Architecture (ISA) is gaining more attention both in research as well as in industry, domain-specific Instruction Set Extensions (ISEs) are becoming more and more relevant [175, 153, 102]. In our previous work [100], we proposed SKIVA, a 32-bit ISE for the SPARC V8 ISA. SKIVA supports protection against a combination of active and passive physical attacks, i.e., power SCA, fault injection, and timing SCA. These protections are in the form of masking [117], redundant computation, and bitslicing.

In this work, we present the following contributions to further the level of hardware support for bitslicing.

1. We port the ISE in SKIVA to RISC-V and call it Skiva-V[1]. Additionally, we propose the 64-bit version of Skiva-V which supports extra security-related modes and add the introduced instructions to the RISC-V GCC assembler.

[1]We will open-source the design files and the modified GCC compiler before the paper's publication.

2. We compare the proposed Skiva-V ISE with the newly proposed bit-manipulation ISA for RISC-V (RV32B, RV64B)[2].
3. To support programming of Skiva-V, we rely on parallel synchronous programming (PSP) [93]. We port a compiler for PSP to the newly proposed instructions, and compare the performance of auto-generated bitsliced codes for masked implementations of light-weight ciphers with their corresponding implementations in the literature.
4. Finally, we propose a Direct Memory Access (DMA) module, called T-DMA, which is capable of transposing data as part of a memory block transfer. This capability of T-DMA in itself shows how an extra-processor support for bitslicing can be beneficial for any bitsliced implementation. However, we further tune this module to add support for our security-related programming needs, namely on the fly masking and redundancy generation/checking.

Our focus in this work is on the performance analysis of the proposed instructions. For the security analysis of the custom instructions in SKIVA, we refer the reader to our previous work [100]. Furthermore, we note that several authors have proposed a security analysis for similar bitsliced masked software [82, 47, 51]. For our implementation-based evaluations, we focus on the 32-bit version of Skiva-V instructions as the representative architecture to highlight the advantage of the proposed instructions in an SoC.

The rest of the paper is as follows: Section 5.2 gives an overview of the concepts underlying the proposed system. Section 5.3 describes the definition of the custom instructions in Skiva-V, their ISA-level performance analysis, and implementation footprint. Section 5.4 demonstrates how to generate bitsliced programs for Skiva-V. Section 5.5 presents our proposed DMA module with support for transposing, masking, and duplicating the data. It further describes its functionality, design, and synthesized implementation footprint. Section 5.6 describes the integration of Skiva-V processor core and T-DMA into an SoC archi-

[2] `https://github.com/riscv/riscv-bitmanip`

tecture. Section 5.7 demonstrates benchmarks to emphasize the impact of hardware support in performance of bitsliced and masked software. Finally, Section 5.8 concludes the paper.

5.2 Preliminaries

In this section, we provide preliminaries on the underlying programming concepts in this work. We describe Bitslicing, Masking and Redundant Computation, while the reader already familiar with these techniques can skip ahead to Section 5.3

5.2.1 Bitslicing

Bitslicing, first introduced by Biham [29], is a technique originally proposed to increase the throughput of a program by running multiple instances of a code in parallel. In bitslicing, all the variables are transposed so that each register contains only one bit of the variable. For example, if a variable is 32 bits wide, in bitsliced program it will reside in 32 different registers and use one bit of each. Each register of width ω then will have the capacity to hold one bit of ω different variables. Consequently, the program needs to be adjusted to work on one bit of its variables at a time. This implies that the adjusted (i.e., bitsliced) program can only contain bit-wise logic operations. Therefore, the bitsliced program will be capable of ω parallel computations.

A fully-bitsliced program needs to be flattened (no branches). In a flattened program, the run-time of the program is known and data-independent. This property of bitslicing benefits the security-sensitive programs as it averts timing side-channel leakage (i.e., correlation between the run-time and the internal data of a program). Furthermore, bitslicing provides a proper base to combine our masking and redundant computation schemes as described in the next section.

5.2.2 Masking

Power-based SCA [110, 37], as a subset of active physical implementation attacks, has shown vulnerabilities in the implementation of algorithms which are expected to be secure at the algorithm level. In power SCA, the correlation between the power consumption and the internal data is exploited to find information about the processed data. A widely-adopted countermeasure against this type of attack is *masking* which tries to break this correlation.

In masking, each signal or variable is divided into *shares* that are independent from the original data. The number of shares depends on the masking scheme. In the d^{th}-order masking scheme, each data bit is divided into $d + 1$ shares. Knowing any strict subset of these shares will not disclose any information about the original data, while knowing all of the shares can reproduce the original data. A simple way to generate these shares is by applying *Boolean masking*. For instance, in Boolean masking for the 1^{st}-order masking scheme, a random bit r is generated (from a uniform distribution) per each original bit b. The shares of the bit b will be computed as the tuple $(b \oplus r, r)$ where $\oplus$ is the exclusive-or operation. Knowledge about one share (either r or $b \oplus r$) will not give any information about the original data b, however, by knowing both of these shares the original data can be disclosed as the exclusive-or result of the two shares $((b \oplus r) \oplus r = b)$.

Once each data is broken into independently-distributed shares, the algorithm should be modified to work on the shares of the inputs and the intermediate data to generate the shares of the outputs. The operations in the algorithm are categorized into *linear* and *non-linear* operations. An operation is linear if a uniform distribution of its inputs results in a uniform distribution for its outputs. Masking is then applied to each operation according to its linearity. In a linear operation, each share of the output can be implemented as a function of at most one share of each input. This property, however, does not hold for non-linear operations and there exists a vast body of research on how a non-linear operation can be

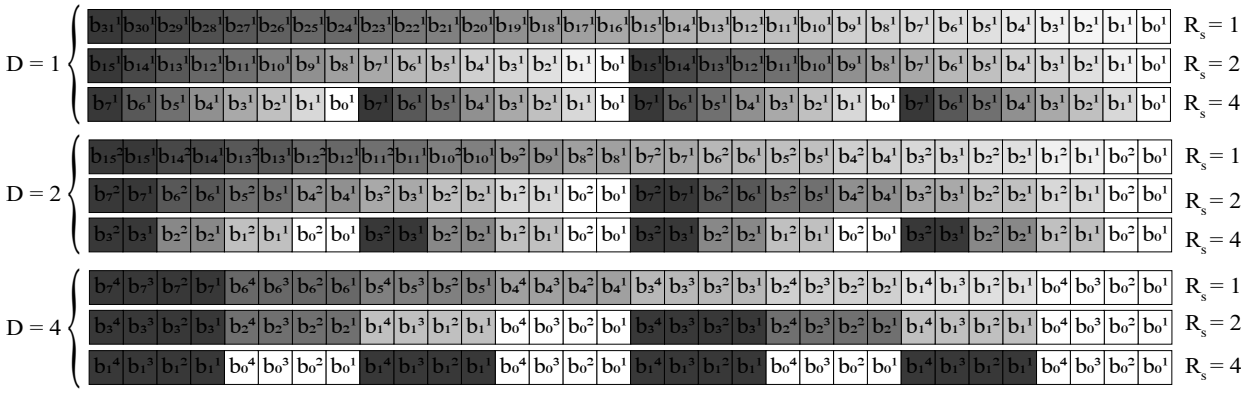


Figure 5.1: Bitsliced data representation on 32-bit registers. b_i^j represents j^{th} share of data b_i. Shares of the same variable have the same color.

masked [17, 18, 25].

In this work, we break every algorithm into a combination of operations from the set {`XOR`, `XNOR`, `AND`, `NOT`}. Since this set of operations is functionally complete, every operation in the algorithm can be written as a combination of these operations. The `AND` operation is therefore the only non-linear operation that can appear in the adjusted algorithm. We follow the parallel masked multiplication method proposed by Barthe et al. [17] for our `AND` operation and a normal masked implementation for our linear operations.

5.2.3 Redundant Computation

Fault injection [34] is another type of implementation attack. Redundant computation is a technique to detect whether a fault has been injected in a circuit. In this technique, every computation is done multiple times and the results are compared. A mismatch between the results shows that a fault has happened. For n number of redundant computations, the occurrence of up to $n-1$ faults can be detected.

In Skiva-V, our goal is to combine countermeasures against both fault injection and power SCA attacks. As shown in our previous work [100], when the redundant copies of the data are in complementary format, the intensity of power side-channel leakage is decreased.

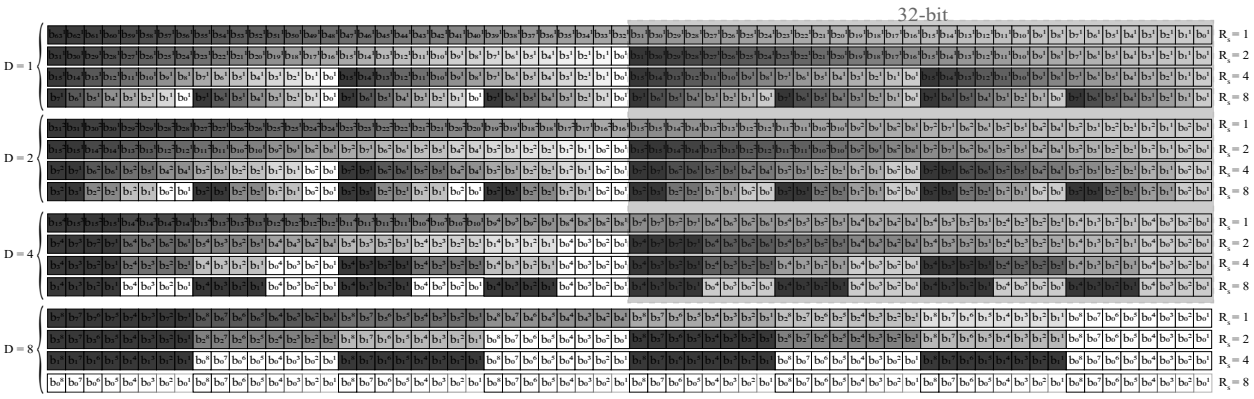


Figure 5.2: Bitsliced data representation on 64-bit registers. b_i^j represents j^{th} share of data b_i. Shares of the same variable have the same color. Parts enclosed in dashed lines show the nine possible configurations in the 32-bit architecture proposed in SKIVA [100] also shown separately in Figure 5.1.

Therefore we support redundancy of two types: direct and complementary. In direct redundancy, the redundant data is a direct (uninverted) copy of the original data, whereas, in the complementary redundancy, half of the redundant copies will be in the inverted format to balance the power consumption of the direct copies. Similar to SKIVA[100], the instructions in Skiva-V can be used for both spatial and temporal redundancy to detect data faults and control faults respectively. The spatial redundancy mode can be integrated into any bitsliced program, whereas the temporal redundancy can be used in round-based applications where half of the slices will be performing round i and the rest of the slices will be performing round $i + 1$.

5.3 Processor Support

We present an instruction set extension (ISE) for both the 32-bit and the 64-bit RISC-V ISAs called Skiva-V. The underlying data representations of Skiva-V are based on the masking order (D) and the spatial redundancy (R_s). In our 32-bit ISE, we support nine different configurations chosen from the sets $\mathcal{D} = \{1, 2, 4\}$ and $\mathcal{R}_s = \{1, 2, 4\}$. For our 64-bit ISE, we

extend the 32-bit representations to add additional masking and redundancy modes. In this new configuration, Skiva-V supports sixteen different configurations from $\mathcal{D} = \{1, 2, 4, 8\}$ and $\mathcal{R}_s = \{1, 2, 4, 8\}$. In all of these configurations, the D shares of the same variable reside in the adjacent bits of a register. Next to the shares of one variable, will sit the shares of the next variable for parallel computation. This pattern repeats in the same register R_s times for redundant computation. Thus in each (D, R_s) configuration for the N-bit architecture, Skiva-V supports $p = \frac{N}{D \times R_s}$ parallel computations. Figure 5.1 and Figure 5.2 show all the possible configurations in the 32-bit and 64-bit ISEs respectively. In these figures, the i subscripts in b_i data bits show different variables in parallel computation. To support both direct and complementary redundancy, the even-numbered redundant copies can be either inverted or direct.

In the rest of this section, we describe the instructions in Skiva-V, their implementation details and footprint, and how programmers can employ them in their codes.

5.3.1 Instruction Definitions

Our proposed instruction set extension for RISC-V is divided into three groups: instructions for bitsliced transposition, instructions for masked implementation, and instructions for redundant computation. In the following subsections, we describe each instruction. Table 5.1 shows the assigned opcodes and formats of the instructions in Skiva-V. The instructions' encodings in Skiva-V follow the RV32I base r-type and i-type instruction formats mentioned in RISC-V ISA manual [7]. Each of the i-type instructions in Skiva-V has its own immediate encoding that clarifies the masking order (required for `subrot` instruction) or the redundancy scheme (required for `redl/h` and `ftchk`). We describe the immediate assignment of each instruction with their definition in the rest of this section. The operations for transposing data and duplicating the data according to the redundancy scheme require two destination

Table 5.1: Opcode assignments in Skiva-V

Instruction	Type	funct7 (instr 31-25)	funct3 (instr 14-12)	opcode (instr 6-0)
subrot	i-type	—	0x0	0x0b (custom-0)
redl	i-type	—	0x1	0x0b (custom-0)
redh	i-type	—	0x2	0x0b (custom-0)
ftchk	i-type	—	0x3	0x0b (custom-0)
andc32 (*only in 64-bit ISA*)	r-type, logic	0x20	0x4	0x0b (custom-0)
andc16	r-type, logic	0x10	0x4	0x0b (custom-0)
andc8	r-type, logic	0x00	0x4	0x0b (custom-0)
xorc32 (*only in 64-bit ISA*)	r-type, logic	0x21	0x4	0x0b (custom-0)
xorc16	r-type, logic	0x11	0x4	0x0b (custom-0)
xorc8	r-type, logic	0x01	0x4	0x0b (custom-0)
xnorc32 (*only in 64-bit ISA*)	r-type, logic	0x22	0x4	0x0b (custom-0)
xnorc16	r-type, logic	0x12	0x4	0x0b (custom-0)
xnorc8	r-type, logic	0x02	0x4	0x0b (custom-0)
tr2l	r-type, transposition	0x00	0x5	0x0b (custom-0)
tr2h	r-type, transposition	0x10	0x5	0x0b (custom-0)
invtr2l	r-type, transposition	0x01	0x5	0x0b (custom-0)
invtr2h	r-type, transposition	0x11	0x5	0x0b (custom-0)

registers. However, multiple destination registers are not by default supported by RISC-V ISA. Therefore, similar to the multiplication instructions, e.g., `MUL` and `MULH`, in RISC-V's "M" extension, we assign two different instructions for the lower and upper halves of the result of `redl/h`, `tr2l/h`, and `invtr2l/h` operations.

Bitsliced transposition

We propose two instructions, i.e. `tr2l rd, rs1, rs2` and `tr2h rd, rs1, rs2`, which, if applied iteratively in the butterfly pattern, can transpose the data from normal representation to its bitsliced format. These instructions take two source registers and reorder their bits in the destination register interchangeably. Instruction `tr2l` reorders the lower half of the source registers while instruction `tr2h` reorders the upper half. To transpose the bitsliced data back to its normal representation, we proposed the inverse of the above instructions, i.e. `invtr2l rd, rs1, rs2` and `invtr2h rd, rs1, rs2`. Figure 5.3 shows how these instructions work. As an example, Figure 5.4 shows how four 4-bit registers can be transposed to their bitsliced positions in two iterations of applying these instructions. In general, for N N-bit registers, it takes $\log_2(N)$ iterations of applying the transposition instructions to

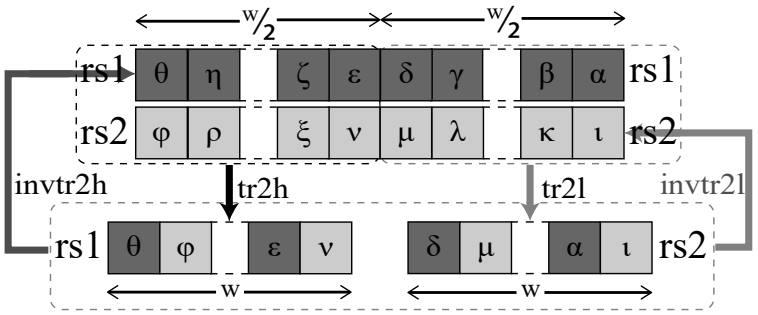


Figure 5.3: (inv)tr2h and (inv)tr2l instructions. W represents the length of the registers which can be either 32 or 64 bits. All four instructions take two input registers and store the results in the destination register.

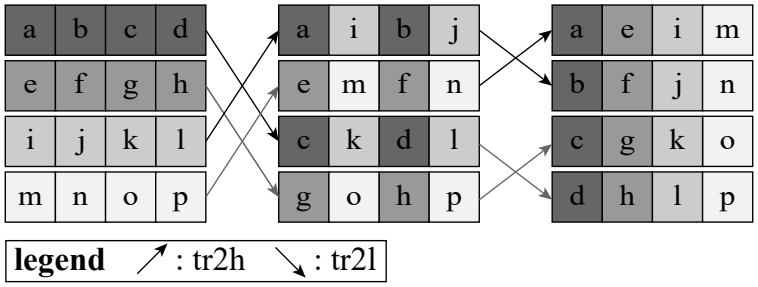


Figure 5.4: Applying tr2l and tr2h instructions to four 4-bit registers iteratively in a butterfly pattern to transpose the bits for bitsliced implementation. To transpose the bits back to their initial positions, we can apply invtr2l and invtr2h from right to left.

completely transpose the bits.

Masked implementation

In our masked data manipulations, we follow the parallel masked multiplication gadget by Barthe et al. [17]. In this gadget, the shares of a variable are adjacent in a register and during the calculations, we need to rotate the adjacent shares. Rotating parts of a register independently is not part of the RISC-V ISA, however, in our masking schemes will be executed quite often. Hence we add this instruction to Skiva-V. In our 32-bit (resp. 64-bit) representation, Skiva-V supports masked implementation with 2 and 4 (resp. 2, 4, and 8) shares. Therefore we need to be able to rotate 2 and 4 (resp. 2, 4, and 8) consecutive bits in a 32-bit (resp. 64-bit) register. Therefore, we add an instruction (`subrot rd, rs1, imm`)

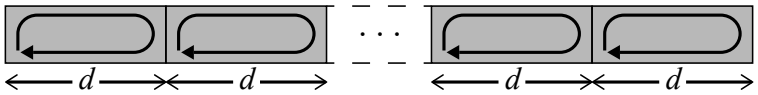


Figure 5.5: `subrot` instruction. This instruction rotates d adjacent bits in a register where d is decided from the immediate input and follows the masking scheme ($d \in \{2, 4\}$ for 32-bit ISA, $d \in \{2, 4, 8\}$ for 64-bit ISA).

which takes a source register and an immediate value. If the immediate value is 2/4(/8 in 64-bit ISA) respectively 2/4(/8 in 64-bit ISA) consecutive bits will be rotated. Figure 5.5 shows how this instruction works.

Note. When using the `subrot` instruction, one must be careful not to use the same register for both the input and the result (i.e. `rs1` $\neq$ `rd`) since this will result in overwriting the shares of the same variable and transiently reducing the intended order of masking scheme. Fortunately, compilers support this type of criteria in their code generation process and we can ensure this property will be held by adding it to the back-end of the compiler (code generator) as a criteria specific to the `subrot` instruction.

Redundant computation

As mentioned in Section 5.2, Skiva-V supports both direct and complementary redundant computations. Direct redundancy enables fault detection while complementary redundancy also reduces the intensity of power side-channel leakage. To prepare data for redundant representation, we introduce instructions `redl rd, rs1, imm` and `redh rd, rs1, imm` to copy data (both directly and in inverted manner) in the same register. The immediate field in these instructions decides which part of the input register has to be copied and whether it should follow direct redundancy or complementary redundancy. Table 5.2 shows the immediate value assignment for each redundancy mode.

To demonstrate in more detail how the bits are duplicated in the destination register, Figure 5.6 demonstrates the result of `redh` and `redl` instructions when their immediate value

Table 5.2: Immediate value assignment for `redh`/`redl` instructions. W represents the word length (32 for the 32-bit and 64 for the 64-bit ISA). Source bits signifies which bits in the source register are being replicated.

Redundancy (R_s)	**Source bits**	**`redh`/`redl` imm.**
2 direct	W-1:0	2
2 compl.	W-1:0	3
4 direct	W/2-1:0	4
4 compl.	W/2-1:0	5
4 direct	W-1:W/2	6
4 compl.	W-1:W/2	7
8 direct (*only in 64-bit ISA*)	15-0	8
8 compl. (*only in 64-bit ISA*)	15-0	9
8 direct (*only in 64-bit ISA*)	31-15	10
8 compl. (*only in 64-bit ISA*)	31-15	11
8 direct (*only in 64-bit ISA*)	47-32	12
8 compl. (*only in 64-bit ISA*)	47-32	13
8 direct (*only in 64-bit ISA*)	63-48	14
8 compl. (*only in 64-bit ISA*)	63-48	15

Table 5.3: Immediate value assignment for `ftchk` instruction.

Redundancy (R_s)	**`ftchk` immediate**			
	32-bit	**32-bit (compl. result)**	**64-bit**	**64-bit (compl. result)**
2 direct	2	10	2	3
2 compl.	3	11	18	19
4 direct	4	12	4	5
4 compl.	5	13	20	21
8 direct (*only in 64-bit ISA*)	NA	NA	8	9
8 compl. (*only in 64-bit ISA*)	NA	NA	24	25

is 7. According to Table 5.2, this means the bits in the range [W-1:W/2] (W=32 for 32-bit ISA and W=64 for 64-bit ISA) should be duplicated in a complementary format.

In cases where our data is in complementary redundancy format, we need a logic operation $f(.)$ to calculate $f(.)$ on the direct copies and the inverse ($\overline{f(.)}$) on the complemented copies to result in complemented outputs according to DeMorgan's theorem. Figure 5.7 shows the structure of complementary logic operations. Therefore, Skiva-V has logic instructions `andc`***n***, `xorc`***n***, and `xnorc`***n*** that calculate the logic operation and its inverse on part of the data in their source registers. In the 32-bit (resp. 64-bit) instruction set, ***n*** can have the value of 8 and 16 (resp. 8, 16, and 32) to operate in direct/complementary format on ***n*** consecutive bits.

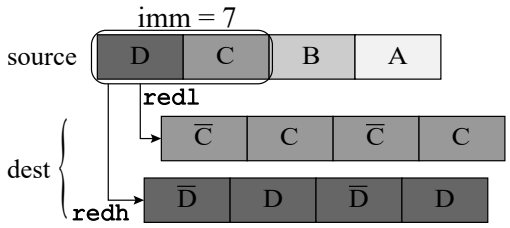


Figure 5.6: Example for `redl`/`redh` instructions with immediate value of 7. In both 32-bit and the 64-bit ISA, immediate=7 means duplicating the upper half-word (16 bits and 32 bits respectively for the 32-bit and 64-bit architectures) in the complemented format. `redh`/`redl` copies the upper/lower half of the source's selected bits in its destination register.

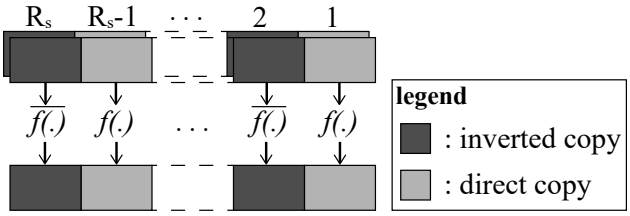


Figure 5.7: Complementary logic operations on complementary redundant data.

Finally, we propose an instruction in Skiva-V to check if the redundant copies of the data agree. The `ftchk rd, rs1, imm` instruction will check the redundant copies in the source register based on the immediate value and set the corresponding bit in the destination register to one if the copies of data do not agree (i.e. a fault is detected). To have continuity in the direct and complementary redundancy, the result of `ftchk` operation can be in the complementary format where the comparison result is copied both directly and inversely in the destination register.

Table 5.3 shows the immediate value encodings for the `ftchk` instruction. For example, if $R_s = 4$ and direct redundancy in the 32-bit ISA, i.e., immediate value is either 4 or 12, the comparison flags are calculated and stored in the destination register (`rd`) based on the

source register (`rs`) as follows for the least significant 8 bits:

$$\begin{aligned} rd[i] = (rs[i] \oplus rs[i+8])|| \\ (rs[i] \oplus rs[i+16])|| \\ (rs[i] \oplus rs[i+24]); \\ \forall i \in [0,7] \end{aligned}$$

The same calculated results will be duplicated directly (when immediate value is 4) or in inverse format (when immediate value is 12) to fill the remaining 16 bits of the destination register (`rd`).

5.3.2 ISA-level Performance Analysis

We evaluate our instruction set extension on RISC-V for its performance. For each proposed instruction, we write a C code defining the functionality of the instruction. On an implementation of Skiva-V, this C code corresponds to only one instruction. We cross compile the C code with GCC once for RV32I and RV64I instruction sets and once for the RISC-V with bit-manipulation extension (RV32B and RV64B) [3]. While GCC with B-extension currently only supports 31 out of 95 proposed instructions in the bit-manipulation draft, it is the only tool available for automatic use of the instructions in the B-extension. Furthermore, a visual inspection of the instructions in this extension confirms that there is no exact match for the instructions in Skiva-V.

As an example, Listing 5.2 and Listing 5.3 show the assembly codes for `ftchk` instruction with immediate value of 2 ($R_s = 2$ with direct redundancy) for RV32I and RV32B respectively. These assembly codes are generated by compiling the C code in Listing 5.1

[3] C equivalent codes and compiled assemblies for RV32/64I/B are accessible at `https://github.com/Secure-Embedded-Systems/Skiva-V`

Listing 5.2: Assembly code of the Skiva-V `ftchk` instruction with immediate value of 2 mapped to RV32I ISA

```
slli    a5,a0,16
srli    a5,a5,16
srli    a0,a0,16
xor     a0,a5,a0
slli    a5,a0,16
or      a0,a5,a0
```

Listing 5.3: Assembly code of the Skiva-V `ftchk` instruction with immediate value of 2 mapped to RV32B ISA

```
srli    a5,a0,16
pack    a0,a0,x0
xor     a0,a0,a5
slli    a5,a0,16
or      a0,a5,a0
```

with RISC-V's open-source GCC compiler and flags `-march=rv32g` and `-march=rv32gb` respectively. As the assembly codes show, the `pack` instruction in B-extension can replace the first two instructions in Listing 5.2 therefore reducing the total number of instructions by one.

Listing 5.1: C equivalent for 32bit `ftchk` with immediate value of 2

```
uint32_t ftchk2 (uint32_t rs1) {
        uint32_t rd;
        uint32_t compare;

        compare = (rs1 & 0x0000ffff) ^
                  ((rs1 & 0xffff0000) >> 16);
        rd = compare | (compare << 16);

        return rd;
}
```

As another example, the C equivalent for 64bit `ftchk` instruction with immediate value of 18 (complementary redundancy $R_s = 2$ with direct output) is shown in Listing 5.4. The compiled assembly codes for RV32I and RV32B are shown in Listing 5.5 and Listing 5.6 respectively. Similar to the previous example, the first two instructions for RV64I are replaced by one `pack` instruction from RV64B. Additionally, `xor` and `not` instructions are replaced

Listing 5.4: C equivalent for 64bit `ftchk` with immediate value of 18

```
uint64_t ftchk18_64 (uint64_t rs1) {
        uint64_t rd;
        uint64_t compare;

        compare = ~(rs1 & 0x00000000ffffffff) ^
                  ((rs1 & 0xffffffff00000000) >> 32);
        rd = compare | (compare << 32);

        return rd;
}
```

Listing 5.5: Assembly code of the Skiva-V `ftchk` instruction with immediate value of 18 mapped to RV64I ISA

```
slli    a5,a0,32
srli    a5,a5,32
srli    a0,a0,32
xor     a5,a5,a0
not     a5,a5
slli    a0,a5,32
or      a0,a0,a5
```

Listing 5.6: Assembly code of the Skiva-V `ftchk` instruction with immediate value of 18 mapped to RV64B ISA

```
srli    a5,a0,32
pack    a0,a0,x0
xnor    a0,a0,a5
slli    a5,a0,32
or      a0,a5,a0
```

by the `xnor` instruction in the B-extension.

Table 5.4 and Table 5.5 show the number of instructions from RISC-V ISA to implement the Skiva-V instructions. Based on our calculations, each of the 32-bit/64-bit Skiva-V operation replaces on average 22.34/29.98 instructions from the RV32/RV64 ISA and 21.84/29.59 instructions from the RV32B/RV64B ISA. All the proposed instructions pass the criteria of replacing a minimum of *three* instructions. The general reason for this poor behavior of RV32/64I/B is the required fine-grained operations at bit-level.

Although the reported numbers for RV32B and RV64B are not significantly different from RV32I and RV64I, the real advantage of the RISC-V's bit-manipulation extension can be much bigger but not yet supported by the GCC code generator. For instance, `rev.p rd, rs, 1` in RV32B/RV64B is functionally equivalent to `subrot rd, rs, 2` in Skiva-V 32/64-bit. However, this was the only instance we found in the bit-manipulation extension

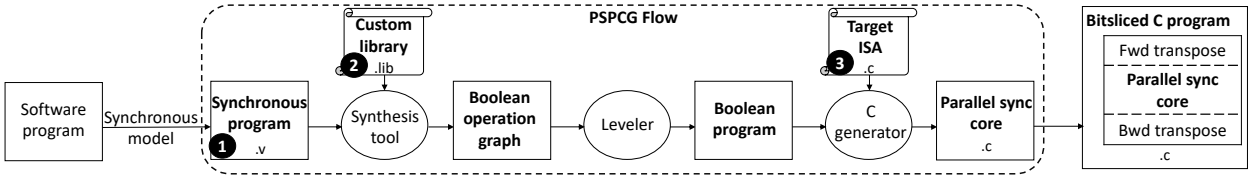


Figure 5.8: High-level description of PSPCG steps.

that was obviously equivalent to the instructions in Skiva-V.

Furthermore, we calculate the number of registers each instruction-equivalent code snippet uses on RV32I/RV32B and RV64I/RV64B as a measure of register pressure. We calculate the register use of each code snippet and compare it to that of its corresponding custom instruction. We make the worst case scenario assumption on the register usage in Skiva-V custom instruction that each r-type instruction (namely `andc`*n*, `xorc`*n*, `xnorc`*n*, `(inv)tr2l/h`) uses 3 *distinct* registers and each i-type instruction (namely `subort`, `redl/h`, `ftchk`) uses 2 *distinct* registers. As shown in Table 5.4 and Table 5.5, even under our pessimistic assumption, on average, Skiva-V custom instructions use 1.47×/1.65× fewer registers compared to RV32I/RV64I and 1.58×/1.75× fewer registers compared to RV32B/RV64B ISA.

5.3.3 Implementation

We integrate the 32-bit Skiva-V instructions into an in-order, five-stage pipeline implementation of the RISC-V RV32I ISA. For this implementation, we use the open-source BRISC-V [12] core. This core consists of five pipeline stages, namely fetch, decode, execute, memory, and write-back. The simplicity of the Skiva-V ISE architecture, enables the easy integration of the instructions which only affect the decode stage, the ALU unit in the execute stage, and the control unit of the processor. The changes applied to the processor are to decode the added instructions (in the decode stage according to the assigned opcodes in Table 5.1), execute them in the ALU (in execute stage), and bypass their outputs to the next instructions in case of dependency (from write back stage to decode stage) to reduce the number of

Table 5.4: ISA-level performance evaluation of Skiva-V 32-bit instructions

Skiva-V 32	**RV32I**		**RV32B**	
	# of instr	**reg. use**	**# of instr**	**reg. use**
tr2h rd, rs1, rs2	115	2×	115	2×
tr2l rd, rs1, rs2	115	2×	115	2×
invtr2h rd, rs1, rs2	115	2×	114	2×
invtr2l rd, rs1, rs2	115	2×	115	2×
subrot rd, rs, 2	9	1.5×	9	1.5×
subrot rd, rs, 4	9	1.5×	9	1.5×
redl rd, rs, 2	4	1×	3	1×
redh rd, rs, 2	4	1.5×	4	1.5×
redl rd, rs, 3	5	1×	4	1.5×
redh rd, rs, 3	5	1.5×	4	1.5×
redl rd, rs, 4	7	1.5×	7	1.5×
redh rd, rs, 4	8	1.5×	8	1.5×
redl rd, rs, 5	9	1.5×	9	1.5×
redh rd, rs, 5	11	1.5×	11	1.5×
redl rd, rs, 6	8	1.5×	8	1.5×
redh rd, rs, 6	8	1.5×	8	1.5×
redl rd, rs, 7	11	1.5×	11	1.5×
redh rd, rs, 7	10	2×	10	2×
ftchk rd, rs, 2	6	1×	5	1.5×
ftchk rd, rs, 3	6	1×	5	1.5×
ftchk rd, rs, 4	16	1.5×	16	1.5×
ftchk rd, rs, 5	17	2×	16	2.5×
ftchk rd, rs, 10	7	1×	6	1.5×
ftchk rd, rs, 11	8	1.5×	8	1.5×
ftchk rd, rs, 12	17	1.5×	17	1.5×
ftchk rd, rs, 13	19	2×	16	2.5×
andc16 rd, rs1, rs2	7	1×	6	1.67×
xorc16 rd, rs1, rs2	3	1×	3	1×
xnorc16 rd, rs1, rs2	4	1×	4	1×
andc8 rd, rs1, rs2	13	1.33×	13	1.33×
xorc8 rd, rs1, rs2	13	1.33×	11	1.33×
xnorc8 rd, rs1, rs2	11	1.33×	9	1.33×

inserted bubbles in the pipeline. These modifications to the processor design do not affect the critical path of the circuit and therefore render the maximum clock frequency unchanged.

Furthermore, to evaluate the area footprint of these instructions, we synthesize the Skiva-V implementation using the open SkyWater 130nm standard cell library[4]. The implementation of the five-stage RV32I ISA without Skiva-V extension has an area of 88356.25um^2 and a cell count of 5006. After adding the Skiva-V instructions, the area and cell count increase to 97381.07um^2 and 5643 showing a 10.21% and 12.72% increase in area and cell count respectively.

[4]https://github.com/google/skywater-pdk

Table 5.5: ISA-level performance evaluation of Skiva-V 64-bit instructions

Skiva-V 64	**RV64I**		**RV64B**	
	# of instr	**reg. use**	**# of instr**	**reg. use**
tr2h rd, rs1, rs2	244	2.33×	243	2.33×
tr2l rd, rs1, rs2	243	2.33×	244	2.33×
invtr2h rd, rs1, rs2	244	2.33×	243	2.33×
invtr2l rd, rs1, rs2	246	2.67×	247	2.33×
subrot rd, rs, 8	9	1.5×	9	1.5×
subrot rd, rs, 4	9	1.5×	9	1.5×
subrot rd, rs, 2	9	1.5×	9	1.5×
redh rd, rs, 10	15	1.5×	15	1.5×
redl rd, rs, 10	16	1.5×	16	1.5×
redh rd, rs, 11	17	2×	17	2×
redl rd, rs, 11	19	2×	19	2×
redh rd, rs, 12	16	1.5×	16	1.5×
redl rd, rs, 12	16	1.5×	16	1.5×
redh rd, rs, 13	19	2×	19	2×
redl rd, rs, 13	19	2×	19	2×
redh rd, rs, 14	16	1.5×	16	1.5×
redl rd, rs, 14	16	1.5×	16	1.5×
redh rd, rs, 15	18	2.5×	18	2.5×
redl rd, rs, 15	19	2×	19	2×
redh rd, rs, 2	4	1×	4	1×
redl rd, rs, 2	4	1×	3	1×
redh rd, rs, 3	5	1×	5	1.5×
redl rd, rs, 3	5	1×	4	1.5×
redh rd, rs, 4	7	1.5×	7	1.5×
redl rd, rs, 4	8	1.5×	7	2×
redh rd, rs, 5	13	2×	13	2×
redl rd, rs, 5	11	1.5×	10	2×
redh rd, rs, 6	8	1.5×	8	1.5×
redl rd, rs, 6	9	1.5×	8	2×
redh rd, rs, 7	10	2×	10	2×
redl rd, rs, 7	13	2×	13	2×
redh rd, rs, 8	16	1.5×	16	1.5×
redl rd, rs, 8	15	1.5×	15	1.5×
redh rd, rs, 9	19	2×	19	2×
redl rd, rs, 9	17	2×	17	2×
ftchk rd, rs, 18	7	1×	5	1.5×
ftchk rd, rs, 19	8	1.5×	7	2×
ftchk rd, rs, 20	20	2.5×	18	2.5×
ftchk rd, rs, 21	21	2.5×	19	2.5×
ftchk rd, rs, 24	39	2.5×	36	3×
ftchk rd, rs, 25	39	2.5×	36	3×
ftchk rd, rs, 2	6	1×	5	1.5×
ftchk rd, rs, 3	7	1×	6	1.5×
ftchk rd, rs, 4	18	2×	18	2×
ftchk rd, rs, 5	19	2×	19	2×
ftchk rd, rs, 8	36	2×	36	2×
ftchk rd, rs, 9	36	2×	36	2×
andc32 rd, rs1, rs2	7	1×	6	1.33×
xorc32 rd, rs1, rs2	4	1×	4	1×
xnorc32 rd, rs1, rs2	4	1×	3	1.33×
andc16 rd, rs1, rs2	9	1.33×	9	1.33×
xorc16 rd, rs1, rs2	4	1×	4	1×
xnorc16 rd, rs1, rs2	4	1×	4	1×
andc8 rd, rs1, rs2	9	1.33×	9	1.33×
xorc8 rd, rs1, rs2	4	1×	4	1×
xnorc8 rd, rs1, rs2	4	1×	4	1×

5.4 Coding Support

One of the challenges for bitsliced programming is its code generation. For SKIVA programming, we adopt Parallel Synchronous Programming (PSP), a model that directly maps into bitsliced programs and integrates control logic into bitsliced code [93]. Examples of parallel synchronous programs have since been shown in software implementation of light-weight encryption ciphers [96] and variable-precision multiplication used in neural networks [145]. In this section, we demonstrate how bitsliced programs are a subset of the parallel synchronous programs and therefore the automated code generator for PSP (*i.e.* PSPCG) can be used to automate the generation of bitsliced code.

PSP is semantically similar to a synchronous finite state machine with datapath (FSMD). Parallel synchronous programs consist of a *core function* with a status output that shows when the results are ready. This core function will be called iteratively until the status output shows the execution is done, while each iteration corresponds to a synchronous evaluation of the PSP design [93].

```
while (!stat_done) {
    core_f(inputs, &outputs, &stat_done);
}
```

Bitslicing becomes a subset of PSP by unfolding the loop and adding it into the logic of the core function. This results in a flattened function containing only logic operations in bitsliced format. Hence we can use the same automatic PSP code generation methodology (PSPCG) for bitsliced codes.

As Figure 5.8 shows, to generate the bitsliced code of a software program using PSPCG, first a synchronous model of the program is needed. This synchronous model in PSPCG flow is encoded as a Verilog file. Once ready, we feed the synchronous model of the program (①) as well as the description of the instructions in our target ISA to PSPCG. These target instructions should be provided in two formats, one following liberty file (used for

describing logic libraries) (②) and the other as inline assembly in C (③). Given these inputs, PSPCG internally synthesizes the given synchronous model to construct a Boolean operation graph and levels the graph to generate a Boolean program. In its last stage, the parallel synchronous core of the given model is generated as a C function. By prepending the forward transposition of the input data and appending the backward transposition of the results to the generated C function we will have the complete bitsliced C code.

Coding for Skiva-V

To generate bitsliced code for Skiva-V, we follow the PSPCG method as mentioned previously. In our custom library for the synthesis step, we use general logic cells `AND`, `OR`, `XOR`, and `NOT`. In our C code, we expand each of these general instructions as a sequence consisting of Skiva-V instructions in the form of inline assembly depending on the desired redundancy and masking scheme. The expanded instructions will implement the secure gadgets used for masking and redundant computation. For instance, in the 32-bit architecture, for the first- (resp. third-) order masking with no redundancy, each `AND` operation will be replaced by the sequence of assembly instructions shown in Listing 5.7 (resp. Listing 5.8).

Finally, we add the proposed instructions to the RISC-V GCC assembler. This way, the mnemonics of the new instructions are recognized by the assembler and will be automatically mapped to the correct opcodes in the executable file[5].

5.5 Direct Memory Access with Transpose Support

In this section, we describe the Transpose DMA (T-DMA) functionality, design, and the area footprint of the synthesized circuit. T-DMA is capable of performing the same operations as Skiva-V's instructions `(inv)tr2l, (inv)tr2h, redl, redh, ftchk` on the fly on up to

[5]We will open-source the modified GCC for Skiva-V before paper's publication.

Listing 5.7: First-order masked implementation of AND operation. Inputs are at `a1,a5`, random numbers are at `a0,a4`, the output is written in `a6`.

```
xor     t0, a1, a0
subrot  s0, a0, 2
xor     t2, t0, s0
xor     s0, s0, s0
and     a7, a5, t2
subrot  t5, t2, 2
and     t1, t5, a5
xor     t5, t5, t5
xor     t3, a4, a7
xor     t4, t3, t1
subrot  t6, a4, 2
xor     a6, t6, t4
```

Listing 5.8: Third-order masked implementation of AND operation. Inputs are at `a2,a4`, random numbers are at `a3,a5`, the output is written in `a1`.

```
xor     s2, a3, a2
subrot  s4, a2, 4
xor     s3, s2, s4
xor     s4, s4, s4
and     a0, s3, a5
subrot  t0, s3, 4
and     a6, t0, a5
subrot  s0, a5, 4
and     a7, s0, s3
subrot  t2, t0, 4
and     t1, t2, a5
xor     t0, t0, t0
xor     s0, s0, s0
xor     t2, t2, t2
xor     t3, a4, a0
xor     t4, t3, a6
xor     t5, t4, a7
subrot  s1, a4, 4
xor     t6, t5, s1
xor     a1, t6, t1
```

32 consecutive memory locations at once.

5.5.1 T-DMA Functionality

The proposed T-DMA module is capable of the following:

- Transposing/Reverse transposing an arbitrary number of memory locations (up to thirty-two) starting from a source address and storing the result in given addresses starting from an arbitrary destination address.
- Generating/Removing the masking shares of data in the source address according to the given Skiva-V working mode.
- Generating/Removing the redundancy for the data stored in given source address according to the given Skiva-V working mode.

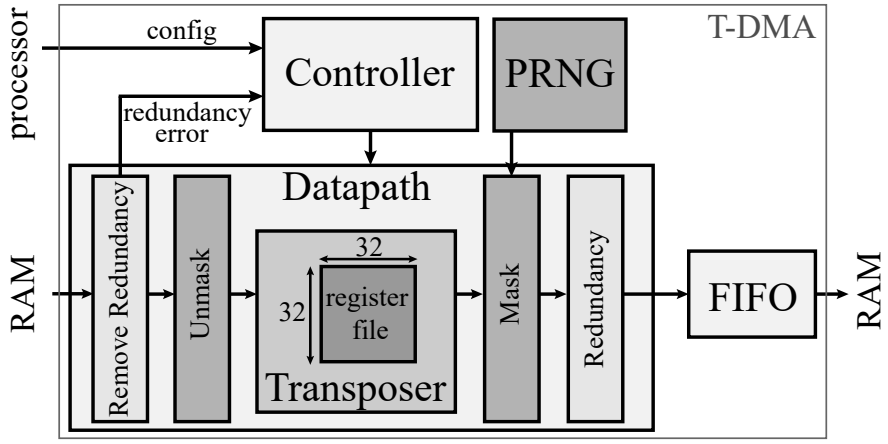


Figure 5.9: Block diagram of the T-DMA module.

- Checking for consistency between the redundant copies of the data stored in a given memory address.

5.5.2 T-DMA Design

Figure 5.9 shows the design of our proposed T-DMA module. The T-DMA module consists of a *controller* and a *datapath*. The system's processor will program the T-DMA by writing to the controller. Programming the DMA includes telling the controller the D, R_s, direct/-complementary redundancy, source memory address, destination memory address, number of memory locations, number of valid bits in each location, and whether we need to {mask and duplicate} the data or {unmask, and check and remove the redundancy}.

The controller, then, sets the signals for the datapath to perform the transformations. At the core of the T-DMA's datapath design, is the *transposer* with a register file of thirty-two 32-bit registers (128 bytes) tuned for a 32-bit micro-architecture. Once the T-DMA starts the memory transfer, it will load the data residing in a programmable number of locations starting from a source address into the register file. While transferring the data from the system's RAM, the existing redundancy and masking will be removed for backward transposition. In case of a forward transposition, the removal of redundancy and masking

are turned off and the masking shares for each bit of the data are generated based on the programmed number of shares ($D \in \{1, 2, 4\}$). We use the Cellular Automata-based PRNG[6] to generate the randomness required for masking the data.

Once the masking shares are generated, the data is formatted according to the programmed redundancy scheme ($R_s \in \{1, 2, 4\}$ and direct/complementary copy configuration) and stored in the destination memory locations.

To perform the reverse transposition, the transposer first checks for the correctness of the redundant data. Once the correctness is ensured, it removes the redundancy and unmasks the data. Finally, the dis-transposed data will be saved to the destination addresses.

The output of the datapath is stored in a First In, First Out (FIFO) memory. This memory stores the address and data of each output to be sent to the system's RAM. In our implementation, the FIFO is 256 bytes with 32 entries of 64 bits wide (to store the concatenated 32-bit address and 32-bit data). Once the transposition is done, T-DMA starts writing each entry of the FIFO to the system's RAM.

Despite only having a 32×32 register file in its transposer, T-DMA is capable of transposing up to $< 2^{16}$ distinct data each of length $< 2^{12}$ bits by being programmed only once. This feature is enabled by the *stride algorithm* (Figure 5.10). Following the stride algorithm, the data is divided into *blocks*, each containing a maximum of 32 distinct data. Each data is of a programmable word length WL $< 2^{12}$. Each block is divided into *offsets* containing up to 32 bits of up to 32 distinct data. Figure 5.10 shows this structure. T-DMA iterates over all the offsets in a block. It takes the first offset containing the least significant bits of each data in a block to load the transposer's register file. Subsequently, T-DMA offloads the transposed data to the memory. The hexagons in Figure 5.10 represent this offloading. It then moves to the next offset containing the next 32 significant bits of the data in the block. Once all the bits of the data in the current block are transposed and stored in the

[6] `https://github.com/secworks/ca_prng`

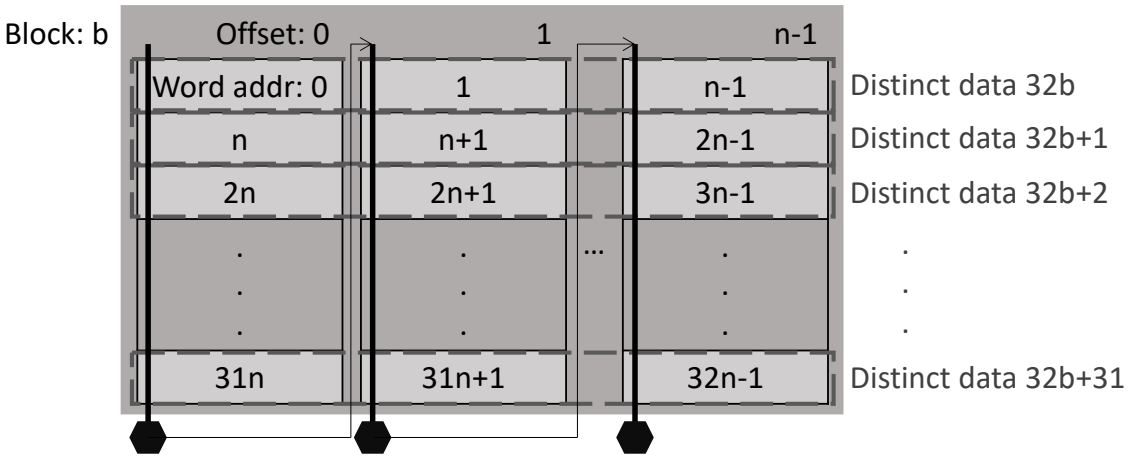


Figure 5.10: Stride algorithm used in T-DMA. $n = \lceil \frac{WL}{32} \rceil$

destination addresses, it moves to the next block. In our implementation, 32-bit parts of the same data are at consecutive addresses therefore the distinct data in each offset are not in consecutive addresses, rather they are $n = \lceil \frac{WL}{32} \rceil$ words apart. In each configuration of the T-DMA there are $\lceil \frac{WC}{32} \rceil$ blocks to transpose. The same structure applies to both forward and backward transposition.

5.5.3 Employing T-DMA

To use the T-DMA, first, the source address should be written in the controller through the processor. In our implementation, a little-endian architecture is assumed therefore the source address is considered to hold the least significant 32 bits of the data. In addition, masking order (D), redundancy order (R_s), word length (WL), and word count (WC) are written to a 32-bit configuration register containing 2 bits for holding D (0 for $D = 1$, 1 for $D = 2$, and 2 for $D = 3$), similarly 2 bits for holding R_s, 16 bits for WC, and 12 bits for WL. Through writing to another configuration register, it is specified whether the redundancy scheme is direct or complementary and whether we are running forward or backward transposition. Lastly, the seed for the on-chip PRNG is also provided to the

controller. Once this configuration is complete, the T-DMA will start operating by receiving the destination register in a specific addressable register inside the controller.

As mentioned previously, the controller is capable of checking the correctness of the redundant data. The result of this check is written to a read-only (by the processor) status register. After the completion of T-DMA's job, the processor can read the status register to confirm the correctness of the redundant data. Furthermore, while the T-DMA is running, a busy flag will be held high in another status register. Using this status register, the processor will know when the T-DMA is ready for the next data transfer. Figure 5.11 shows the control and status registers in T-DMA.

5.5.4 Implementation

To evaluate the size of T-DMA, we synthesize the circuit for SkyWater 130nm standard cell library. The T-DMA implementation shows a total area of 161524.07um^2 and a cell count of 9017 with the FIFO being the biggest contributor occupying more than 50% of the total area.

5.6 System Integration

To integrate Skiva-V processor and the T-DMA, we make the T-DMA implementation programmable from the processor by making all the aforementioned configuration and status registers in the controller address-accessible. Figure 5.11 shows these registers.

Every memory access from the memory stage of the pipeline goes through the memory interface. Figure 5.12 shows the connection between the modules in the integrated system. The memory interface detects whether the address is within the range of T-DMA or data memory.

In case of addressing the T-DMA, memory interface starts the transmission with the T-

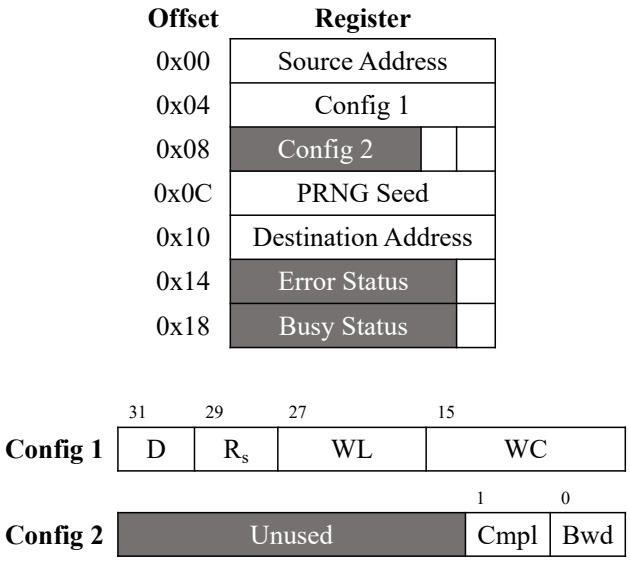


Figure 5.11: Address-accessible 32bit registers for communicating with and programming the T-DMA. Grey cells are unused. Backward transposition when `Bwd`=1, forward otherwise. Complemented redundancy when `Cmpl`=1, direct redundancy otherwise.

DMA which can include programming the T-DMA (write) or accessing its status bits (read). When the processor is trying to access the data memory, the interface module communicates with the memory arbiter.

Memory arbiter takes care of prioritizing memory accesses from the processor core and T-DMA. When T-DMA is programmed to access the data memory, memory arbiter prioritizes T-DMA's memory access over the memory access requests from the processor core. Therefore, the processor core will insert bubbles into its pipeline while waiting for the result of its memory access.

Implementation

We synthesize the integrated system (Figure 5.12) for the SkyWater 130nm standard cell library and measure the total area of 270,152.09um^2 and a total cell count of 14204. Subtracting the synthesized area of the Skiva-V and T-DMA (reported in the previous sections) from the integrated system, consisting of the memory arbiter module and the added logic to

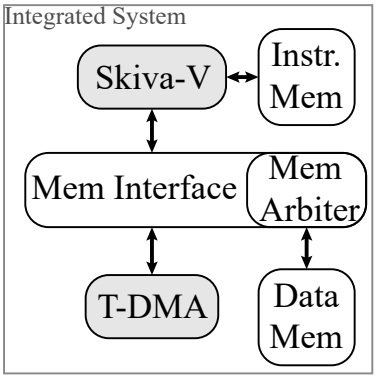


Figure 5.12: Integration of Skiva-V and T-DMA.

the memory interface module, the integration adds around 11,246.95um^2 (4.16% overhead) to the overall area.

5.7 Benchmark

In the following, we run all the experiments on our integrated system. We demonstrate the advantage of hardware support for data transposition, the performance cost of redundant computation, and the benefit of instruction-support for performance of masked implementations.

5.7.1 Cost of Transposition

To characterize the overhead of transposition more thoroughly, we evaluate the cost of transposition in our implemented system in terms of the required number of clock cycles. We write a program in which K ($2 \leq K \leq 32$) adjacent bits in a 32-bit register need to be transposed to reside in 1 bit of K registers. We run the same program in three different settings: using only standard RV32I instructions, using Skiva-V's transpose instructions, i.e., `tr2l` and `tr2h`, and using the T-DMA. We compare the first two cases in terms of number of required instructions and all three cases in terms of number of clock cycles.

Using the instructions in Skiva-V provides between 3× to 10× decrease in the number of instructions depending on the value of K. Furthermore, as Figure 5.13 shows, for each K, the number of clock cycles required to transpose K adjacent bits in a 32-bit register is reduced between 3× to 6× using the Skiva-V instructions. T-DMA and Skiva-V perform closely in this scenario with T-DMA having a better performance for $K \geq 19$. This is due to the overhead of programming the T-DMA module (29 clock cycles). Increasing the number of bits to transpose, causes an increase in run-time of the transposition using the T-DMA by 7 and using the Skiva-V instructions by 9 clock cycles. Therefore, for transposing only a few adjacent bits, the programmer is better off using the Skiva-V operations.

These results confirm the benefits of the transpose instructions in Skiva-V. However, to demonstrate the benefits of having the T-DMA, we run another experiment in which $K \in \{2, 4, 8, 16, 32\}$ bits in K registers need to be transposed. Figure 5.14 shows that as K increases, the run-time of this transposition increases linearly ($14k + 19$, $R^2 = 1$) using the T-DMA but quadratically using the instructions in Skiva-V ($1.54k^2 - 4.3k + 64.9$, $R^2 = 1$) and RV32I ($55.1k^2 + 125k - 469$, $R^2 = 1$).

Depending on the application, a designer can choose between deploying Skiva-V instructions or the T-DMA module. If the core is expected to have many applications running concurrently, using T-DMA will take the load off the processor core which will prevent execution overlap. On the other hand, application-specific hardware accelerators will reduce the power consumption of the circuit compared to running the application on the processor (general-purpose). Furthermore, in cases where there is streaming data coming in, which needs to be transposed, using the T-DMA will perform faster and will not occupy the processor resources.

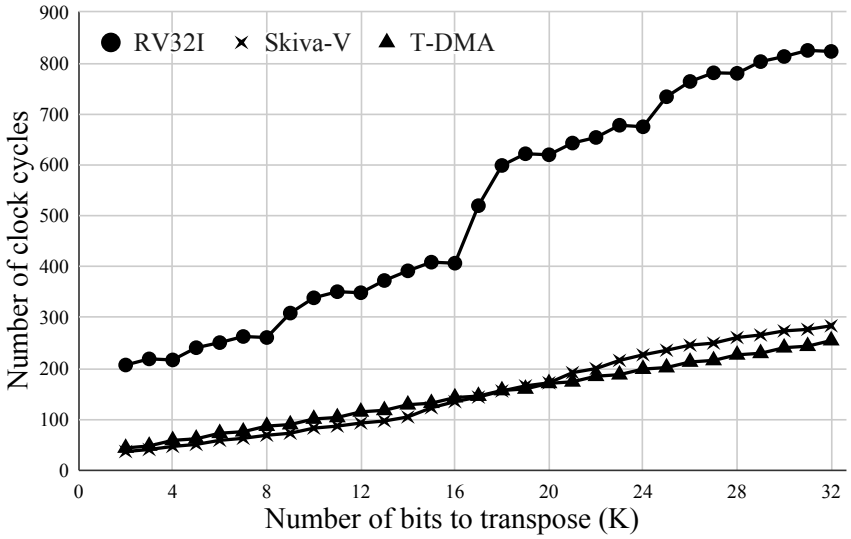


Figure 5.13: Number of clock cycles to transpose K adjacent bits of one register.

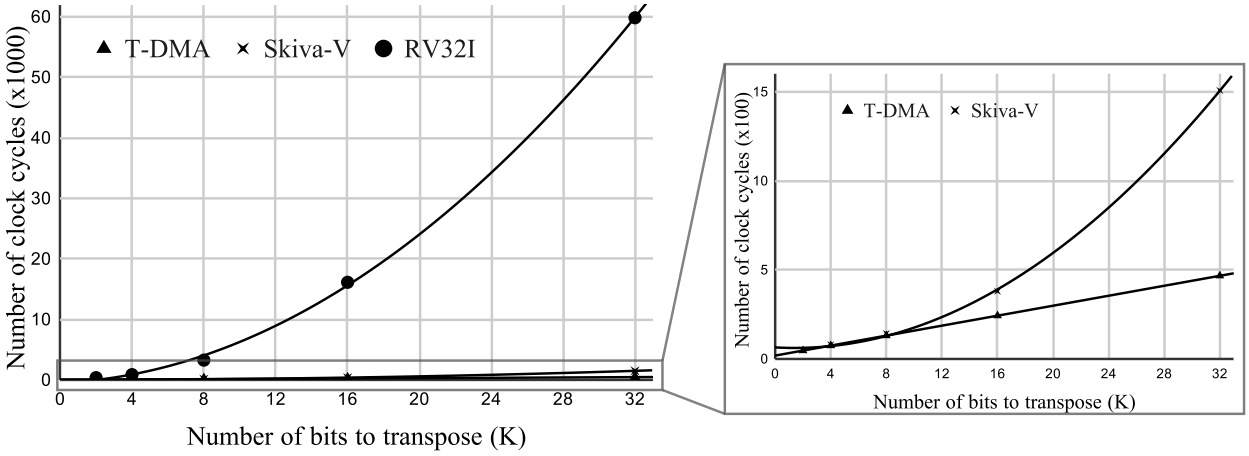


Figure 5.14: Number of clock cycles required to transpose K adjacent bits in K registers.

Table 5.6: Reciprocal of performance (cycles/byte).
The PRNG is assumed to have a high enough throughput to not cause any reading delay.
Tornado results are for ARM Cortex-M4; **Skiva-V** results are for RISC-V RV32I with extensions.

Cipher	D=1 (no masking)			D=2 (first-order masking)			D=4 (third-order masking)		
	Tornado	Skiva-V	Speed-up	Tornado	Skiva-V	Speed-up	Tornado	Skiva-V	Speed-up
ASCON	101	159.677	0.633	-	717.495	-	3070	1988.903	1.544
GIFT	358	441.941	0.810	-	1378.141	-	11080	3435.656	3.225

5.7.2 Cost of Redundant Computation

The redundant computation schemes affect the throughput of an execution as they reduce the number of parallel runs of a bitsliced software. For instance, when $R_s = 2$ each bit of data is copied twice in the same register therefore reduces the number of parallel runs by half. In general, in an R_s redundant scheme, the number of parallel runs will be divided by R_s therefore the throughput of the bitsliced software will also be divided by R_s.

5.7.3 Masked Implementations of LWC Ciphers

Table 5.7: Reciprocal of performance (cycles/byte).
The PRNG is assumed to have a high enough throughput to not cause any reading delay.

Cipher	D=2 (first-order masking)			D=4 (third-order masking)		
	RV32I	Skiva-V	Speed-up	RV32I	Skiva-V	Speed-up
ASCON	767.989	717.495	1.070	2547.578	1988.903	1.281
GIFT	1434.078	1378.141	1.041	4178.469	3435.656	1.216

We take the finalists of the NIST's Light-Weight Cryptography (LWC) competition that mention masking as their design options; ASCON [55] and GIFT-COFB [14]. We generate the masked implementation of their permutations (shown in Listing 5.9 and Listing 5.10) for $D \in \{1, 2, 4\}$ number of shares using the discussed code-generation method (Section 5.4). In the D=1, D=2, and D=4 settings, we support 32, 16, and 8 parallel executions respectively. We run the generated programs on Skiva-V system and calculate the number of cycles.

Listing 5.9: ASCON permutation

```
void ascon_perm (int* state, int* round_const) {
    for (i = 0; i<12; i++) {
        add_constant(state, round_const[i]);
        substitution(state);
        linear_diffusion(state);
    }
}
```

Listing 5.10: GIFT permutation

```
void gift_perm (int* state, int* key) {
    for (i = 0; i<40; i++) {
        sub_cells(state);
        perm_bits(state);
        add_roundkey(state, key);
        key_update(key);
    }
}
```

To supply the required randomness, we assume that the system has access to a pseudo-random number generator (PRNG) with a high throughput so that accessing a random number is equivalent to reading a register. Listing 5.7 and Listing 5.8 show the assembly code for masked 2-input `AND` instruction with D=2 and D=4 masked shares which follow the scheme described by Barthe et al. [17] and use the `subrot` instruction available in Skiva-V for rotation of shares sitting adjacently in the registers. Note that in this section, we do not perform any redundant computation, i.e., $R_s = 1$. In case of using complementary redundancy, the corresponding complementary logic instructions (described in Section 5.3.1) would replace the `and` and `xor` operations in Listing 5.7 and Listing 5.8.

Table 5.6 reports the number of cycles per byte calculated as $\frac{c}{s \times p}$ where c is the number of clock cycles, s is the size of the state of the cipher in bytes ($\frac{320}{8}$ for ASCON and $\frac{128}{8}$ for GIFT-COFB), and p is the number of parallel runs.

We compare our results with a similar work, Tornado [26], which reports the same cycles/-byte metric for the masked implementations (with the same fast assumption on the PRNG)

of the same permutations of the LWC candidates but on Cortex-M4. Table 5.6 highlights the advantage of having hardware support for bitslicing. First, for an unmasked implementation (D=1), Tornado reports higher performance. Note that we assume the data is already in bitsliced (and masked if $D \neq 1$) format hence the transposition is not included in our measurements. Therefore, for unmasked implementations, the Skiva-V instructions are not used and the comparison is between the RISC-V RV32I and Cortex-M4 ISAs and the code generation process. Thus, the higher performance reported by Tornado can be attributed to the more advanced nature of Cortex-M4 ISA compared to the RISC-V ISA and to the code generation tool. Second, for a third-order masked implementation (D=4), we observe that Skiva-V can result in 1.5× and 3.2× speedup for ASCON and GIFT-COFB respectively. Since Tornado does not report first-order masking results, we were not able to compare with Skiva-V for the D=2 setting. Moreover, we calculate the gained speed-up using Skiva-V instruction set over standard RV32I instruction set. As Table 5.7 shows, for both of the considered ciphers, Skiva-V provides a better performance and the gain in performance increases as the masking order increases.

We further analyze this data in terms of added number of clock cycles per unit increase in the masking order. This criterion depends on the cipher algorithm and the implementation of the algorithm. Since our goal is to compare the implementations, and not the cipher algorithms, we compare this criterion for ASCON and GIFT separately. For this purpose, we make a linear regression of the cycles/byte vs. number of shares (D) as reported in Table 5.6.

The trend-line of the linear regression for ASCON's perfomance is $613D-476$ for Skiva-V and $990D-889$ for Tornado. This means increasing the order of masking by one, will cause 613 extra clock cycles for Skiva-V and 990 for Tornado (1.6× increase compared to Skiva-V).

The same experiment for GIFT's performance shows a trend-line of $1002D - 587$ for Skiva-V and $3574D - 3216$ for Tornado. For this cipher, the increase of clock cycles is

more significant than ASCON which can be attributed to the multiplicative complexity of its algorithm. Furthermore, for an increase of one in the masking order, Tornado is affected by a 3.6× higher increase in the required clock cycles than Skiva-V.

As bitslicing has been shown to be useful for post-quantum applications [146], Skiva-V can help further boost the performance and security of such ciphers. As future work, we study usage of Skiva-V for post-quantum and asymmetric key algorithms.

5.8 Conclusion

In this contribution, we demonstrated how selected hardware techniques can significantly enhance the performance of bitslice software programs. By creating custom hardware to speed up frequent bit-level manipulation instructions, we illustrated a reduction on the register pressure for software bitslicing, and a performance boost over two state of the art bitsliced lightweight cipher designs. We demonstrated hardware support in the form of ISE as well as a stand-alone T-DMA peripheral. We presented synthesis results for the complete design in 130nm standard cells, and estimate the area overhead of the proposed extensions to be less than 5% at SoC level.

Chapter 6

Saidoyoki: Evaluating side-channel leakage in pre-and post-silicon setting

In this chapter, we introduce two in-house designed chips and compare their pre-silicon simulation-based power side-channel leakage assessments with measured post-silicon evaluations. This work was presented at IEEE International System-on-Chip Conference (SOCC) in 2021 [98].

6.1 Introduction

Power-based side-channel leakage is a known vulnerability in security SoC, yet it is hard to predict the amount of side-channel leakage at design-time. The fundamental reason is that the source of the vulnerability, namely data-dependency in the power dissipation of a design, is found at every abstraction level in the system stack [39]. For example, a design may appear perfectly side-channel resistant at RTL level, yet due to imperfections of the implementation at gate-level or below, the ideal side-channel leakage properties of RTL break down and cause side-channel leakage in the form of glitches [119] or cross-talk [75]. This strongly suggests

that extensive verification of side-channel leakage properties, at every abstraction level of the design, is crucial. In fact, contemporary *provably secure* countermeasures against side-channel leakage always assume a leakage model, a set of assumptions that must be supported by the implementation to deliver the security properties claimed. These leakage models (and their correctness for a given implementation) are an ongoing area of research [52].

Design-time verification of power-based side-channel leakage can be supported through power modeling and simulation. But at lower abstraction levels, power modeling is complex, and it comes with steep trade-offs between simulation time and resolution. Therefore, in the absence of comprehensive leakage modeling and/or efficient power simulation, the current design practice in side-channel resistant design in many cases still relies on prototyping. A prototype provides a real-life design test-case that can be measured and evaluated from a power side-channel leakage perspective. Field Programmable Gate Arrays (FPGA) are often selected as a prototyping target. However, the low level structure of FPGA does not reflect the gate-level netlist that is mapped on it, and therefore the FPGA may not be the best choice for the study of ASIC side-channel leakage behavior at low level. To compare the power side-channel leakage of a gate level netlist model to that of a prototype, ASIC technology with standard cells provides a better match.

In our recent research, we have designed several ASICs as a byproduct (and often as a proof of concept) of our experimental work. FAMEv2 (Fault Aware Microprocessor Extensions) is an SoC with fault-sensing capability based around a LEON-3 core. PICO is a similar SoC based around a 32-bit RISC-V core. Both SoCs contain several coprocessors as well as on-chip RAM, and they are built in 180nm standard cells. Most importantly, they are in-house designs, so we have access to all design information down to layout level.

To study the problem of pre-silicon vs post-silicon side-channel leakage modeling, we integrated both of these SoCs in a test platform. The Saidoyoki board provides a programming interface to download applications to either chip, and supports high-bandwidth power

measurement of each individual chip. The purpose of Saidoyoki is to validate a design flow for pre-silicon side-channel leakage estimation, by providing estimations next to actual side-channel leakage measurements. Our long term objectives are to understand and address two crucial shortcomings of side-channel leakage estimation from high-level models: (a) *false positives*, where the side-channel leakage estimation on the pre-silicon design model indicates side-channel leakage that cannot be confirmed by measurements; (b) *false negatives*, where the side-channel leakage estimation on the pre-silicon design does not show side-channel leakage while the measurements confirm the opposite. Both problems are hard and, to our knowledge, still unsolved.

The remainder of this chapter is organized as follows. In Section 6.2, we introduce the Saidoyoki platform at system-level, including the design decisions on the PCB to instrument it for side-channel leakage measurement, as well as a brief overview of the ASIC designs. Section 6.3 describes pre-silicon side-channel leakage estimation techniques. We describe the design flow used for the SoC power estimation, and handle several practical challenges related to design complexity. Section 6.4 describes post-silicon side-channel leakage measurement using the Saidoyoki board. Section 6.5 describes several experiments using Saidoyoki, and the analysis of the results obtained so far. Section 6.6 concludes the chapter.

6.2 Saidoyoki Platform

This section describes the features of the Saidoyoki platform, including the platform architecture, and the system architecture of the chips. We also compare with related work.

6.2.1 Saidoyoki PCB

A side-channel measurement setup includes a test target, a means to digitize power side-channel leakage, and a side-channel campaign controller. The controller exchanges stimuli

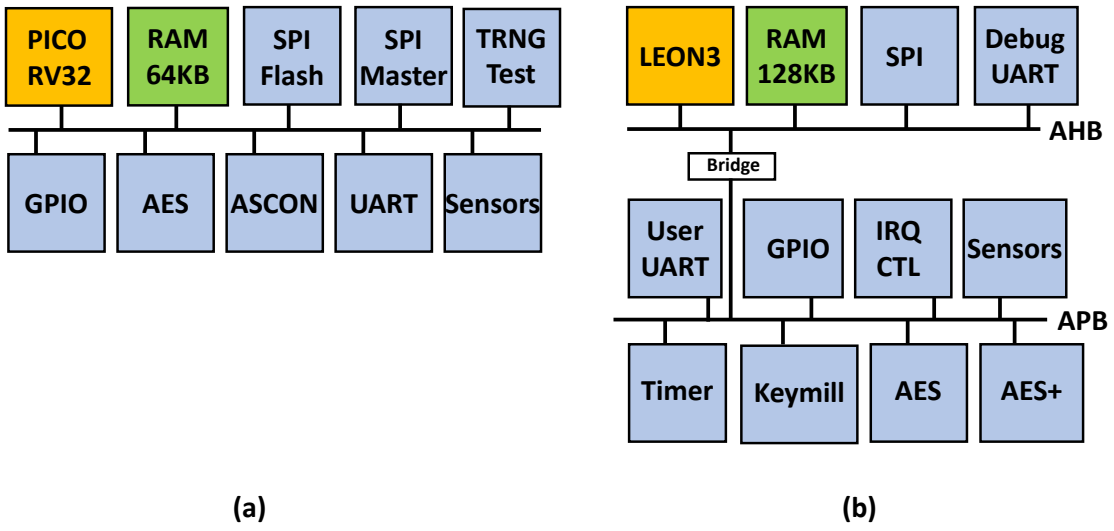


Figure 6.1: (a) PICO Block diagram and (b) FAMEv2 Block diagram.

with the test target, and while the target executes the stimuli, the target's power signature is captured. Afterwards, the campaign controller collects the power signatures and performs side-channel analysis. A PCB to support a test target thus has to provide multiple functions, including (a) providing easy access to the power consumption of the target, (b) real-time data input/output to exchange test stimuli with the campaign controller, (c) debugging of programmable/reconfigurable targets, and optionally, (d) adjusting target voltage/clock to study the impact of environmental factors. The Saidoyoki PCB was developed to support the PICO and FAME (Figure 6.1) chips as targets for side-channel measurement campaigns, and it supports all functions enumerated above.

Figure 6.2 and Figure 6.3 show the photo and the block diagram of the board respectively. The board is connected to the side-channel campaign controller with a single USB connection to multiplex multiple data and control channels used in a campaign (FT4232HL USB Bridge IC). These control channels include a UART debug connection for FAMEv2, a shared user UART, a shared SPI to program flash memories, and an I2C control channel for clock configuration.

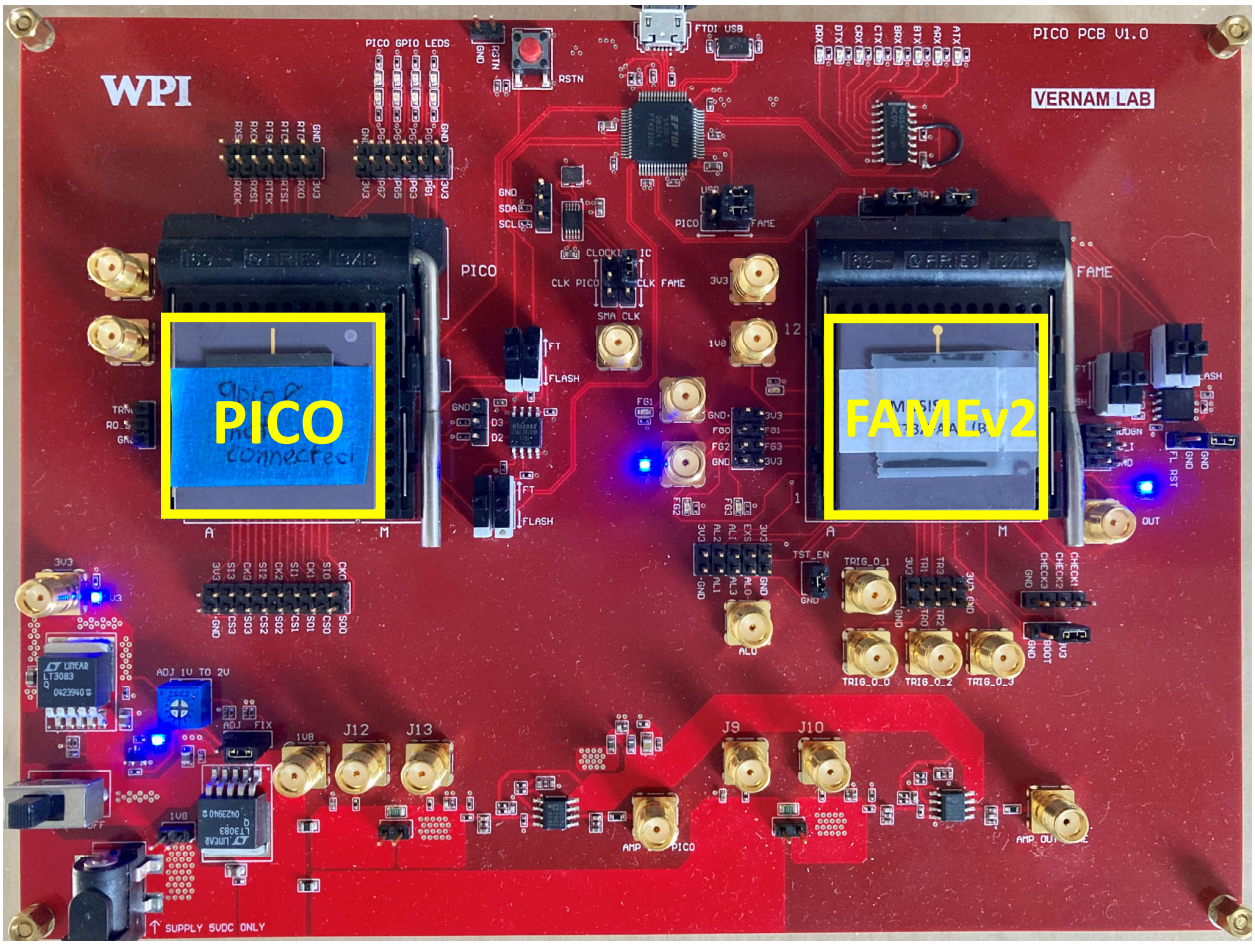


Figure 6.2: Photo of the Saidoyoki Board.

The side-channel leakage of each ASIC is captured by measuring voltage drop over a shunt resistor. The signal is also amplified through a differential broadband opamp, one for each ASIC. Saidoyoki uses a single 5V power supply that is regulated into a fixed 3V3 supply and an adjustable 1V8 supply. The adjustable supply feeds the ASIC core and can be varied between 1V and 2V. This rail is also split into two using ferrite beads and then connected to each chip independently. A precise shunt resistor is inserted into each branch of the 1.8V rail for power measurement.

The flash chips connected to PICO and FAMEv2 are externally programmable through USB. The board has several 3-point slider switches to physically switch the flash chips from

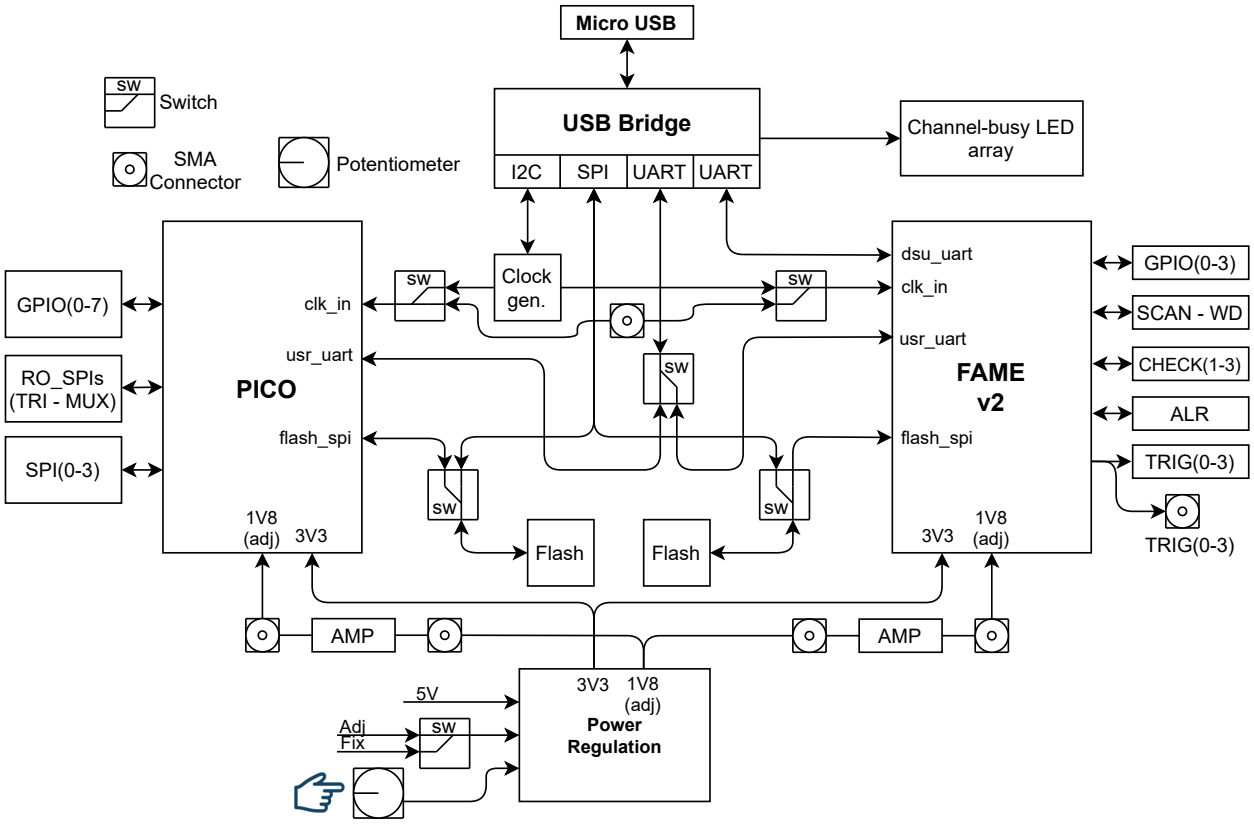


Figure 6.3: Block diagram of Saidoyoki Board.

external configuration to ASIC configuration. In the ASIC configuration, the ASIC will boot from the firmware configured in the flash chips.

Saidoyoki also includes a clock generation IC (SI5351A-B-GTR), which is able to generate configurable clock frequencies for the ASICs between 2.5KHz and 200MHz. The clock can also be provided from an external source through an SMA connector.

Finally, the Saidoyoki board supports low-level debugging tasks by bringing every chip pin out on a jumper header or an SMA connector. Additionally, the GPIO ports in each ASIC are visible through LEDs.

6.2.2 FAMEv2 ASIC

The FAMEv2 ASIC is a 180nm SoC with a LEON3 core and 128 kByte of internal memory, and several coprocessors. The program can execute either from on-chip SRAM or else from off-chip flash through an SPI flash ROM. A debug unit, controlled through an on-chip Debug UART, provides program loading, monitoring, breakpoints. The coprocessors are isolated from the processor through a bus bridge. All coprocessors exclusively operate as bus slaves, and communicate with the software through memory-mapped registers. FAME contains cryptographic accelerators for symmetric-key encryption (AES and AES+, a hardened version of AES) and pseudo-random stream generation (KeyMill). The sensors in FAME detect timing faults injected through clock glitching and voltage glitching. A detailed description of the fault detection mechanisms, and their integration with software, was presented earlier [174].

6.2.3 Pico ASIC

The PICO ASIC is a 180nm SoC with a RISCV (RV32) core and 64 kByte of internal memory, and several coprocessors. The program exclusively runs from off-chip flash through a Quad-SPI flash ROM. The system is integrated on a single bus. All coprocessors run as bus slaves and communicate with the RISC-V software through memory-mapped registers. PICO contains cryptographic accelerators for symmetric-key encryption (AES), authenticated encryption (ASCON), and hardware testing of true random bitstreams (TRNG test). The sensors in PICO detect fault injection as well as side-channel leakage. The FPGA prototype design of the sensors was presented earlier [169]. Furthermore, the high-level description of the sensors on PICO ASIC was presented earlier [107].

6.2.4 Related Work

Several other solutions have been proposed for high-bandwidth power monitoring of hardware. The SASEBO series of side-channel analysis boards [67], originally developed by AIST, is the oldest and arguably best known implementation. One version of SASEBO, SASEBO-R, is the only open source board to support an ASIC. The SAKURA series of boards [85], also by AIST, are based on Kintex-7/Spartan-6 FGPA or chip card microcontroller. The FOBOS from GMU [2] is an FPGA-based board oriented towards benchmarking. The HAHA board from U Florida [165] is a hybrid FPGA/microcontroller board oriented towards education. The Chip Whisperer is a low-cost measurement environment with FPGA-based and microcontroller based target boards [129]. The majority of these boards are oriented at studying side-channel leakage in a (configurable or programmable) test target. With Saidoyoki, since we have full knowledge of the test target's internal design, we can evaluate side channel leakage in either pre-silicon or else post-silicon side-channel leakage evaluation scenario's.

6.3 Pre-silicon Side-channel Leakage Estimation

In a pre-silicon setting, power-based side-channel leakage is estimated through a time-based simulation of the power consumption of the target.

6.3.1 Design flow for Hardware Targets

There are many solutions towards capturing side-channel leakage by simulation [39]. Figure 6.4 describes the steps for a hardware target such as for example a coprocessor in one of the SoCs. The Saidoyoki flow starts from a gate-level netlist, obtained from the FAME/PICO chip design or else through RTL synthesis of the design flows. Through a suitable testing scenario, a set of testing stimuli are defined. For classic DPA/CPA analysis of cryptographic

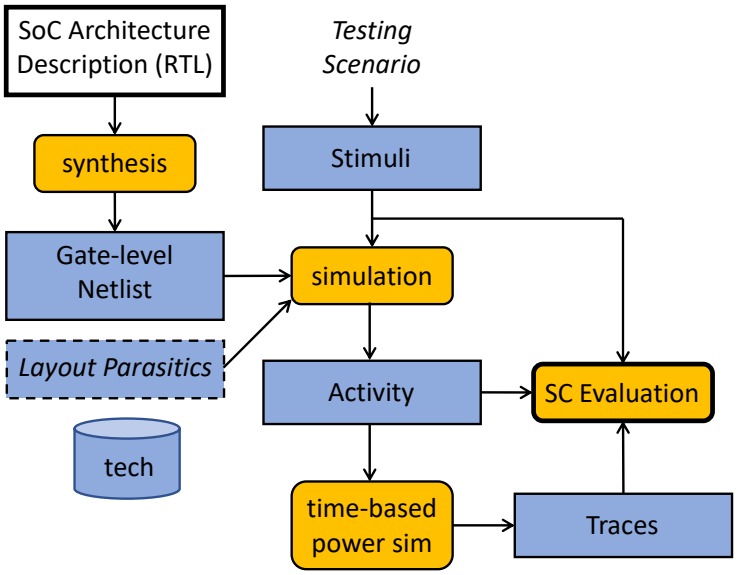


Figure 6.4: Flow for SCL assessment of hardware

hardware, for example, one selects a fixed key and a set of random plaintext/ciphertext. Specific or non-specific TVLA tests, on the other hand, require a combination of random and fixed key/plaintext inputs [152]. The activity files created from hardware simulation are used by a time-based power simulation tool, which provides a trace for every input test vector. Finally, the stimuli, activity files and traces are used as input for the side-channel evaluation. Our prototype implementation supports Cadence Genus or Synopsys Design Compiler, Cadence XCelium or Mentor ModelSim, and Cadence Joules respectively as synthesis, simulation, and power estimation tool. The side-channel evaluation is a customized Chipwhisperer script.

There is a trade-off between the level of simulation detail, and the ability of a design model to capture different causes and sources of side channel leakage. The gate-level abstraction provides reasonable accuracy. Because of the post-silicon artifacts available in Saidoyoki, we prefer lower simulation abstraction levels to investigate and verify lower-level effects.

6.3.2 Design flow for Software Targets

To ensure sufficient accuracy with software and SoC firmware targets, we extend the hardware based flow and simulate the microprocessor as a gate-level design as well. The main challenge is to convert the firmware into a form that can be integrated into the SoC hardware simulation. We compile the firmware into a binary that is used to initialize the on-chip memory at the start of the simulation. By compiling input test vectors (key, input) as hard-coded constants in the firmware binary, we also eliminate complex input/output schemes during simulation.

To enable testing only specific parts of the firmware, we make use of the GPIOs on the SoC as triggers. We set a GPIO pin high right before the point of interest and reset it to low right after. In the test bench module, during simulation, we monitor the trigger signal and log its set and reset time stamps. Later, in power trace generation phase, we generate the power trace only for the logged time period.

6.4 Post-silicon Side-channel Leakage Measurement

Measuring side-channel leakage in the post-silicon setting requires a campaign controller to measure power from the PICO or FAME chips on Saidoyoki while they execute a test application. We use a Chipwhisperer kit [129] integrated as in Figure 6.5. Chipwhisperer uses synchronous sampling and generates a clock signal for the target. This allows the side-channel leakage to be captured with low overhead of one to four times the target clock. A campaign executes a large collection of encryption operations on the target, with different input stimuli. For each operation, Chipwhisperer sends an input plaintext. The target responds with a trigger when the cryptographic operation starts. Chipwhisperer then collects the power signal and finally performs side-channel leakage analysis or assessment.

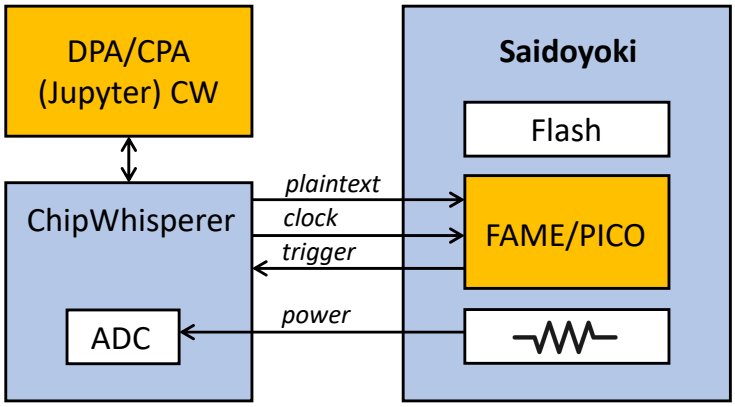


Figure 6.5: Integration of Chipwhisperer and Saidoyoki

6.5 Results

In this section, we describe three experiments performed using the Saidoyoki board. The first is a post-silicon side-channel leakage analysis on FAME; the second and third are a pre-silicon side-channel leakage analysis on PICO. In each case, we used AES encryption as the target algorithm, with the SubBytes output as the leakage model.

6.5.1 Post-silicon evaluation of FAME SoC firmware

Figure 6.6 (top) shows a power signal captured from the first round of AES encryption running on the LEON3 core in the FAME chip. Typically, post-silicon traces are very noisy and it's not easy to visually recognize the different portions of the algorithm. However, since the code on the target is fully known, it's easy to determine when each function executes. In this campaign, the FAME target runs at 4MHz and the side-channel is sampled at 16 MHz. Each trace contains 24,400 points. Figure 6.6 (bottom) shows a correlation plot obtained from running CPA on key byte 7 on the traces of 25,000 encryptions. The correlation plot shows two spikes: one of them when the SubBytes output is computed and stored in memory, and a second when ShiftRows reads that result and moves it to another memory location.

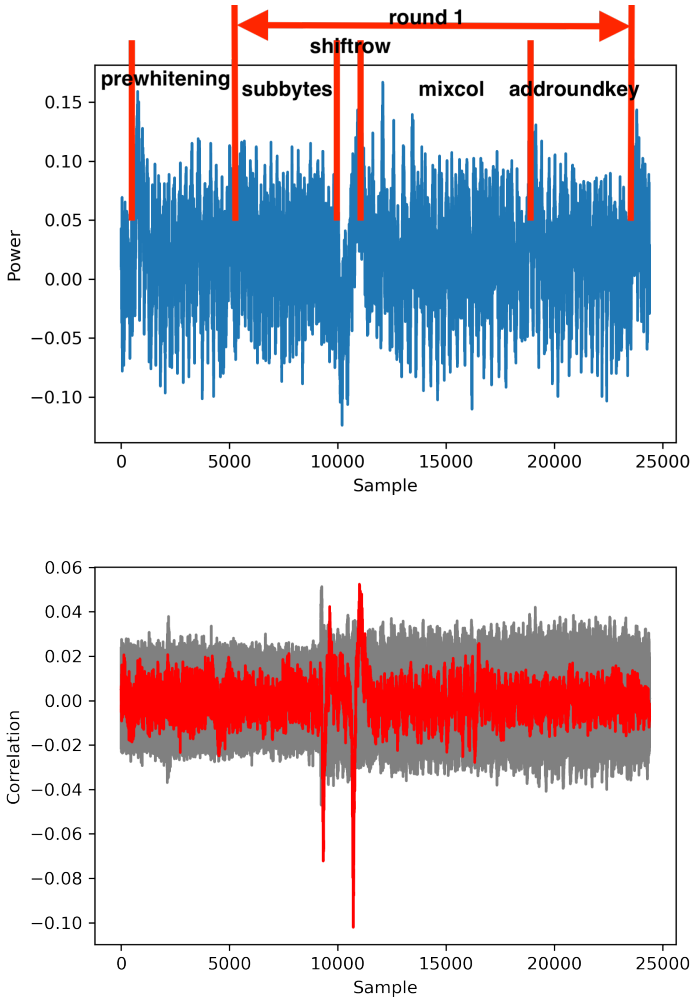


Figure 6.6: CPA HW on FAME executing AES firmware: (top) power traces identifying portions of the first round (bottom) outcome of Correlation Power Analysis

While this type of side-channel analysis is a standard operation, it is not without its pitfalls. The black-box nature of the power signal, as well as the high level of noise, requires careful tuning of the measurement parameters.

6.5.2 Pre-silicon evaluation of PICO SoC coprocessor

A significant advantage of Saidoyoki is its ability to support pre-silicon side-channel leakage assessment using gate-level power simulation. There is a trade-off between the speed of

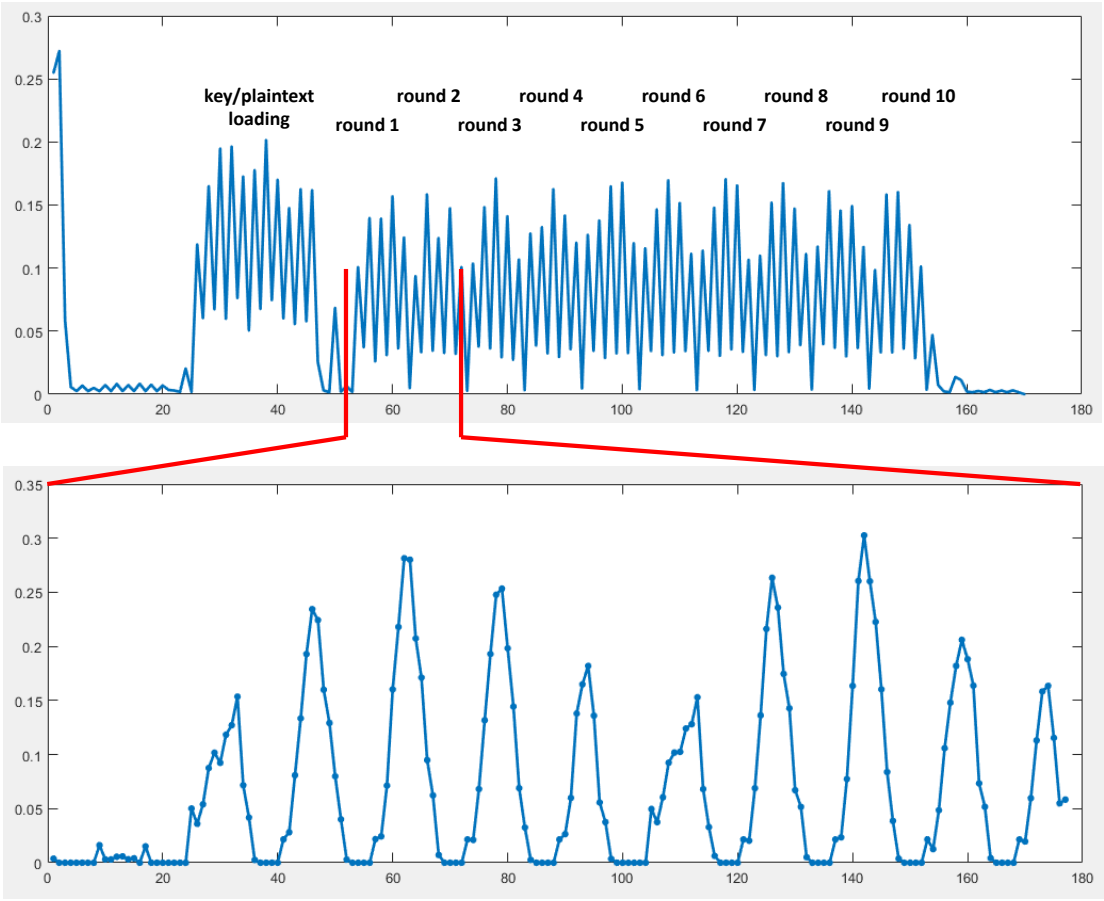


Figure 6.7: Gate-level Power simulation of an AES hardware coprocessor: (top) entire encryption at two samples per cycle (bottom) zoom on first and second round at 16 samples per cycle

a campaign (determined by the speed of gate-level simulation) and the noise level of the side-channel leakage. In a pre-silicon setting, we capture only a fraction of the traces that are collected in a post-silicon setting. On the other hand, the absence of noise implies that side-channel leakage assessment or analysis will converge much quicker.

Figure 6.7 shows the result of a gate-level simulation of the AES coprocessor in the PICO chip. The top plot is a simulation at two power samples per clock cycle, while the bottom plot is a simulation at 16 power samples per clock cycle. The resolution of the power trace can thus be easily adjusted without penalty on the noise level. Increasing the resolution

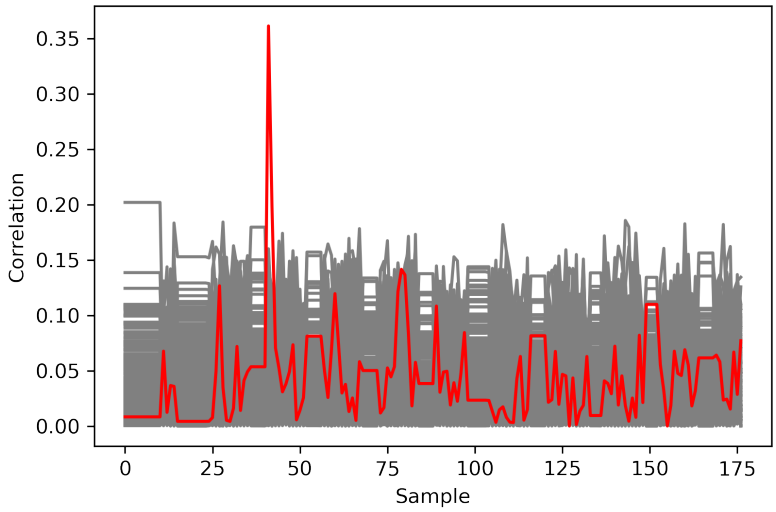


Figure 6.8: Correlation on the PICO HD AES pre-silicon trace with 16 samples per cycle

of a simulated power trace has a different effect as well: at higher resolutions, fewer gates contribute to a single power sample. This is because a gate-level simulation properly models the switching time of each gate, and at sub-cycle resolution, different gates will switch at different time instants.

Figure 6.8 shows a CPA result on 400 simulated traces, showing a sharp correlation peak in cycle 2 of round 1. While the length of a simulation trace is in principle unbounded (when compared to the limited length of a sample buffer in a post-silicon setup), in practice we aim to make the traces as short as possible to minimize the simulation time overhead. Choosing the proper time window of simulation can be a challenge when the source of side-channel leakage is unknown. On the other hand, in a pre-silicon simulation, we can undersample the power consumption, as the simulator will accumulate power over multiple cycles without adding (physical) noise or precision. The third experiment, discussed next, builds on this property.

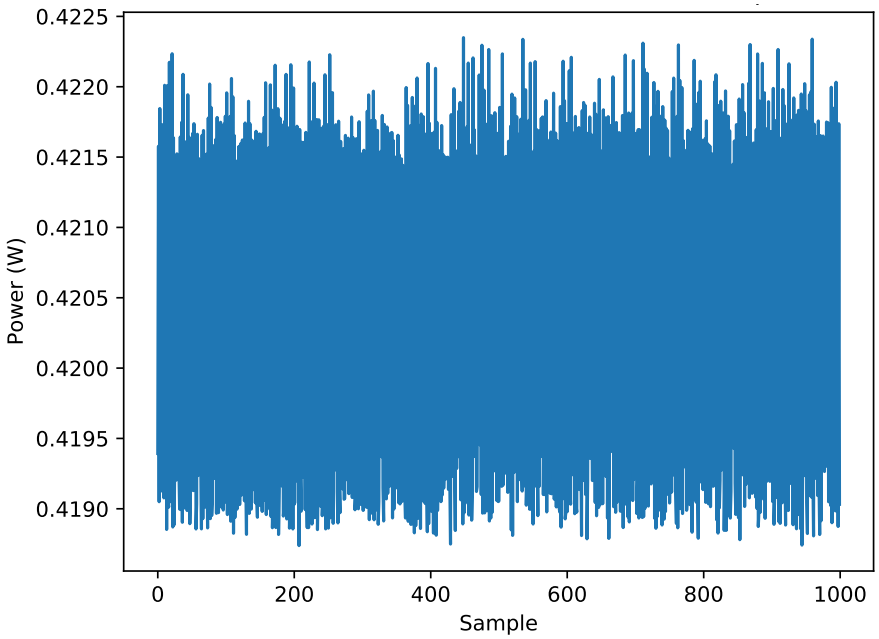


Figure 6.9: Sample simulated power trace of software AES running on PICO chip. This trace includes only the first add round key and SBox in the first round of encryption.

6.5.3 Pre-silicon evaluation of PICO SoC firmware

We experiment with the software implementation of AES encryption in ECB mode running on the PicoRV32 core in PICO chip. As we aim to target the SubBytes in the first round, we set a GPIO pin high before the first key addition and reset it to low after the SBox in first round. We use a Python script that generates random plaintexts and automatically generates the C code with hard-coded palintext values. Furthermore, we store the plaintext values in a text file for later use in side-channel analysis. We simulate the AES software running on the synthesized netlist of PICO with 180nm CMOS standard cell library. We run the simulation with a clock frequency of 80MHz and store the switching activity information in Value Change Dump (VCD) format. Using Joules, we compute the power trace for each VCD file. As an example, Figure 6.9 shows the plot of one of the generated power traces. As PicoRV32 is a non-pipelined architecture, even a small portion of the AES algorithm (first AddRoundKey and SubBytes of the first round) takes about 79k clock cycles to execute.

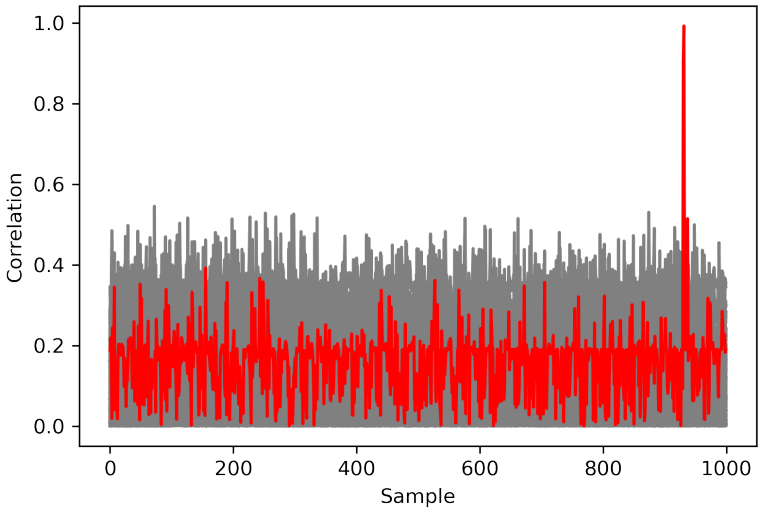


Figure 6.10: Correlation outcome on the PICO HW AES pre-silicon trace

Our power computation tool, Joules, can generate a maximum of 1000 samples per power trace. Each VCD file expands over around 990us, therefore, generating one power trace per VCD file results in a power trace with a sample rate of around 1MHz. While this is a strong under-sampled power trace for a device running at 80MHz (one sample per eighty clock cycles), CPA is able to find the private key with only 60 traces. Figure 6.10 shows the CPA result with Hamming Weight as the power model on 60 simulated traces, showing a sharp correlation peak (close to perfect correlation) in the first round SubBytes region.

With the simulation and VCD generation taking about 30 seconds and the power computation taking about 4 minutes, it took us less than five minutes to generate one power trace for our software implementation of AES. This experiment was performed as a single-thread process on an Intel Xeon Gold 6248 server.

Table 6.1: Performance factors for each of the case studies

	V.A	**V.B**	**V.C**
Pre/Post Silicon	Post	Pre	Pre
ASIC	FAME	PICO	PICO
Target	AES SW	AES HW	AES SW
Correlation Peak	0.1	0.36	0.99
Number of Traces	25,000	400	60
Samples per Cycle	4	16	1/80
Samples per Trace	24,400	200	1000
Capture Time per Trace (s)	0.06	0.55	260
Assessment Time per Byte (s)	660	<1	<1

6.5.4 Performance Evaluation

Table 6.1 summarizes our experiments. As this table shows, the correlation peaks in a software implementation are significantly higher than that of hardware implementations while also requiring orders of magnitude fewer traces even for an under-sampled trace. The capture time per trace in a post-silicon measurement is much faster than that of pre-silicon simulation, however, because of the reduced SNR in a physical implementation, a successful attack requires much more traces and thus takes more time to reveal each byte of the secret key.

6.6 Conclusion

The pre-silicon tooling of side-channel leakage is rapidly catching up with the more traditional post-silicon prototyping strategy. The main advantage of pre-silicon techniques is that design flaws can be fixed at a low cost. There are two open challenges, both of them related to the accuracy by which pre-silicon modeling can reflect post-silicon measurement. False positive errors occur when pre-silicon tooling identifies side-channel leaks that are practically unexplotable in post-silicon context. False negative errors occur when pre-silicon tooling fails

to identify exploitable leaks. Both of them are practical challenges, and will require a detailed comparison of pre-silicon models with post-silicon measurement.

Chapter 7

Leverage the Average

In this chapter, we present a methodology to reduce the power simulation time without loss of accuracy with respective to power side-channel leakage assessment. This work was presented at the Great Lakes Symposium on VLSI (GLSVLSI) in 2022 [99].

7.1 Motivation

Power side-channel attacks (PSCA) are a threat to computing systems and it is imperative that vulnerabilities of hardware products to PSCA are detected as early as possible during a chip design. To study such vulnerabilities, a designer must obtain high-resolution power traces of the design for a large number of test-vectors, and apply statistical analysis on those power traces. Two common solutions to early side-channel leakage assessment are hardware prototyping and simulation. Each solution presents unique challenges as they make a different trade-off between the speed of collecting high-resolution power traces and the fidelity of their predicted power traces to those of the actual chip. Hardware prototyping using FPGAs is a popular technique that can be applied as soon as the RTL code is available. The power traces of the actual chip under development can be predicted by measuring the

power traces of the FPGA. However, ensuring fidelity to the actual chip power traces is a challenge because of the fundamental difference between the building blocks of FPGAs (configurable logic) and those of ASIC designs (standard cell libraries). Power simulation of the actual gate-level netlist using CAD tools can ensure a higher fidelity to the actual chip under development. Furthermore, the simulated power traces are noiseless and thus represent the observation of the best-equipped attacker. A major disadvantage of simulation is that the power simulation time increases drastically at the lower abstraction levels of design. Furthermore, simulation-based power side-channel leakage assessment can show the presence of leakage in a design, however, it cannot guarantee the absence of leakage.

In this chapter, we describe a technique to decrease the power simulation time without raising the abstraction level of the simulation. Our method is to reduce the sample rate of the simulated power traces of a design, from one sample per clock cycle down to one sample over multiple clock cycles. Each of the resulting power samples thus represents the average power consumption over multiple clock cycles. We observe that reducing the sample rate in this manner significantly decreases the power simulation time. In addition, we show that the resulting traces at a reduced sample rate still return meaningful side-channel leakage assessment results. We derive the precise conditions under which this averaged sampling technique can be applied to side-channel leakage assessment. We also apply our results to several realistic case studies including a hardware crypto-coprocessor and a pipelined RISC-V processor.

Power simulation is commonly performed at three different abstraction levels: RTL, gate-level (post-synthesis/ post-layout), and transistor-level. RTL and gate-level power simulations use event-driven simulation traces in combination with power estimators, while transistor-level power simulations are based on analog simulation. We apply our averaged sampling methodology at the gate-level, as it provides a middle ground with manageable simulation time and sufficient accuracy [167]. For example, using Composite Current Source

(CCS) delay model in synthesis has shown to generate power and timing estimates comparable to that of the analog transistor-level [57]. In addition, the fidelity of gate-level power simulation can be increased by replacing the post-synthesis Standard Delay Format (SDF) file for a post-layout SDF file, generated after place and route of the chip under development, without any changes to the power simulation.

We study the averaged sampling technique as follows. In the next section, we review related work in pre-silicon side-channel leakage assessment. In Section 7.3, we describe the basic concepts of averaged sampling and explain its effect on Correlation Power Analysis (CPA), a technique for side-channel leakage assessment. Section 7.4 introduces our case studies, including a pipelined RISC-V, and a hardware coprocessor design. We then conclude this work.

7.2 Related Work

At the early stages of a hardware design, after behavioral simulation/verification, power side-channel leakage (PSCL) can be investigated at the RTL-level to find and harden vulnerable parts of the design. For this purpose, both RTL-PSC [87] and Param [90] use the switching activity from RTL simulation to model the power consumption of a design.

Šijačić *et al.* [143] introduce a simplified model of power consumption called marching stick model (MSM) for faster less accurate modeling of power side-channel. As they point out, this model is useful for RTL-level simulations and can reduce the trace generation time significantly. However, it cannot capture the effects modeled by CAD tools in post-synthesis and later stages of the design. Furthermore, they demonstrate the most time-consuming part of the PSCL at design time is the power simulation using CAD tools.

The use of CAD tools for gate-level simulation is adopted in ACA [166] and Karna [148]. ACA finds the leaky gates in the gate-level netlist and replaces them locally with hardened

structures. Karna finds the leaky parts of the implementation and replaces each gate with their lower-power versions if available in the standard cell library. Recently, the authors of Saidoyoki[92] showed simulated power traces can provide sufficient information for successful attacks with very few downsampled traces which greatly reduces the power simulation time as the bottleneck of using power simulation CAD tools in PSCL assessment at design-time.

Most pre-silicon PSCL assessment tools have explored the different abstraction layers (RTL, gate-level, transistor-level) [38]. To the best of our knowledge, our work is the first to study power simulation cost from the dimension of time resolution and its implication on pre-silicon PSCL assessment. As increasingly more researchers are emphasizing the integration of PSCL evaluation into ASIC design flow, it is crucial to study the implication of simulated power traces using CAD tools on leakage analysis.

7.3 Theoretical Background

7.3.1 Power Side-Channel Analysis

Problem Statement. PSCAs find sensitive information processed in a device by a divide and conquer approach. Even cryptographic modules with long secret keys are susceptible to such attacks. In AES-128 (128-bit secret key) for instance, a PSCA can focus on finding one byte of the key at a time and therefore reduce the search space from 2^{128} to 16×2^8. In our evaluations, we focus on CPA as the most prominent form of PSCA. However, the same conclusions can be made for other statistical moment-based measures of leakage such as DPA [110] and TVLA [22]. We specifically avoid TVLA in our evaluations to prevent the known false-positive issues related to this test [151] to affect our conclusions.

DPA. Differential power analysis (DPA) [110] reveals secret values (subkeys) based on a statistical method by comparing the means of two large sets of power traces. A key

hypothesis is assumed to be correct when the difference exhibits a distinct peak.

CPA. Correlation power analysis (CPA) [37] finds all the subkeys processed on a device by assuming a leakage model for the device. Through calculating the Pearson correlation coefficient (ρ) between the collected power dissipation from the device for random known inputs and the constructed leakage for each subkey guess, an attacker finds the correct subkey as the one resulting in the highest absolute ρ value.

7.3.2 Simulating Power Traces

Side-channel leakage assessment requires accurate modeling of the statistical properties of data-dependent power effects, and requires power consumption analysis over many different test-vectors. The data-dependent power effects that may count as side-channel leakage include data-dependent logic transitions, glitches, static leakage, propagation delays, and parasitic coupling. We use gate-level power modeling, as it is able to capture most of the data-dependent logic effects, while at the same time being much more efficient than transistor-level (SPICE-level) simulation. Time-based gate-level power modeling is supported in commercial tools such as Cadence Joules and Synopsys Primepower.

A brief review of time-based gate-level power estimation follows. A simulated power side-channel analysis campaign on a circuit collects N power traces of K frames each, for a total of $N.K$ power estimations. Each power estimation determines the average power consumption of the circuit within that frame. At gate level, the circuit power consumption includes three components: switching power, internal power, and leakage power (Figure 7.1). These factors are determined over each gate in the design, and the per-gate contributions are added up to yield circuit power. The switching power of a gate depends on the per-frame toggle rate of the output pin(s) and the capacitive load of the output pin(s). The internal gate power depends on the per-frame, per-pin input toggle rate. The gate leakage power depends on the

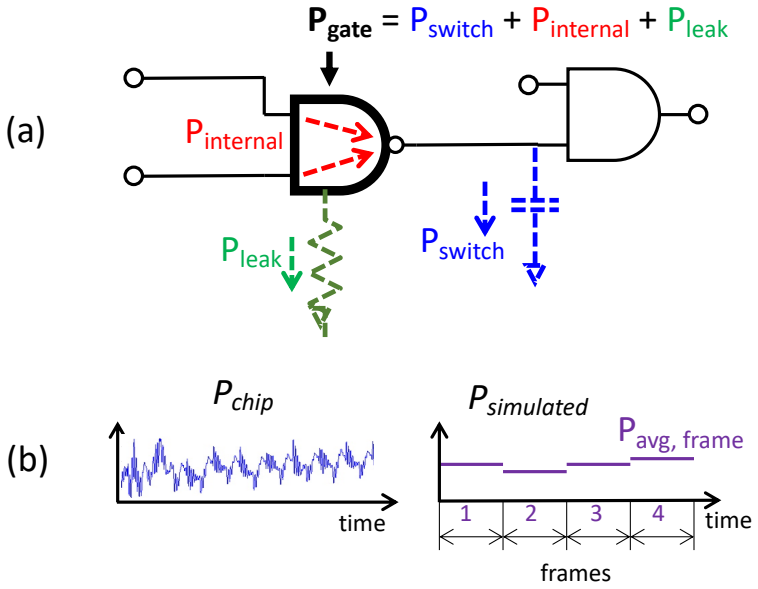


Figure 7.1: (a) Gate-level power estimation for side-channel leakage captures per-gate and per-event switching power, leakage-power and internal power. (b) Gate-level power estimation partitions time in frames and determines average circuit power per frame to construct a power trace.

Table 7.1: Normalized complexity of power estimation time for three different design sizes and four different frame widths.

Frames	**1 Counter**	**8 Counters**	**32 Counters**
1	1.0	7.5	22.7
10	1.3	8.5	32.0
100	3.6	29.7	139.3
1000	20.8	202.6	969.1

per-frame state-dependent leakage power. The power simulator uses a technology library to reflect proper scaling factors for each gate configuration and the circuit operating conditions (temperature, voltage).

The power traces in a simulated power side-channel analysis campaign are obtained in two steps. First, a logic-level simulation records the gate and net activities for each of the N test vectors in a Value Change Dump (VCD) format. Next, the VCD stimuli are analyzed for the gate-level netlist to obtain per-frame power traces.

We observe that a larger frame window size may shorten the power estimation time, and illustrate this effect in Table 7.1. Three different designs containing one, eight or thirty-two 32-bit counters are implemented in SkyWater 130nm standard cells. Next, their power consumption over a 10-million clock cycle testbench is estimated by Cadence Joules (v20.01). We computed traces containing 1, 10, 100, and 1000 frames. Table 7.1 shows a strong dependence of power estimation time over design size, which is expected: larger designs contain more events, and therefore take more time for power estimation. However, Table 7.1 also shows a strong dependence over the number of frames. Indeed, all events corresponding to a single gate or net within the same frame are accumulated into the toggle rate for that frame. Reducing the number of frames over a set of events results in reducing the number of times a power model will be computed. This observation encourages us to investigate the impact of downsampling on the quality of side-channel leakage assessment. Clearly, power traces with fewer samples provide a simulation time advantage; the question is if they are still useful to identify side-channel leakage.

7.3.3 Sampling Power Traces

In this section, we discuss non-averaged and averaged sampling. Non-averaged sampling is prevalent in power measurement using oscilloscopes where each power sample is a snapshot of the power consumption in a moment of time. Averaged sampling is how power simulation tools produce traces where each sample is the averaged power consumption of the design during the time interval of a frame. In the following, we give a mathematical model for each sampling type and study its implications on DPA and CPA as a measure of leakage assessment. A summary of the analyses is given in Table 7.2. Note that we do not perform averaging as an extra step in our proposed method. Rather, we start from the averaged traces as the output of commercial power simulators.

Table 7.2: Summary of sample types

Sampling Type	Power Model	Implications for DPA	Implications for CPA
1) Non-averaged	Equation 7.1	Theorem 1	Theorem 2
2) Averaged - Constructive	Equation 7.3	Theorem 4	Theorem 5
3) Averaged - Destructive	Equation 7.5	Theorem 6	Theorem 7

Non-Averaged Sampling

In non-averaged sampling, we use the following model for each power sample:

$$P_{n,t_s} = \alpha.\zeta(x_n, k_{c_{t_s}}) + r_{n,t_s} + \mu_r \tag{7.1}$$

where P_{n,t_s} denotes the power consumption at time sample t_s of the n^{th} power trace, α is an unknown constant, $\zeta(x_n, k_{c_{t_s}})$ is the selection function dependent on the controllable variable in trace n (x_n) and the correct key in time sample t_s. We model the algorithmic noise as a zero-mean random noise r_{n,t_s} and a constant unknown bias μ_r.

This model is similar to the model used by Fei *et al.* [65]. However, each trace may contain leakage of multiple key bytes, so that one trace can correlate with different correct key values. An example is the SubBytes function in the AES-128 algorithm, which will process 16 key bytes in every round.

Implications for DPA attack. Following the non-averaged sampling in Equation 7.1, we arrive at Theorem 1.

Theorem 1. *In non-averaged sampling, a sufficiently large number of power traces is required for a DPA attack to be successful.*

Proof. Expanding the difference of means for the power consumption at time t_s we have:

$$\begin{aligned}&\mathbb{E}_n[P_{n,t_s} \mid \zeta_n = 1] - \mathbb{E}_n[P_{n,t_s} \mid \zeta_n = 0] \\ &= \alpha + \mathbb{E}_n[r_{n,t_s} \mid \zeta_n = 1] - \mathbb{E}_n[r_{n,t_s} \mid \zeta_n = 0]\end{aligned}$$

where ζ_n is short for $\zeta(x_n, k_{c_{t_s}})$. With a sufficient number of traces, $\lim_{N\to\infty}(\mathbb{E}_n[r_{n,t_s}]) = 0$ therefore the difference of means criteria is unaffected by the algorithmic noise. □

Implications for CPA attack. Next, we derive the performance of a CPA attack on non-averaged traces.

Theorem 2. *In non-averaged sampling, a sufficiently large number of traces is required for CPA attack to be successful.*

Proof. The correlation coefficient between the selection function with key guess and the power trace is the covariance between the two sets divided by their standard deviations:

$$\rho = \frac{\sum_{j=1}^{N}\left(\left(\Delta\zeta_{j,k_g}\right)\left(P_{j,t_s} - \mathbb{E}_n[P_{n,t_s}]\right)\right)}{\sqrt{\sum_{j=1}^{N}\left(\Delta\zeta_{j,k_g}\right)^2 \sum_{j=1}^{N}\left(P_{j,t_s} - \mathbb{E}_n[P_{n,t_s}]\right)^2}} \tag{7.2}$$

where k_g is the key guess, and $\Delta\zeta_{j,k_g} = \zeta(x_j, k_g) - \mathbb{E}_n[\zeta(x_n, k_g)]$. The only term affected by the random noise in ρ is $P_{j,t_s} - \mathbb{E}_n[P_{n,t_s}] = \alpha.\Delta\zeta_{j,k_{c_{t_s}}} + r_{j,t_s} - \frac{1}{N}\sum_{n=1}^{N} r_{n,t_s}$ where $\Delta\zeta_{j,k_{c_{t_s}}} = \zeta(x_j, k_{c_{t_s}}) - \frac{1}{N}\sum_{n=1}^{N}\zeta(x_n, k_{c_{t_s}})$. Because of the dependence of P_{j,t_s} on random noise (r_{j,t_s}), the covariance term is imprecise and requires sufficiently large number of traces to converge. □

Averaged Sampling

In averaged sampling, the power sample for time interval (t_{s_1}, t_{s_2}) in the n^{th} power trace is:

$$\begin{aligned} P_{n,t_{s_1},t_{s_2}} &= \mathbb{E}_{t_s\in(t_{s_1},t_{s_2})}[P_{n,t_s}] \\ &= \alpha.\mathbb{E}_{t_s\in(t_{s_1},t_{s_2})}[\zeta(x_n, k_{c_{t_s}})] + \mathbb{E}_{t_s\in(t_{s_1},t_{s_2})}[r_{n,t_s}] + \mu_r. \end{aligned}$$

We separately analyze two scenarios in the following: 1) only one key value is processed during the time interval, 2) multiple key values are processed during the time interval.

Case 1) One key value during the time interval

If a single key value (k_{B_1}) is processed during the interval (t_{s_1}, t_{s_2}), the power sample in this interval will be:

$$P_{n,t_{s_1},t_{s_2}} = \alpha.\zeta(x_n, k_{B_1}) + \mathbb{E}_{t_s \in (t_{s_1},t_{s_2})}[r_{n,t_s}] + \mu_r. \tag{7.3}$$

Theorem 3. *With enough time samples processing the same key value, the averaged power samples are shifted by the constant bias in the algorithmic noise.*

Proof. For a sufficiently long interval, we have:

$$\lim_{\Delta_{t_s} \to \infty} \mathbb{E}_{t_s \in (t_{s_1}, t_{s_1}+\Delta_{t_s})}[r_{n,t_s}] = 0.$$

In practice, given a long enough time interval correlating to one key value, *i.e.*, $\Delta_{B_1} = t_{s_2} - t_{s_1} > \Delta_{thr}$, we have:

$$P_{n,t_{B_1},t_{B_1}+\Delta_{B_1}} \approx \alpha.\zeta(x_n, k_{B_1}) + \mu_r. \tag{7.4}$$

□

Definition 1 (constructive samples)**.** *Since power samples corresponding to the same key value make the correlation between the power trace and key value stronger by removing the random noise, we name them* ***constructive samples****.*

Implications for DPA attack. Assuming the averaged power samples we derive Theorem 4.

Theorem 4. *Given enough constructive samples in a power trace by design, DPA requires fewer traces to succeed using averaged sampling.*

Proof. Depending on the design, if there are enough constructive samples to average out the random noise ($\Delta_{B_1} > \Delta_{thr}$), power samples will follow Equation 7.4. In this case, the

difference of means criteria used in DPA requires much fewer traces to recover the key. However, if the design does not provide sufficient constructive samples ($\Delta_{B_1} < \Delta_{thr}$), the power samples follow Equation 7.3 and difference of means is:

$$\begin{aligned}
&\mathbb{E}_n[P_{n,t_{s_1},t_{s_2}} \mid \zeta_n = 1] - \mathbb{E}_n[P_{n,t_{s_1},t_{s_2}} \mid \zeta_n = 0] \\
&= \alpha + \mathbb{E}_{t_s \in (t_{s_1},t_{s_2})}(\mathbb{E}_n[r_{n,t_s} \mid \zeta_n = 1]) \\
&- \mathbb{E}_{t_s \in (t_{s_1},t_{s_2})}(\mathbb{E}_n[r_{n,t_s} \mid \zeta_n = 0]).
\end{aligned}$$

where ζ_n is short for $\zeta(x_n, k_{B_1})$. Therefore in order for DPA to be successful, a sufficiently large number of traces is required to bring the average of r_{n,t_s} close to zero. □

Implications for CPA attack. Assuming averaged sampling of constructive samples, the following theorem holds.

Theorem 5. *Given enough constructive samples in a power trace by design, averaged sampling will require fewer traces for CPA to succeed than non-averaged sampling.*

Proof. The correlation coefficient between the selection function with key guess and the power trace is the same as Equation 7.2 replacing the power samples with Equation 7.3. Given enough constructive samples to be averaged in the interval (t_{s_1}, t_{s_2}), we have:

$$P_{j,t_{s_1},t_{s_2}} - \mathbb{E}_n[P_{n,t_{s_1},t_{s_2}}] = \alpha.\left(\zeta(x_j, k_{B_1}) - \mathbb{E}_n[\zeta(x_n, k_{B_1})]\right).$$

Therefore, the correlation is not affected by the algorithmic noise and CPA can converge with fewer traces.

However, without enough constructive samples, we have:

$$\begin{aligned}
P_{j,t_{s_1},t_{s_2}} - \mathbb{E}_n[P_{n,t_{s_1},t_{s_2}}] = \alpha.\left(\zeta(x_j, k_{B_1}) - \mathbb{E}_n[\zeta(x_n, k_{B_1})]\right) \\
+ \mathbb{E}_{t_s \in (t_{s_1},t_{s_2})}[r_{j,t_s} - \mathbb{E}_n[r_{n,t_s}]].
\end{aligned}$$

Therefore a sufficiently large number of traces is required for the random noise to reach its zero mean and render CPA successful. □

Case 2) Multiple key values during the time interval

We assume key value k_{B_1} is processed in the first part of the interval (t_{s_1}, t_{s_m}) and other key values from the set $\{k_{B_1}\}'$ ($k_{B_1} \notin \{k_{B_1}\}'$) are processed in the rest of the interval (t_{s_m}, t_{s_2}). To further simplify the analysis, we assume the time samples in interval (t_{s_m}, t_{s_2}) correspond to processing the same key value $k'_{B_1} \neq k_{B_1}$. The power sample in interval (t_{s_1}, t_{s_2}) can be written as:

$$\begin{aligned} P_{n,t_{s_1},t_{s_2}} &= \alpha.\zeta(x_n, k_{B_1}) + \alpha.\zeta(x_n, k'_{B_1}) \\ &+ \mathbb{E}_{t_s \in (t_{s_1},t_{s_2})}[r_{n,t_s}] + \mu_r. \end{aligned} \tag{7.5}$$

Definition 2 (destructive samples)**.** *Since the power samples corresponding to different key values being processed makes the correlation between the power trace and the secret key value weaker, we name them* ***destructive samples****.*

Implications for DPA attack. Assuming more than one key value is processed in a power trace, we derive Theorem 6.

Theorem 6. *Averaging destructive time samples diminishes the success probability of DPA attack even with a large number of power traces.*

Proof. Following the simplified averaged sample in Equation 7.5, the difference of means

used in DPA is:

$$\begin{aligned}
&\mathbb{E}_n[P_{n,t_{s_1},t_{s_2}} \mid \zeta_n = 1] - \mathbb{E}_n[P_{n,t_{s_1},t_{s_2}} \mid \zeta_n = 0] \\
&= \alpha + \alpha.(\zeta'_n \mid_{\zeta_n=1} - \zeta'_n \mid_{\zeta_n=0}) \\
&+ \mathbb{E}_{t_s \in (t_{s_1},t_{s_2})}(\mathbb{E}_n[r_{n,t_s} \mid \zeta_n = 1]) \\
&- \mathbb{E}_{t_s \in (t_{s_1},t_{s_2})}(\mathbb{E}_n[r_{n,t_s} \mid \zeta_n = 0])
\end{aligned}$$

where ζ_n and ζ'_n are short for $\zeta(x_n, k_{B_1})$ and $\zeta(x_n, k'_{B_1})$ respectively. Since $\zeta'_n \mid_{\zeta_n=1}$ and $\zeta'_n \mid_{\zeta_n=0}$ are not predictable, even with a sufficiently large number of traces for the random noise to reach its mean value of zero, still the DPA can be inconclusive. □

Implications for CPA attack. Similarly, Theorem 7 is derived for CPA attack.

Theorem 7. *Averaging destructive samples diminishes the success probability of CPA attack even with a large number of power traces.*

Proof. The correlation coefficient between the selection function and the power trace is the same as the previous case. Similarly, following the simplified averaged sample in Equation 7.5, we have:

$$\begin{aligned}
P_{j,t_{s_1},t_{s_2}} - \mathbb{E}_n[P_{n,t_{s_1},t_{s_2}}] = \alpha.\Delta\zeta_{j,k_{B_1}} + \alpha.\Delta\zeta_{j,k'_{B_1}} \\
+ \mathbb{E}_{t_s \in (t_{s_1},t_{s_2})}[r_{j,t_s} - \mathbb{E}_n[r_{n,t_s}]]
\end{aligned}$$

where $\Delta\zeta_{j,k_B} = \zeta(x_j, k_B) - \mathbb{E}_n[\zeta(x_n, k_B)]$. Therefore, even with enough traces to cancel out the random noise, the term $\alpha.\Delta\zeta_{j,k'_{B_1}}$ greatly impairs the correlation. □

7.3.4 Empirical verification of theorems

Setup. Following the assumption that at each clock trigger there is a rush of current in the circuit, we generate power traces in the shape of sawtooth function where the amplitude is

a function of the current level and the period is similar to the system clock period. This model follows the so-called Delta-I noise [81] and is similar to the model used by Tiran et al. [155]. SCA attacks, like DPA and CPA, exploit the dependency of the power consumption of a circuit on the secret data as well as a controlled data. To make our generated power traces applicable to such attacks, we further make the amplitude of the samples in each trace dependent on a function of a secret and a controlled byte. For our traces, we choose $f_{n,t_s} = xor(x_n, k_{t_s})$ where x_n is the controlled value for the n^{th} trace and k_{t_s} is the secret value for the time sample t_s. Each trace corresponds to one controlled value and contains 64 clock cycles. This is similar to measured traces in practice where an attacker feeds a device with known inputs and acquires one trace for each given input. Every 16 consecutive clock cycles correspond to one key byte value (constructive). These traces are interpolated in Matlab up to a continuous-time power trace such that we can create both a non-averaged sample trace (oscilloscope) as well as an averaged sample trace (CAD power simulation) from the same source version. We generate 256 such traces, one for each controlled byte value.

Experiment 1 - averaged vs. non-averaged sampling. For each of the 256 traces, we generate noisy continuous-time traces with Signal-to-Algorithmic Noise Ratio (SANR)[1] values of {10, 5, 1, 0.5, 0.25, 0.1}. For each SANR value we create two power traces from the same source data: a non-averaged sampled version with two samples per clock cycle, and an averaged sampled version with two frames per clock cycle. We run CPA on the two sets of noisy traces for each key byte separately and calculate the rank of the correct key value. We repeat the same process of noisy trace generation, sampling, and CPA 100 times and report the average correct key rank and average success rate (SR) calculated as $\text{SR} = \frac{\text{number of successful attacks}}{\text{total number of attacks}}$ in Figure 7.2.

This experiment shows a substantial improvement of averaged traces over non-averaged

[1]We differentiate between the SNR pertaining to the measurement noise and the SNR pertaining to the algorithmic noise, calling the latter Signal-to-Algorithmic Noise Ratio (SANR). Algorithmic noise is any signal not correlated with the target of the attack.

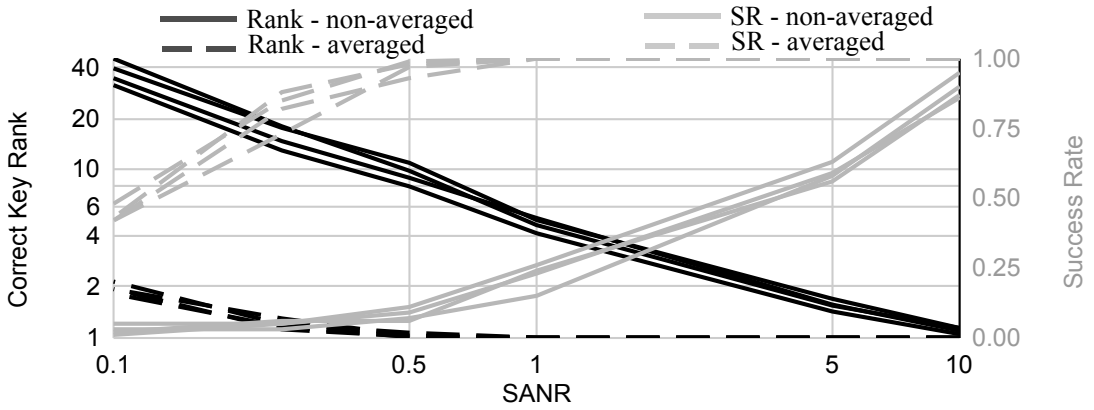


Figure 7.2: Correct key rank and SR of CPA attack on 256 noisy traces with different SANR values for the four different key bytes. Solid (resp. dashed) lines show results for non-averaged (resp. averaged) sampled traces. Averaged over 100 runs.

traces. For example, attacks on averaged traces approach a success rate of 1 as soon as SANR> 0.5, while non-averaged traces achieve the same success rate only at SANR> 10.

Experiment 2 - constructive samples. For the same set of power traces with SANR=0.1 (weakest SANR), we average up to 16 clock cycles corresponding to the same key byte (constructive samples). We report the SR of CPA attack to compare the different levels of constructive averaging in Table 7.3. As this table shows, as we average more constructive samples, it becomes more likely to find the correct key.

Table 7.3: SR of CPA on key bytes in simulated traces for different number of averaged constructive samples with SANR= 0.1 (averaged over 100 runs).

# Constructive	SR byte 1	SR byte 2	SR byte 3	SR byte 4
2	0.68	0.61	0.68	0.66
4	0.78	0.78	0.79	0.71
8	0.83	0.92	0.83	0.81
16	0.92	0.96	0.9	0.95

Experiment 3 - destructive samples. We average 16 constructive cycles for each key byte and add up to 16 destructive cycles to the averaged sample to study the effect of destructive samples on CPA results. Table 7.4 shows the SR decreases as more destructive

samples are added, attesting to Theorem 7.

Table 7.4: SR of CPA on key bytes in simulated traces for different ratio of destructive to constructive samples with SANR=0.1 (averaged over 100 runs).

$\frac{\#\text{ Destructive}}{\#\text{ Constructive}}$	SR byte 1	SR byte 2	SR byte 3	SR byte 4
0	0.93	0.93	0.91	0.9
0.25	0.87	0.79	0.86	0.86
0.5	0.58	0.6	0.68	0.68
0.75	0.39	0.36	0.62	0.57
1	0.04	0.03	0.06	0.07

7.4 Case Studies

In the following case studies, we show how different hardware designs affect the level of averaging that can be done without loss of precision. Our case studies contain a pipelined microprocessor running software AES, and an AES hardware accelerator. We chose AES-128 as its vulnerabilities are well-understood in the literature, and therefore this case makes a good driver for our experiments.

7.4.1 Case Study 1: Software AES on a Pipelined Processor

For our pipelined processor, we use the five-stage BRISC-V[2] implementation of RISC-V RV32I ISA (integrated into a system-on-chip [91]). The five stages in this core are instruction fetch, instruction decode and operand access, execution, memory access, and write back. We synthesize the design for the open standard cell library SkyWater 130nm and 50MHz frequency using Cadence Genus. In each gate-level simulation, we load the binary file of the compiled AES software code into the program memory of the processor, feed a new plaintext into the simulation, and collect the vcd file. We then simulate the power traces of

[2] `https://ascslab.org/research/briscv/index.html`

the SubBytes step in the first round using Cadence Joules. In the leakage assessment step, we run CPA on the simulated traces with a leakage model of the Hamming weight of the first round SBox output, *i.e.*, `HW(SBox(k⊕p))` where `k` is the key byte and `p` is the corresponding plaintext byte.

Listing 7.1 shows the assembly code for SubBytes running on RISC-V. In this code, the SBox is implemented as a table look-up. Iterating 16 times by two nested loops, line 10 loads the SBox result for each state byte and line 11 stores the result in the memory location of the state byte. According to the tested leakage model (`HW(SBox(k⊕p))`), both lines 10 and 11 will leak. The leakage will stem from any sequential or combinatorial part of the processor handling the SBox output.

Listing 7.1: Assembly code of AES SubBytes for RISC-V

```
1  250:  lui   a1,0x1
2  254:  addi  a1,a1,1032  # state address
3  258:  addi  a3,a1,4
4  25c:  addi  a1,a1,20
5  260:  lui   a2,0x1
6  264:  addi  a2,a2,212   # sbox address
7  268:  addi  a5,a3,-4    # start loop 1
8  26c:  lbu   a4,0(a5)    # start loop 2
9  270:  add   a4,a2,a4
10 274:  lbu   a4,0(a4)    # load sbox byte
11 278:  sb    a4,0(a5)    # store sbox byte
12 27c:  addi  a5,a5,1
13 280:  bne   a5,a3,26c   # end loop 2
14 284:  addi  a3,a3,4
15 288:  bne   a3,a1,268   # end loop 1
```

Figure 7.3 demonstrates the data dependency of instr. 11 on instr. 10 causing a stall in cycle 3 (data dependency between decode and execute stage in cycle 2) and another stall in cycle 4 (data dependency between decode and memory stage in cycle 3). Furthermore, there is a one-cycle delay for memory load instructions, which causes instr. 10 to stall

Clock Cycle	Fetch	Decode	Execute	Memory	Write Back
1	F11	D10	-	-	W9
2	F12	D11	E10	-	-
3	stall	stall	flush	**M10**	-
4	stall	stall	stall	**M10**	flush
5	F12	**D11**	stall	stall	**W10**
6	F13	D12	**E11**	stall	stall
7	F14	D13	E12	**M11**	stall
8	stall	stall	flush	M12	W11

Figure 7.3: Flow of instructions in Listing 7.1 through five stages of RISC-V pipeline. Highlighted cells show the leaking parts.

in the memory stage cycle 4 and subsequently flush in the write-back stage. Once instr. 10 reaches the write-back stage, its result is forwarded to instr. 11 in the decode stage. Consequently, this pipeline diagram demonstrates that cycles 3 through 7 cause leakage of the `HW(SBox(k⊕p))` during the highlighted cells. Therefore, power samples taken from these cycles are constructive samples for our leakage model. Since we have a 5-cycle window of constructive samples and we don't control the starting point of averaged samples, we average 3 clock cycles to ensure only constructive samples are averaged.

We simulate 1k power traces for RISC-V running the aforementioned assembly code and simulated traces with 1 sample per clock cycle (1s1cc) and 1 sample per 3 clock cycles (1s3cc). CPA on both trace sets resulted in $\rho > 0.999999$ for every key byte. The achieved high correlation despite the downsampled traces is thanks to the constructive samples. Therefore, using the downsampled 1s1cc and 1s3cc traces, we can still find the leakage with highest possible correlation while reducing the power simulation time by 1.86×.

Additionally, we simulate traces with even fewer samples (1 sample every 10, 20, 30, 40, and 50 cycles) to compare their correlation results in CPA. Although CPA is still able to reveal all key bytes successfully, the correlation coefficients for the heavily downsampled traces (1s10cc, 1s20cc, 1s30cc, 1s40cc, and 1s50cc) reduce. We can use heavily downsampled

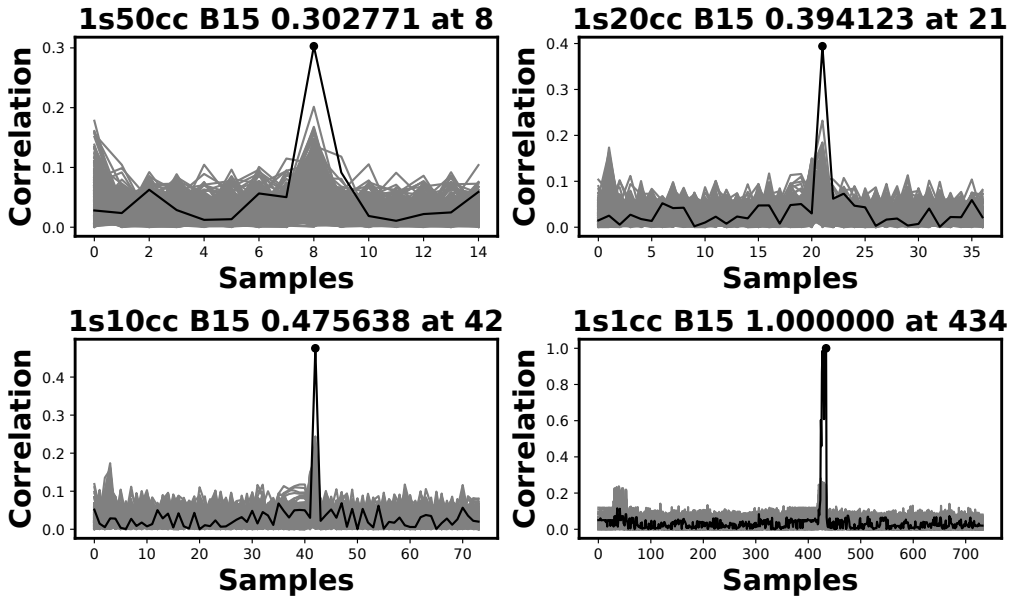


Figure 7.4: CPA ρ vs. sample number for locating the power leakage of last key byte (B15) from coarse (1s50cc) to finer (1s1cc) traces for software AES running on RISC-V. Grey lines are correlation values for incorrect key guesses and black line is the correlation value for correct key guess.

traces to quickly locate the leaky time periods and gradually increase the sample rate on the leaky regions to find more localized leaky points and therefore reduce the simulation time. For instance, Figure 7.4 shows correlation coefficients from CPA on the last key byte of AES running on RISC-V. Table 7.5 shows the performance gain from increasing number of averaged samples for the same number of traces (normalized to 1s1cc).

Table 7.5: Speed-up of averaged RISC-V traces compared to 1s1cc

Num. of Averaged Clock Cycles	1s10cc	1s20cc	1s30cc	1s40cc	1s50cc
Simulation Speed-up	3.35×	4.86×	5.45×	5.78×	6.50×

7.4.2 Case Study 2: Hardware AES

In the next experiment, we shift our focus from constructive samples to destructive samples. We consider a hardware accelerator for AES-128 that runs 4 SBoxes in parallel in each

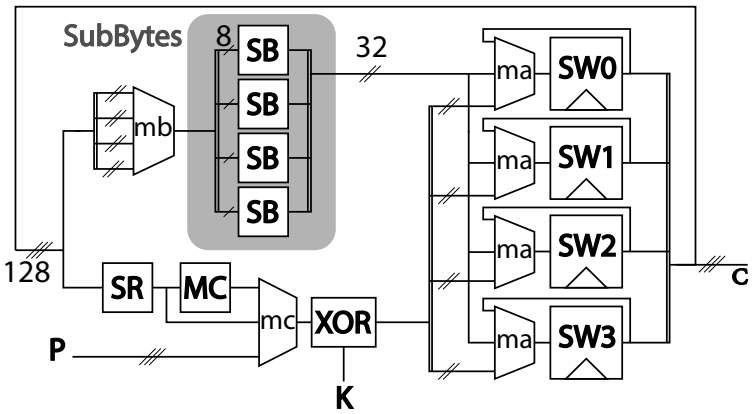


Figure 7.5: Data path of the analyzed AES hardware accelerator. SB: SBox, SR: ShiftRows, MC: MixColumns.

clock cycle, therefore, destructive samples are intrinsically present in each power sample. Figure 7.5 shows the structure of the AES implementation. Each AES round is executed in 5 clock cycles. The first four clock cycles each execute 4 SBoxes (grey shade in Figure 7.5). The `ma` (resp. `mb`) multiplexers choose which 32bit part of the state (SW0 through SW3) should be updated (resp. calculated) through the SBox modules. The last clock cycle executes the rest of the blocks in succession (`SR`, `MC`, and AddRoundKey) following which `ma` multiplexers choose the appropriate update values for each state word. `mc` sets the correct input for the AddRoundKey step, *i.e.*, plaintext (`P`) at the start, `MC` output for the first 9 rounds, and `SR` output for the last round. We synthesize the AES module for SkyWater 130nm standard cell library and 50MHz clock frequency using Cadence Genus and simulate 1k downsampled (1s1cc, 1s2cc, 1s3cc, 1s4cc) post-synthesis gate-level power traces using Cadence Joules.

In the first round of AES, each `SW` register will overwrite the first AddRoundKey output with its corresponding SBox output. Therefore, for CPA, we use `HD(k⊕p,SBox(k⊕p))` as the leakage model. Figure 7.6 shows the CPA correlation result for the last subkey as an example. The correlation values have drastically reduced and in some cases (e.g. 1s4cc for subkey 15) the CPA fails. This example shows that a secure hardware designer should

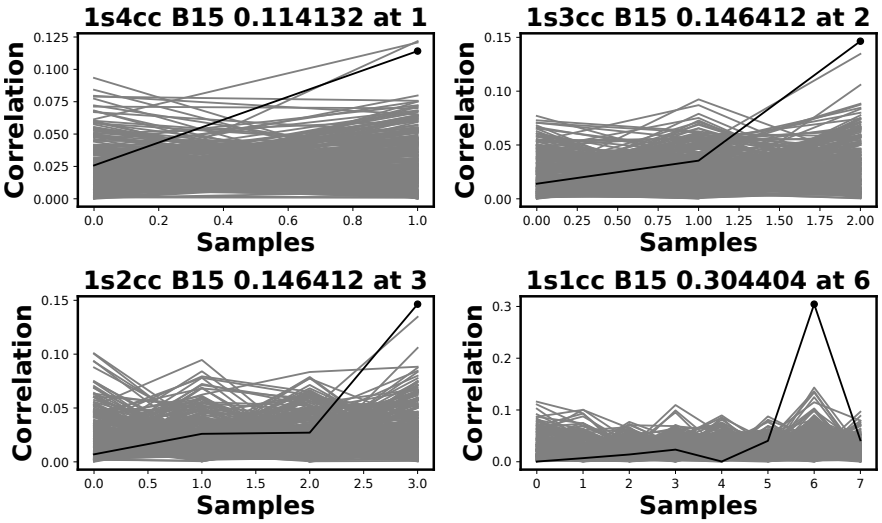


Figure 7.6: CPA ρ vs. sample number on last key byte (B15) for AES accelerator. Grey lines are correlation values for incorrect key guesses and black line is the correlation value for correct key guess.

avoid combining destructive samples in the power analysis stage to prevent false negative conclusions in the assessments.

Discussion. As shown experimentally and theoretically, a secure hardware designer can take advantage of constructive samples in a design to downsample traces and reduce the power simulation time which is the bottleneck for pre-silicon PSCL assessment. At the same time, the designer should be aware that averaging destructive samples can lead to false negative conclusions in leakage assessment. Fortunately, classifying power samples as constructive or destructive is straightforward for well-known algorithms and implementations such as the ones analyzed in our case studies. However, additional preprocessing may be needed for unknown designs. For instance, using specific test vectors that only exercise part of the key and applying statistical tests to tag a given power sample as destructive or constructive. In our future work, we study techniques that can help a designer classify power samples as such for certain leakage criterion.

7.5 Conclusion

We studied the implication of averaged sampling for pre-silicon side-channel leakage assessment. We introduced the concept of constructive and destructive samples as categories of power samples which will facilitate or hinder the leakage assessment of a certain sensitive data. We showed how using downsampling in designs for which the constructive samples are well understood aid in reducing power simulation time without significant loss in precision. In future work, we will study tests that can categorize samples as constructive or destructive to be applied to less-known designs.

Chapter 8

Generic Gate-Level Power Side-Channel Leakage Assessment

In this chapter, we introduce a methodology to find power side-channel leakage at design-time using simulated power traces at gate-level and rank the gates in the design according to their contributions to the leakage. This work is under review at IEEE Transactions on Emerging Topics in Computing [105].

8.1 Introduction

Power-based side-channel leakage occurs when a secure chip performs operations that depend on an internal secret value such as a secret key. An adversary who observes the chip power consumption can derive the internal secret value through differential analysis techniques that correlate a power model of the secret activity with the observed power consumption. In recent years, side-channel vulnerabilities have risen to prominence and successful side-channel attacks have been demonstrated on a wide range of devices from small IoT devices to large cloud computing systems. Therefore, the evaluation of the power-based side-channel leakage

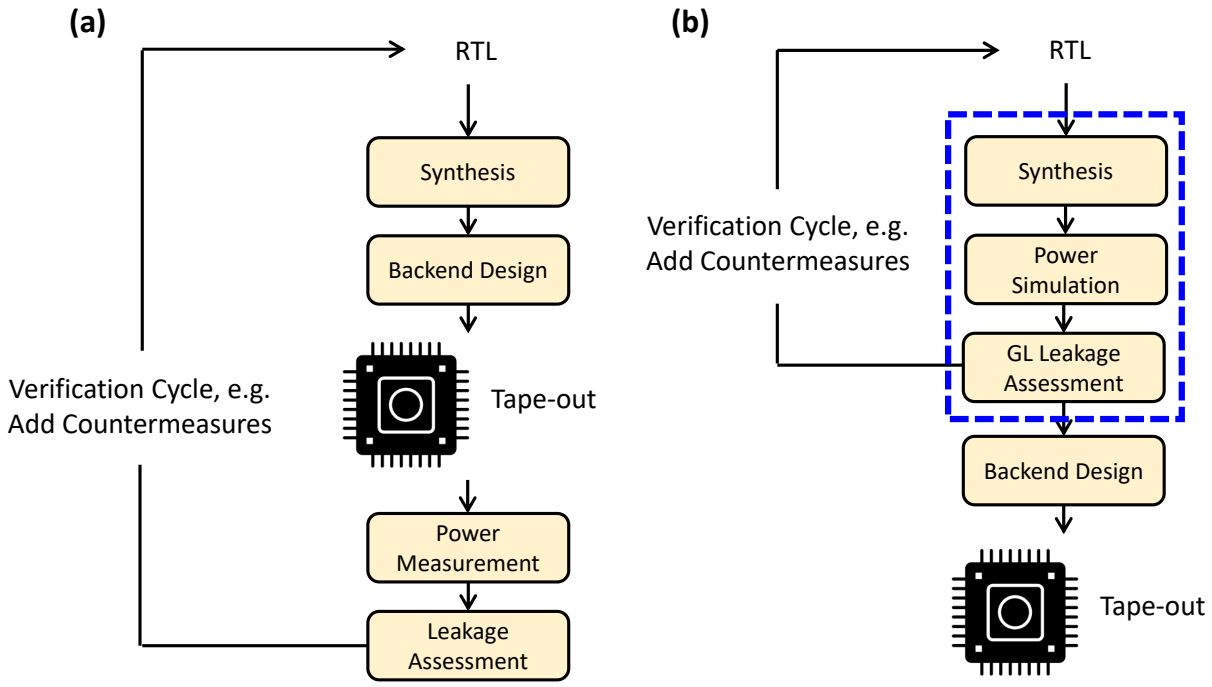


Figure 8.1: (a) Post-silicon side-channel leakage assessment flow. (b) Proposed flow.

has become a critical component in the design flow of secure chips. It is particularly helpful to perform side-channel leakage assessment prior to manufacturing because it reduces the cost of post-manufacturing testing, and it reduces the probability of side-channel vulnerabilities in the chip tape-out.

Figure 8.1 compares the Side-channel Leakage Assessment (SLA) process of a post-silicon assessment flow with a pre-silicon assessment flow. The starting point is identical in both cases and assumes that a RTL description of the design under consideration is available. With a post-silicon SLA flow, the RTL design is first prototyped into a physical implementation. Power measurements are then collected from the design and statistically tested to confirm the presence of side-channel leakage or to estimate the quantity of side-channel leakage. In a pre-silicon strategy, the RTL design is synthesized into a gate-level netlist including optional parasitic effects from place-and-route. Next, power traces are simulated and then statistically tested to confirm the presence of side-channel leakage.

These two flows appear similar from a macro-level objective, but they have very different properties. A post-silicon flow is expensive because of the extra prototyping step, which slows down the verification cycle. The statistical tests are applied globally on the measured leakage of the overall design. In a large and complex design, it therefore remains difficult to pinpoint the leakage source.

In contrast, a pre-silicon flow makes use of simulated power traces, and it is able to perform side-channel leakage assessment at a fine granularity. In this paper, we present a side-channel leakage assessment methodology with a resolution of a single gate. The simulated traces of the pre-silicon flow are noiseless, and therefore they represent the attacker with the best possible observation. Due to the absence of noise, a pre-silicon flow can work with a fraction of the number of power traces compared to a post-silicon flow. On the downside, a pre-silicon flow must make a trade-off between accuracy and simulation speed. We use a gate-level power simulation methodology that is able to capture many technology-dependent effects (such as glitches [127] and static leakage [125]). Some side-channel leakage effects, including those based on coupling [42] or the long-wire effect [74], require a simulation accuracy beyond what gate-level power simulation can offer. Our proposed flow offers gate-level side-channel leakage assessment but makes no assertion of leakage below that abstraction level.

This paper presents the following contributions. We describe a methodology called *Architecture Correlation Analysis* (ACA) which determines the side-channel leakage of a design at the granularity of a single gate. The basic principle of ACA has been initially proposed in our previous work [63, 170]. This paper serves as an extension to our original publication. In this extended work, we propose two different side-channel leakage assessment techniques for use in ACA. The first one is based on a specific test and it demonstrates the presence of correlation between a specific power model and individual logic gates. The second is based on a non-specific test that ranks a gate's power according to its ability to distinguish between

two distinct groups of test vectors. The specific test is used to identify the gates that enable a specific side-channel attack, while the non-specific test is used to make a generic assessment on how much potentially harmful leakage can be produced by a gate. We apply our proposed ACA leakage assessment technique to two case studies: a cryptographic AES coprocessor, and the driver software for that coprocessor when running on a five-stage pipelined RISC-V processor. The side-channel leakage properties of AES are already well understood, and our experiments are specifically aimed at the ability of pre-silicon leakage assessment to identify the source of side-channel leakage.

The remainder of the paper is organized as follows. The next section describes related work. Section 3 provides the overall methodology of Architecture Correlation Analysis (ACA), highlighting both the specific and the non-specific testing strategy. We also discuss a prototype implementation of the flow. Section 4 applies our proposed methodology to a cryptographic coprocessor. Section 5 applies the methodology to cryptographic driver software running on a RISC-V processor. Section 6 evaluates the performance of the proposed methodology. We then conclude the paper.

8.2 Related Work

Gate-level side-channel leakage assessment is built on two components of design automation: (a) power simulation under a set of selected test vectors, and (b) identification of a leakage source at the sub-module level or gate level. We discuss related work on each of these two aspects.

8.2.1 Power simulation for side-channel leakage analysis

To simulate a design's power consumption, one needs a model of the design implementation details to estimate the power of the physical implementation under a set of test vectors. The

model can be constructed at different abstraction levels, and there is a trade-off between modeling detail and simulation performance. Buhan *et al.* review many of the recent proposals to simulate side-channel leakage [39] and here we only describe the most representative ones.

The origin of side-channel leakage power simulation is found in simulators for smart cards, starting with PINPAS [50]. The objective of these instruction-level simulations is to generate a power trace corresponding to a software application running on an embedded micro-controller. These simulators are processor-specific, and require knowledge of the internal design of the processor. Recent research efforts have addressed power model construction techniques to handle the case when the internal design is unknown. This includes ELMO [122] and ROSITA [142] for power, and EMSIM [139] for Electromagnetic Radiation. In our approach, we build on the basic assumption that the hardware source code is available.

It is now commonly understood that instruction-level power modeling by itself is inadequate to accurately capture all aspects of power-based side-channel leakage, and that additional modeling detail is required to capture circuit-level effects. The CASCADE power simulation flow aims at a comprehensive simulation of power traces at the gate-level [144], while making the argument that gate-level power simulation hits a sweet spot for the known power-based side-channel leaks. A similar flow (and argument) is found in SCRIPT [126]. However, transistor-level power simulation has been investigated as well to address specific side-channel leakage assessments with a limited scope in time and in design size [133]. More recently, power simulation for side-channel analysis is starting to appear in commercial tooling.

8.2.2 Identification of the leakage source

By simulating power with a structural model, it becomes feasible to identify the *source component* of side-channel leakage. In traditional measurement-based side-channel leakage analysis, this type of analysis is not possible because the design under test remains a black box. We review several recent proposals aimed at identifying the structural source of side-channel leakage, starting at low abstraction levels.

Karna partitions the gates of a design according to a spatial grid over the circuit layout [61]. A gate-level simulation leads to a power trace per grid cell. A leakage metric then ranks different cells according to their contribution to the side-channel leakage. The resolution of the Karna method depends on the granularity of the grid cells on the layout, since all logic cells within the same grid cell receive the same leakage score. The Karna authors include around 150 logic cells in one grid cell. Another tool, RTL-PSC. works at the register-transfer level of abstraction [88]. RTL-PSC analyzes the power in terms of transitions on the state variables of a design, while sub-cycle effects such as glitches and the effects of physical routing are abstracted out. The leakage metric ranks the different modules of a design. A similar register-transfer level analysis tool is PARAM [64], where the authors identify the sources of side-channel leakage in a processor's micro-architecture.

Another view on the problem of leakage source identification is to formally prove that a design meets a predefined side-channel leakage criteria. This technique works well for designs based on masking, a countermeasure based on secret-sharing. One example is Coco, which combines event-driven simulation with a SAT-solver-based verification of the statistical distribution of the secret shares [76]. Strictly speaking, these tools do not simulate the power consumption, but they verify the statistical properties of design activity.

Compared to this related work, ACA creates a ranking of the cells in the design according to their contribution to side-channel leakage, with a user-defined leakage metric. ACA can

handle both specific and non-specific leakage testing. ACA is processor-independent as well as technology-independent. ACA builds a flow on commercially available synthesis and power simulation tooling. To the best of our knowledge, no such tool has been presented by earlier work.

8.3 Architecture Correlation Analysis

In this section, we outline the strategy of Architecture Correlation Analysis. The objective of ACA is to identify a ranking among the cells[1] of a design according to their contribution to side-channel leakage. The cell ranking does not have to reflect the absolute level of side-channel leakage generated by a cell. Knowing just a relative ranking already provides critical insight into the parts of a design that are most prone to side-channel attacks.

Traditional side-channel leakage assessment uses the overall power consumption of a design to make an assessment on the global design. In contrast, ACA uses not only the overall power consumption but also the internal design construction details to make an assessment of leakage on a *part* of a design and to rank these local assessments. ACA uses gate-level power simulation in order to capture power events with sub-cycle accuracy as well as structural effects such as wire-loading and static leakage. The challenge of ACA is to perform such a gate-level side-channel leakage assessment with reasonable accuracy but *without* exhaustively generating the power consumption trace for each individual cell in the design.

Side-channel leakage assessment aims to minimize assumptions regarding the specific strength and know-how of the attacker. This leads to the use of specific and non-specific tests. A *specific* test for side-channel leakage uses a high-level power model that the adversary would presumably use in a Differential Power Analysis. A *non-specific* test uses two groups of

[1]We use the term *cell* over *gate* as it reflects better the technology encountered in standard-cell based IC design. A single cell often corresponds to multiple primary gates.

inputs and aims to demonstrate a statistically distinguishable power consumption difference between those two groups. The Test Vector Leakage Assessment (TVLA) methodology provides guidance on the selection of the two groups of input vectors [58]. Both specific and non-specific tests have their merits and limitations. Specific tests are limited by specific assumptions on the capabilities and activities of the attacker, but provide specific assertions on the existence of a side-channel attack. Non-specific tests avoid such assumptions, but they are unable to assert the existence of a side-channel attack that can exploit the leakage. We, therefore, present an ACA methodology for either approach.

8.3.1 Overall Methodology

The ACA methodology includes three phases: (a) activity trace and power trace generation, (b) leakage time interval selection, and (c) leakage impact factor evaluation. The first phase is common to specific and non-specific tests, while the second and third phases differ according to the testing strategy. We will discuss each phase separately.

Figure 8.2 describes the common first phase of specific and non-specific ACA, covering logic synthesis, gate-level simulation, and gate-level power simulation. The design parameters, shown in italic, include the testing strategy, the target technology, the target cycle period, and the frame size. The frame size is the time step used in the traces of the power simulator. All of the intermediate results of the flow are used by later phases of ACA.

Logic synthesis transforms the input RTL under a given performance constraint (speed/area) into a gate-level netlist. Next, the gate-level netlist is simulated for a set of test vectors while recording the circuit activity for each net in the design over the simulation time window of interest. The type and number of test vector stimuli depend on the assessment type (specific/non-specific) and the acceptable statistical uncertainty of the leakage assessment result. We will address the selection of test vectors in the next subsections.

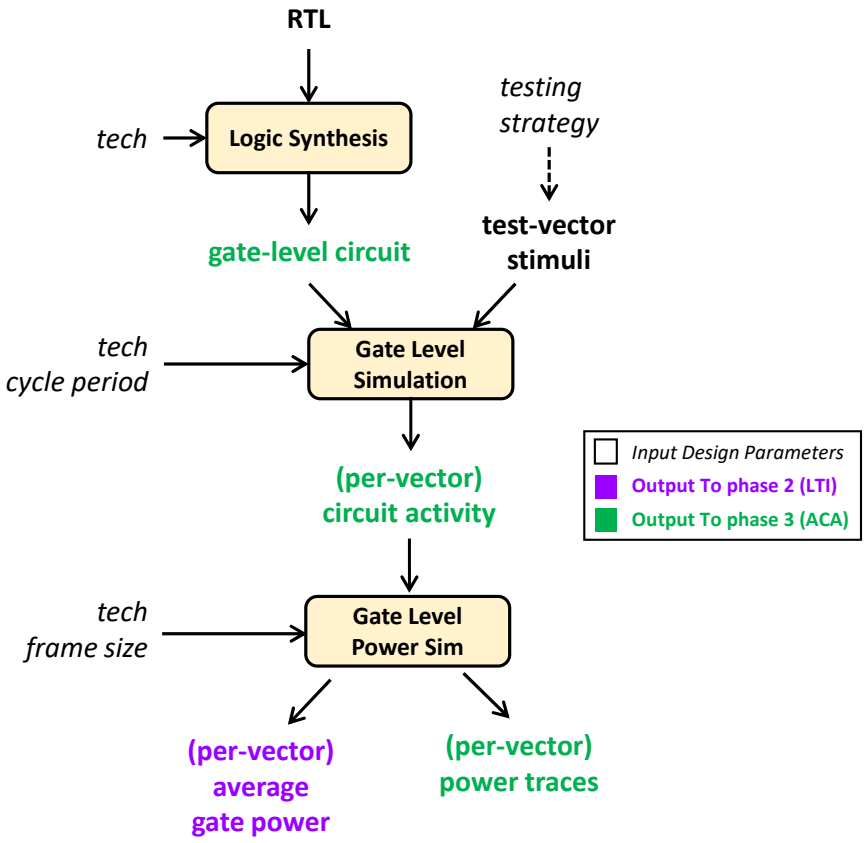


Figure 8.2: ACA Phase 1: Stimuli and Trace Generation for ACA

Two factors greatly help in reducing the number of test vectors for a side-channel leakage assessment. The first factor is that power simulation is noiseless so that each test vector and internal design state needs to be simulated only once. The second factor is that each simulation run can be isolated from the next by re-initializing the design in between simulation runs. In a typical side-channel leakage assessment, we gather between a few hundred and a few thousand simulated traces and we aim for a turn-around of all three phases of ACA in less than 24 hours.

The gate-level power simulation estimates the time-varying power consumption of the design for each test vector as follows. First, the simulation time window is partitioned into

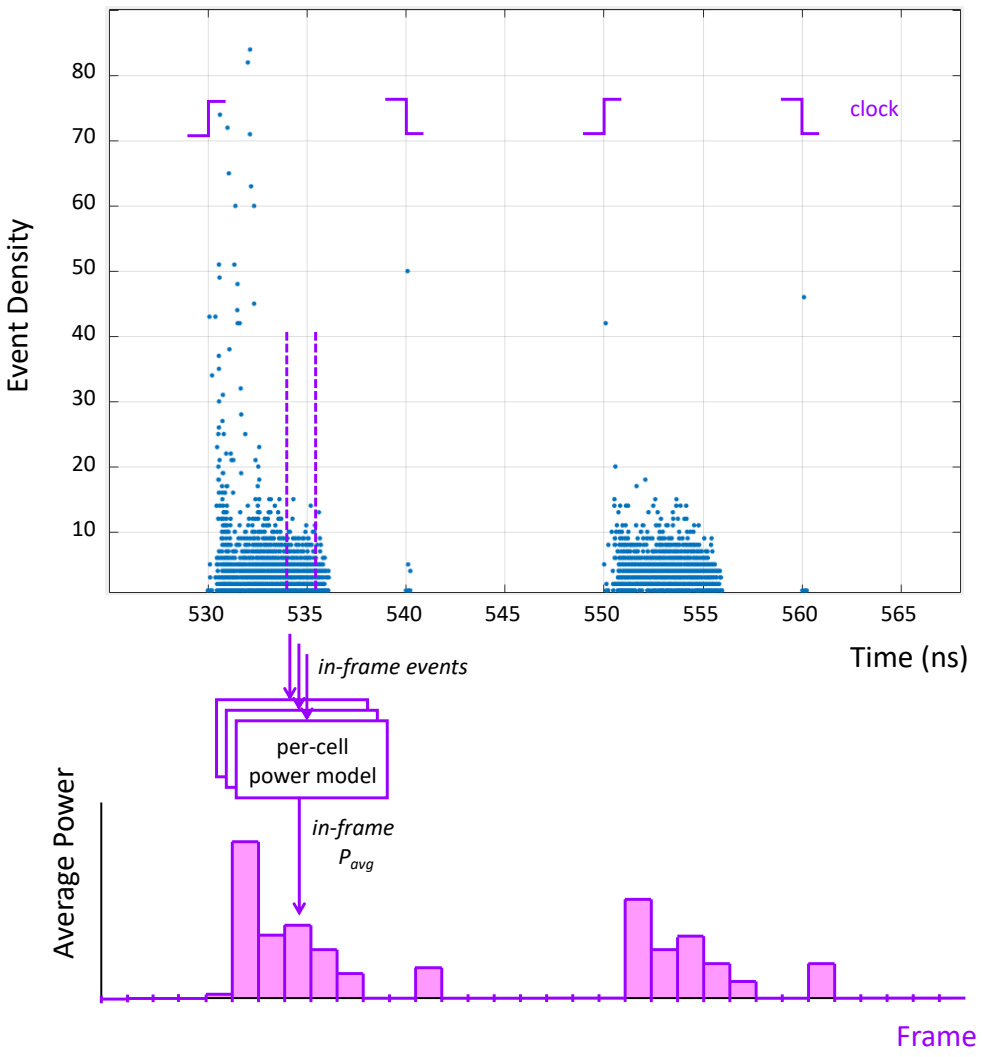


Figure 8.3: Event Density for two cycles of an 9,640-cell hardware AES design

frames. Next, for each frame, the gate-level power simulator computes the average power of a gate as a combination of the switching power, the internal power, and the leakage power. The gate switching power depends on the per-frame output toggle rate and the capacitive loading at the gate output. The gate internal power depends on the per-frame, per-pin input toggle rate. The gate leakage power depends on the per-frame state-dependent leakage power. All these factors are scaled by the technology selection and by the gate type. The sum of all these factors for all active gates within the frame determines the average frame power.

The frame size is an important selection parameter of the gate-level power simulation. The frame size determines the smallest time interval analyzed by the ACA flow for side-channel leakage. Each single cell in a design typically switches over a time interval much smaller than the clock cycle, and much smaller than the frame size. A single frame will thus contain the leakage of many different cells. A smaller frame size helps in detecting power variations caused by a specific cell. Figure 8.3 illustrates this point in further detail. The top half of the figure is an event density plot for two cycles from a hardware AES design containing 9,640 cells. The cycle time of this design is 10ns, and after each up-going clock edge, there is about 5ns of activity as the combinational cells settle to the new register output. In this simulation, there are about 10,000 events in the first clock cycle, and 6,500 events in the second clock cycle. To estimate the power, we select 8 frames per clock cycle (as an example). The power simulator will then compute the power per frame by analyzing the events within that frame. The power trace will thus contain 8 points per clock cycle, and side-channel leakage must be detected by power variations on any of these points. Clearly, with a smaller frame size, fewer events will contribute to that frame, so that it becomes easier to identify which events (and which cells) are a root cause of side-channel leakage.

On the other hand, a *large* frame size is beneficial because it improves simulation time. We demonstrate this effect with the following experiment. A design containing 1, 4 resp. 16 AES S-boxes is driven by a set of counters that each count from 0 to 255. We determine the power estimation cost for a 256-cycle test-bench under various frame sizes. The total simulated time remains 256 cycles for each case, and therefore a smaller frame size requires more frames to be computed. Table 8.1 demonstrates that the power simulation cost significantly depends not only on the design size, but also on the number of frames.

In a practical assessment of hardware, we over-sample the clock cycle at least several times, in order to run the analysis on sub-cycle events. However, with long-running, complex simulations, we have already successfully down-sampled the frame size to as much as 80 clock

Table 8.1: Normalized complexity of power estimation time for three different design sizes and four different frame widths.

Samples/cycle	# Frames	1 SBOX	4 SBOX	16 SBOX
1/256	1	1.0	1.72	4.54
1	256	6.9	23.5	93.8
2	512	11.1	37.1	149.4
4	1000	13.17	45.6	184.8

Skywater 130nm power simulation with Cadence Joules

cycles per frame, while still being able to demonstrate side-channel leaks [97]. Successfully finding side-channel leaks in an under-sampled power trace is due to the averaged sampling technique employed by power simulation tools [99].

In Phase 2 and Phase 3 of the ACA flow, we aim to identify the cells' individual contribution to side-channel leakage. The challenge is to complete this task using only the global power traces. Indeed, the generation of per-cell power traces has quadratic complexity (namely $design\_size \times frame\_count$) and is therefore not scalable to large applications. We solve this with a two-step approach. First, using the global power traces, we identify the *leakage time interval*, the time window within which a design leaks information. Next, within the leakage time interval, we use the activity traces to identify the contribution of an individual cell to a design-level leak. Therefore, Phase 2 and Phase 3 of the ACA flow are defined as *Leakage Time Interval Selection* and *Leakage Impact Factor Computation* respectively. Leakage Time Interval Selection uses design-global power traces, while Leakage Impact Factor computation uses per-cell event traces.

To test individual cells for side-channel leakage, we can use two different testing scenarios. In the following sections, we clarify each testing scenario separately.

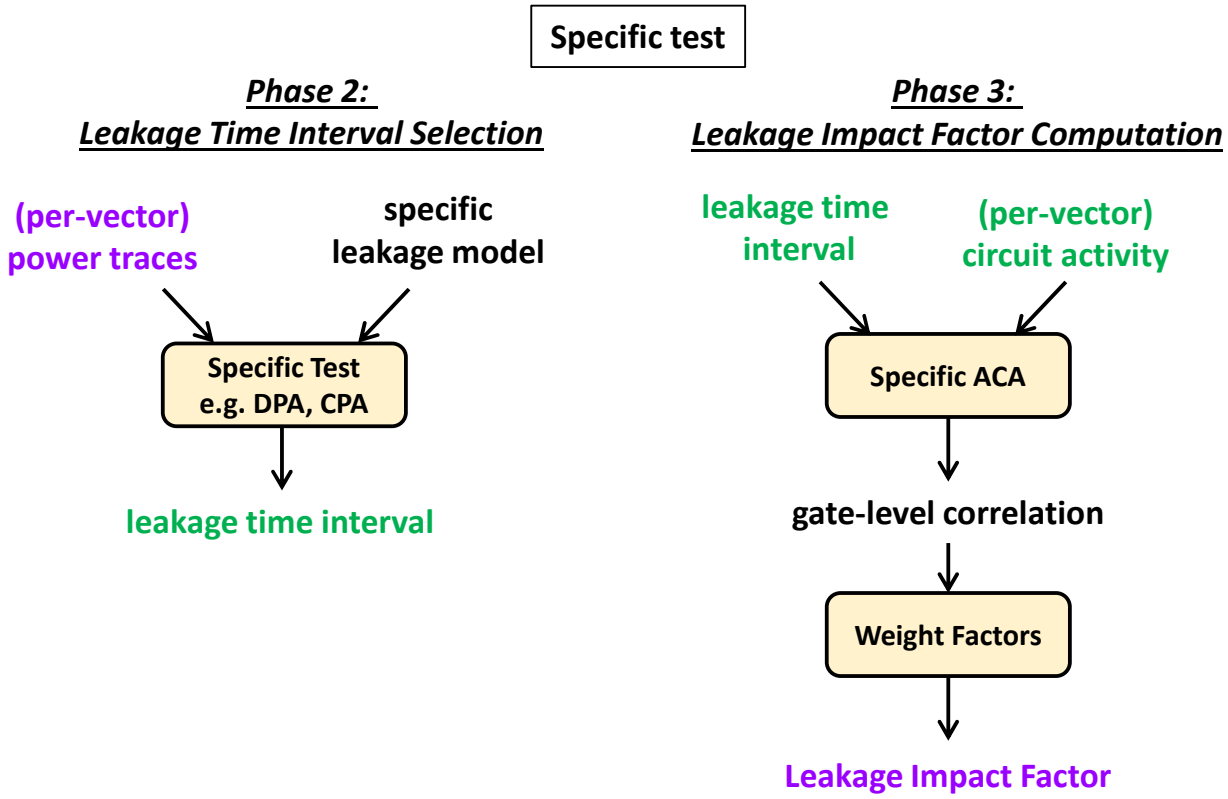


Figure 8.4: Phase 2 (Leakage Time Interval) and Phase 3 (Leakage Impact Factor) computation for specific ACA

8.3.2 ACA for Specific Testing

Figure 8.4 describes the two steps to apply ACA using a specific test. The specific test requires the definition of a leakage model, similar to the leakage model used in Differential Power Analysis or Correlation Power Analysis. The leakage model is correlated with the simulated power consumption to identify a window of high correlation as the Leakage Time Interval (LTI), the time window during which a design leaks. Next, each individual cell is ranked by computing its correlation to the leakage model within the LTI.

Table 8.2: Pearson Correlation Threshold Levels as a function of test vectors m and confidence

Confidence Level	**m=600**	**m=1000**	**m=2000**
99%	± 0.105	± 0.081	± 0.058
95%	± 0.080	± 0.062	± 0.044
90%	± 0.067	± 0.052	± 0.037

Leakage Time Interval for Specific Testing

Given a cipher which computes an internal result $V_k = f(K, C_k)$ with a partial key K and a controlled input C_k from k-th test vector, a possible leakage model $L(V_k)$ is the Hamming Weight $HW(V_k)$. Alternately, for an internal tuple $(V1_k, V2_k) = f(K, C_k)$, the leakage model can be expressed as the Hamming Distance $HD(V1_k, V2_k)$. We then compute the correlation between the leakage model $L(V_k)$ or $L(V1_k, V2_k)$ and the simulated power $P_k[n]$ for each frame n of the simulated power traces:

$$\rho[n] = \frac{cov(L(V_k), P_k[n])}{\sigma_L \sigma_P} \tag{8.1}$$

The LTI consists of the set of frames for which the absolute correlation is above a given threshold.

$$LTI = \{n\} : abs(\rho[n]) > \rho_{threshold} \tag{8.2}$$

The choice of the threshold is a sensitivity parameter that must be chosen such that the LTI covers design activity that contains a likely correlation peak. Table 8.2 shows several examples of threshold levels as a function of the number of test vectors m and the confidence level. As expected, a requirement for higher confidence or the use of fewer traces will increase the confidence interval, which means that stronger correlation peaks must be identified before the frame is flagged as leaky and added to the LTI.

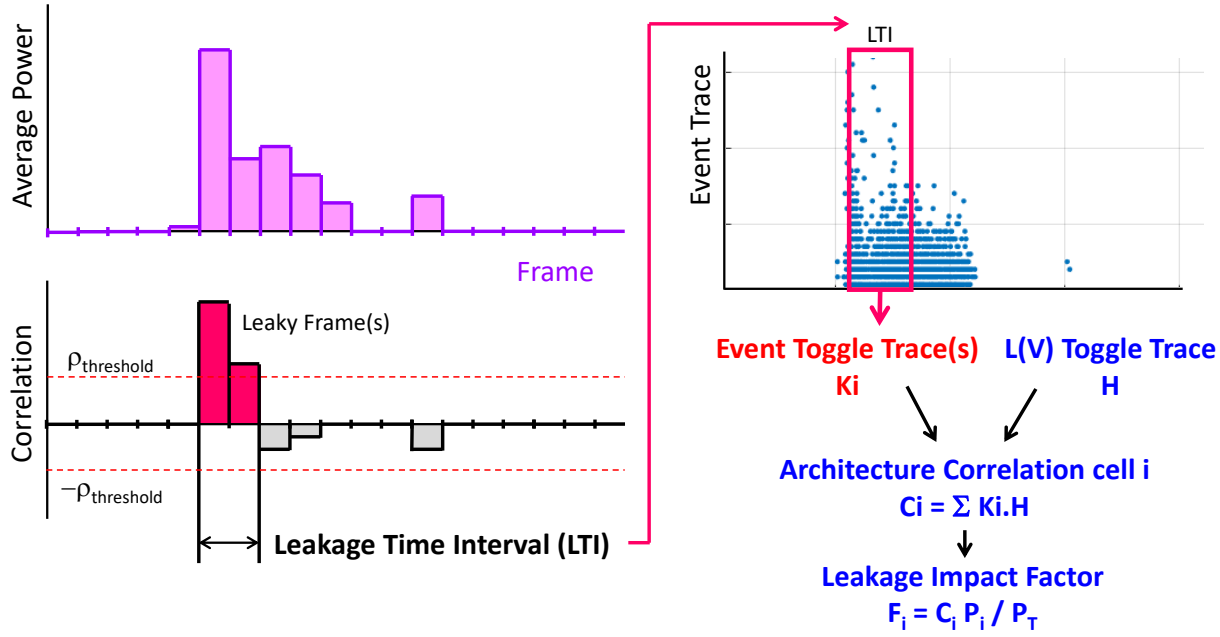


Figure 8.5: (left) Leakage Time Interval Detection (right) Architecture Correlation Analysis

Leakage Impact Factor for Specific Testing

Within the LTI, we compute the contribution of each individual cell to the leakage. This contribution is quantified in the Leakage Impact Factor (LIF), which is a dimensionless number that expresses the relative amount of side-channel leakage from a cell.

The LIF of a cell is computed as the correlation of cell output activity and leakage model activity. The LTI is a set of frames that are considered *leaky* (Figure 8.5, left). To investigate the cell leakage, the LTI is superimposed over the activity trace. For each net (or each net-driving cell), we then compute a toggle trace as follows. When a given net switches during the LTI, then that transition is counted as a +1 toggle. When a given net does not include such a transition in the LTI, then that is counted as a -1 toggle. Hence, the activity of each net i under test vector j is converted to a bi-valued signal K_{ij} with values {-1, +1}. To compute the architecture correlation C_i of net i, K_{ij} is multiplied with the toggle trace H_j of the leakage model $L(V)$ (Figure 8.5, right).

$$C_i = \sum_{stimuli} K_{ij}.H_j \tag{8.3}$$

This correlation can be computed for every frame within the LTI. A high value in C_i indicates a strong correlation between the cell activity and the leakage model, and hence a strong indication that the cell contributes side-channel leakage. All cells in the design are ranked according to their C_i from most leaky to least leaky. We also include an additional weighing factor for each C_i, defined as the ratio of the cells' average power consumption P_i during the LTI over the total average power consumption of all gates P_T. This increases the weight of high-drive cells with a high correlation. This leads to the weighed per-cell LIF F_i:

$$F_i = C_i \frac{P_i}{P_T} \tag{8.4}$$

Unlike computing a gate's power for every frame, computing a gate's average power over all frames is relatively quick (Table 8.1). Therefore, the weighing process is scalable. Each LIF factor is bound to a specific cell within a specific frame in the LTI. Hence, for a design with G gates and J leaky frames, the list of LIF factors contains $G \times J$ entries. These entries are sorted by LIF value to determine the overall leakage ranking.

8.3.3 ACA for Non-specific Testing

When a specific test is hard to apply, or when the number of leakage models $L(V)$ becomes too numerous, it may be helpful to apply a more generic non-specific test for leakage. We can run ACA using a non-specific leakage model following a strategy as in Figure 8.6. Similar to TVLA, non-specific ACA requires the definition of two groups of stimuli. These two groups should exhibit some systematic difference in the design behavior. For example, Goodwill *et al.* suggest AES test vectors that are random for group 1, while introducing a specific bias

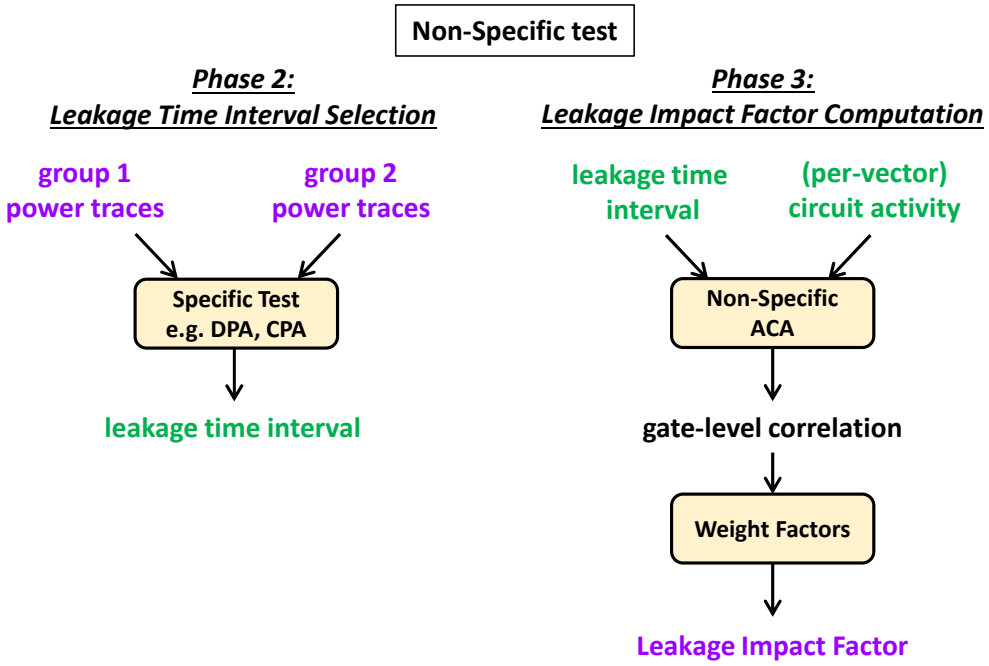


Figure 8.6: Phase 2 (Leakage Time Interval) and Phase 3 (Leakage Impact Factor) computation for nonspecific ACA

within a middle-round state for group 2. Using these two vector groups, ACA then follows the same two-step strategy as for specific testing. First, the LTI is computed to bound the leakage in time, and next the LIF per gate is computed. The testing statistic is adjusted to a non-specific test.

Leakage Time Interval for Non-Specific Testing

The test-statistic compares the distribution of power values at a specific frame between two test vector groups. One solution is to use a Welch-t statistic, which tests the difference between the mean values of both groups. Another solution is to measure the correlation of the power value to the group number. We use the leakage model $N = groupid$, with $groupid$ equal to -1 for vectors from group 1, and +1 for vectors from group 2. With this leakage model, we can compute a non-specific test statistic as a correlation value that can be

compared against a threshold value $\rho_{threshold}$. With $P[n]$ the power consumed during frame n, the correlation is given by:

$$\rho[n] = \frac{cov(N, P[n])}{\sigma_N \sigma_P} \tag{8.5}$$

The LTI is then defined using the same method as in Equation 8.2.

Leakage Impact Factor for Non-Specific Testing

Once the LTI is fixed, we proceed with computing the LIF using a similar strategy as for the specific test. The LTI is superimposed over the activity trace of the design. A net transition during the LTI counts as a +1 toggle, and an absence of transition counts as a -1 toggle. We compute the architecture correlation C_i of net i by correlating the toggle trace K_{ij} with the groupid N_j.

$$C_i = \sum_{stimuli} K_{ij}.N_j \tag{8.6}$$

Again, a high value in C_i indicates a strong correlation between the cell activity and the non-specific leakage model, and hence flags the cell C_i as leaky. To find a cell's non-specific leakage impact factor F_i, a weighing factor is introduced as in Equation 8.4.

The advantage of the non-specific test over the specific test is that no high-level leakage model is necessary. For example, we have used non-specific tests based on one or more state bytes at a middle round being 0 or else random. In the experimental results, we will demonstrate the selectivity of both the specific as well as the non-specific ACA method.

8.3.4 Implementation

Our flow is fully realized in commercial tooling along with customized scripting to implement the statistical post-processing. We use Cadence Genus 20.1 for logic synthesis from RTL,

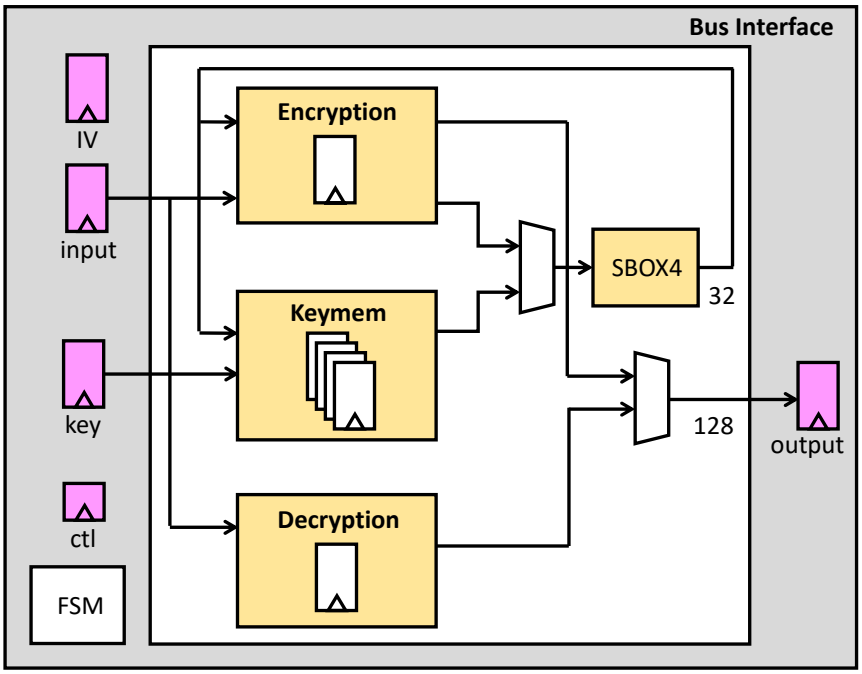


Figure 8.7: Block diagram of the AES encryption/decryption unit

Cadence XCelium 20.09 for functional simulation, and Cadence Joules 10.1 for gate-level power simulation. Computation of ACA LIF and LTI is scripted on top of the Jlsca toolbox[2]. We have used Skywater 130nm standard cells as technology targets during experiments.

8.4 ACA on a Cryptographic Coprocessor

In this section, we describe the application of ACA on an AES encryption/decryption coprocessor. The architecture selected for analysis is typical for a medium-throughput accelerator residing in an embedded SoC. The coprocessor handles encryption and decryption and uses an offline key schedule, which computes the roundkeys once upon loading of the key. A single round takes 5 clock cycles. In the first four cycles, the coprocessor computes 16 Sbox lookups in sets of 4, and in the fifth clock cycle, the remainder of the round is computed. The encryption/decryption core is encapsulated by a bus interface which contains software-

[2] `https://github.com/Riscure/Jlsca`

Table 8.3: Cell type and area for AES coprocessor

Type	Cell Count	Area (%)
Sequential	2,479	51.8
Logic	7,161	48.2
Total	9,640	100.0

accessible registers and a controller. The bus interface handles various modes of operation for the coprocessor. We synthesized this coprocessor for SkyWater 130nm standard cell technology and a 50MHz clock. Table 8.3 reports the type and number of cells used in the design, as well as their relative area.

8.4.1 Architecture Correlation Analysis

Stimuli selection plays an important role in ACA, as it enables a designer to choose which part of a design will be exercised. In this analysis, we will focus on Architecture Correlation Analysis for a single key byte in the AES coprocessor. The objective of the ACA is to determine which cells, among the 9,640 cells in the design, contribute to side-channel leakage of this key byte. We explicitly differentiate this objective (finding leaky gates) from a more traditional side-channel analysis of the hardware. There is no doubt that there is side-channel leakage in this design. However, the object of this experiment is to find *what cells are most responsible for this leakage*?

Results

We performed ACA as follows. We selected a set of 1024 vectors under a random plaintext and a constant key. Next, we ran a gate-level simulation and a gate-level power simulation for ACA. Figure 8.8 shows the average power trace of the first round at 64 frames per clock cycle, as well as the (sign-flipped) standard deviation. The clarity of this power trace illustrates the strength of noiseless simulation and outlines each clock cycle of operation as

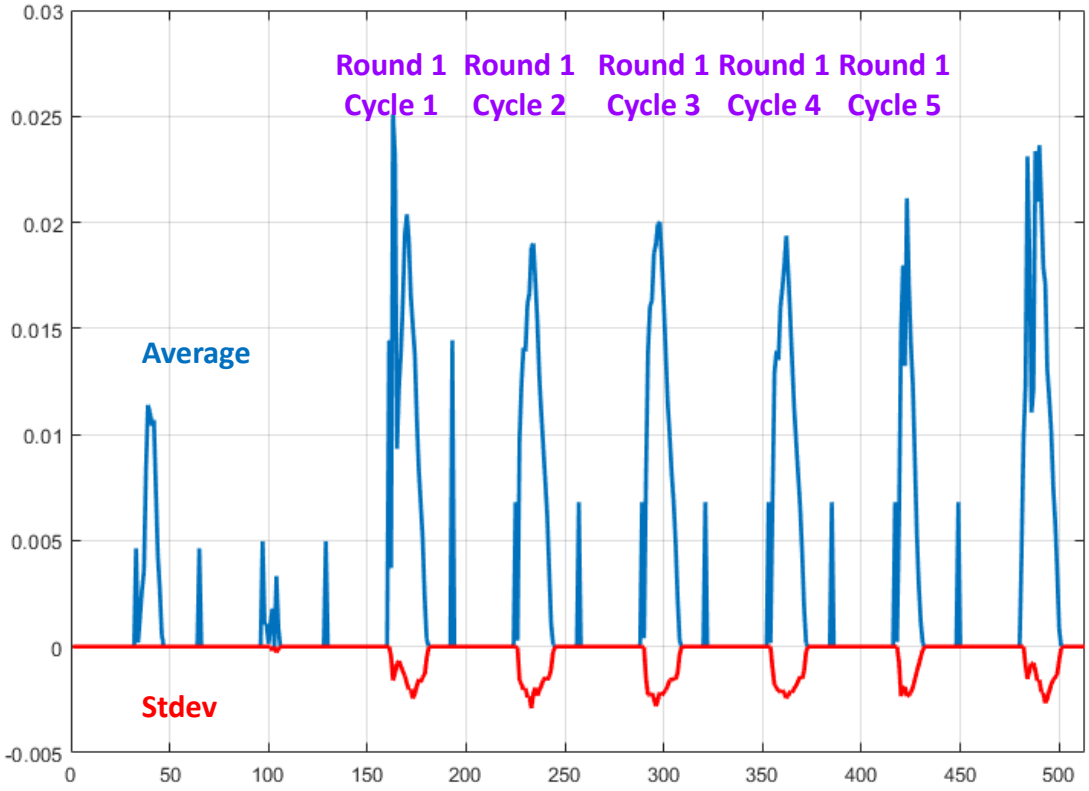


Figure 8.8: Average Power Trace and Standard Deviation of AES Coprocessor in first round

well as the location of power variations. In this simulation, there are 512 frames in each power trace.

We next ran ACA using a specific leakage model on the Hamming Weight of the SBOX output of the first key byte. We selected a specific leakage model on the Hamming Weight of the SBOX output of the first state byte. We identify the LTI with a $\rho_{threshold}$ of 0.2, which flags 230 frames out of the 512 frames as containing potentially leaky samples. By correlating the transitions by cells within LTI with the specific leakage models, 412 cells are then flagged as leaky. This group represents 4.3 % of the total number of 9,640 cells. We will analyze the relation of these 412 cells to the overall AES coprocessor in Section 8.4.2.

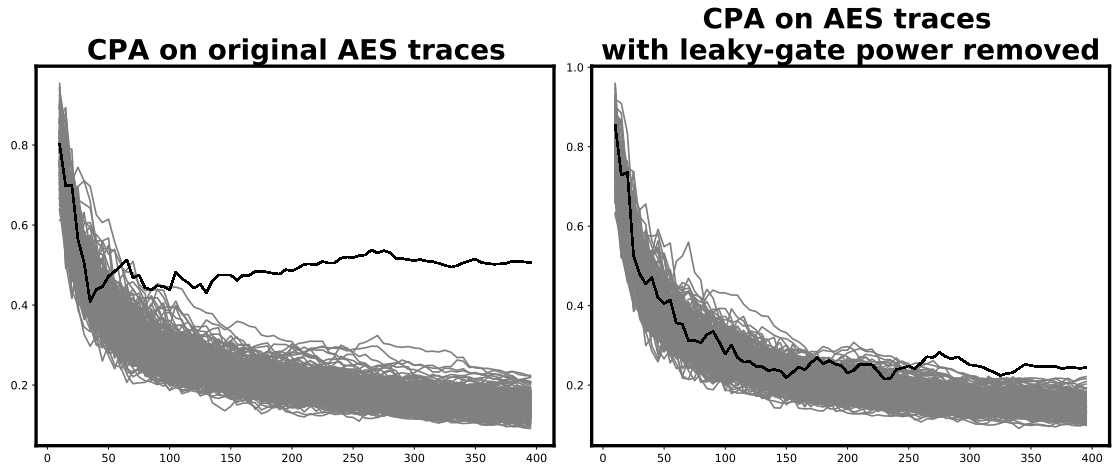


Figure 8.9: (left) CPA on AES Coprocessor traces reveals a correlation peak at about 75 traces (right) CPA on modified AES Coprocessor traces significantly delays correlation peak disclosure to at least 250 traces.

Result Verification

The assertion made by ACA is that the side-channel leakage under the selected leakage model is primarily caused by these 412 cells. To verify the correctness of the selection, we performed the following verification. We re-ran the simulation while collecting individual power traces for *each* cell for all vectors and all frames. The per-cell power traces of the 412 selected leaky cells are then subtracted from the overall power traces to construct a modified set of power traces. Next, we apply a Correlation Power Analysis (CPA) on the original trace set as well as on the modified trace set, with the results summarized in Figure 8.9(left: unmodified set, right: modified set). For the modified set, the number of measurements to disclosure increases with a factor of 3.

We emphasize that this CPA experiment only verifies the selection of leaky cells. ACA is not a countermeasure but a detection tool. One cannot remove an arbitrary cell from a netlist without substituting it with an equivalent cell with identical functionality. However, ACA is useful in conjunction with countermeasure tools that protect individual cells or subsets of cells, such as Karna [61] or STELLAR [46]. A second observation is that a power

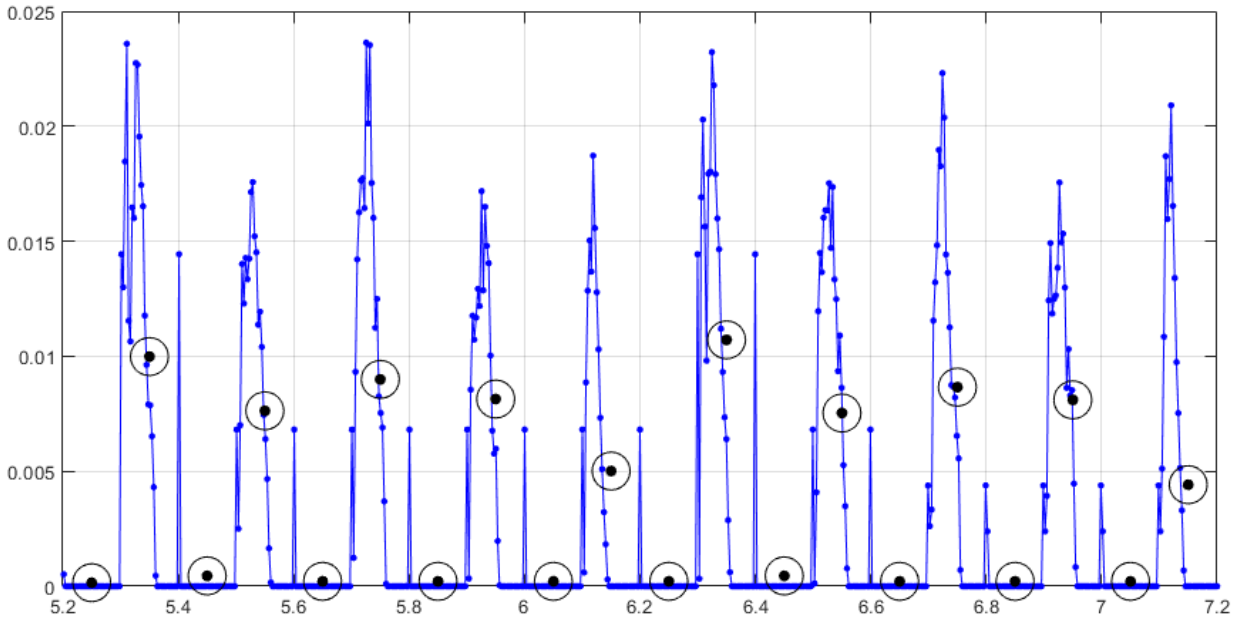


Figure 8.10: Comparison of a power trace at 64 frames per cycle to a power trace at 2 frames per cycle. At lower frame counts, the power sample converges to average power, and the frame size increases.

simulation that collects an individual power trace for every gate is extremely complex both in disk space and in time. We found the overhead of single-gate power tracing (compared to standard ACA) to be around 4 orders of magnitude in disk storage and one order of magnitude in simulation time, and worsening with design size. The high cost of per-gate power tracing highlights the strength of ACA to use per-gate activity traces, which are a byproduct of functional verification.

Impact of Frame Size

We also performed an ACA analysis at a frame size of 2 frames per cycle rather than 64 frames per cycle. Figure 8.10 shows the effect on the power trace. At wide frame size, the power converges to the average of the smaller frame size. In our experiments, we found that a wide frame size is less precise to pick out leaky gates. For the same trace set as the previous experiment, the 2-frame-per-cycle version flags only 122 cells (as opposed to 412

Table 8.4: Leaky Gate Identification for AES Coprocessor

Module	File	# Cells	Sequential
Top-level	`aes_comp_core.v`	5	0
Decryption	`aes_comp_decipher_block.v`	29	0
Encryption	`aes_comp_encipher_block.v`	204	26
Keymem	`aes_comp_key_mem.v`	1	0
SBOX	`comp_sbox.v`	130	0
Bus Interface	`picoaes.v`	22	10

cells) as leaky.

However, the use of a wider frame size may still have advantages. The 122 cells that are found at a wider frame size are a subset of the 412 cells found at a smaller frame size, with an exception of a single cell. Furthermore, there is a significant performance gain in power simulation time at wider frame sizes (Section 8.6). We can thus think of power simulations at wide frame sizes as a quick assessment to determine the LTI and to scan the overall properties of side-channel leakage in the design.

8.4.2 Leaky Gate analysis

We analyzed the type and nature of the 412 cells that are being flagged as leaky by ACA, under the specific leakage model of the SBOX output. The direct analysis of the gate-level netlist is cumbersome because the synthesized netlist is flattened, and because most gates have non-descriptive names such as `g136941`. However, it is possible to direct the synthesis tool to keep track of the originating line of RTL code that results in a specific gate. This way, we found that the 412 cells come from 47 unique sites in the RTL code. This allows the user to identify the RTL source code location of the leakage, and Table 8.4 summarizes the identified gates by RTL source file. 21 leaky cells are not identified by their RTL origin and are not listed in the table.

The list of cells in Table 8.4 is intriguing. ACA is able to identify non-trivial leakage,

Table 8.5: Leaky Gate Identification using non-specific round-6 state bias

Module	File	# Cells	Sequential
Top-level	`aes_comp_core.v`	79	0
Decryption	`aes_comp_decipher_block.v`	351	0
Encryption	`aes_comp_encipher_block.v`	1172	128
Keymem	`aes_comp_key_mem.v`	0	0
SBOX	`comp_sbox.v`	870	0
Bus Interface	`picoaes.v`	277	0

often occurring as a result of the integration of cryptographic functions. The following example illustrates this point. In the Bus interface, the IV register is flagged as a source of leakage, which is unexpected. However, upon inspection of the code, it can be shown that the IV register senses every output value of the encryption module. Furthermore, due to the sequential nature of the computation, the encryption module reflects intermediate round values, *including* each individual SBOX output. The results in SBOX-related leakage appear at the IV register. The identification of individual RTL files and line numbers as leaky, based on a gate-level simulation, is an important debugging tool in the hands of the designer.

8.4.3 Non-specific ACA

We illustrate how ACA identifies leaky cells with a non-specific leakage model. In the following example, we create a non-specific test on the state variable in round 6 of the encryption. We use two sets of test vectors, and both contain random plaintext and key values. However, the second group contains specially selected (plaintext, key) pairs that create an all-zero round-6 state variable. Such pairs are easy to create: select a random key, and decrypt an all-zero state starting at round 4 of the decryption. The resulting plaintext is the sought starting value.

Using non-specific ACA we can identify the gates that are most affected by this bias, and thus the gates that are responsible for side-channel leakage. Figure 8.11 illustrates the LTI

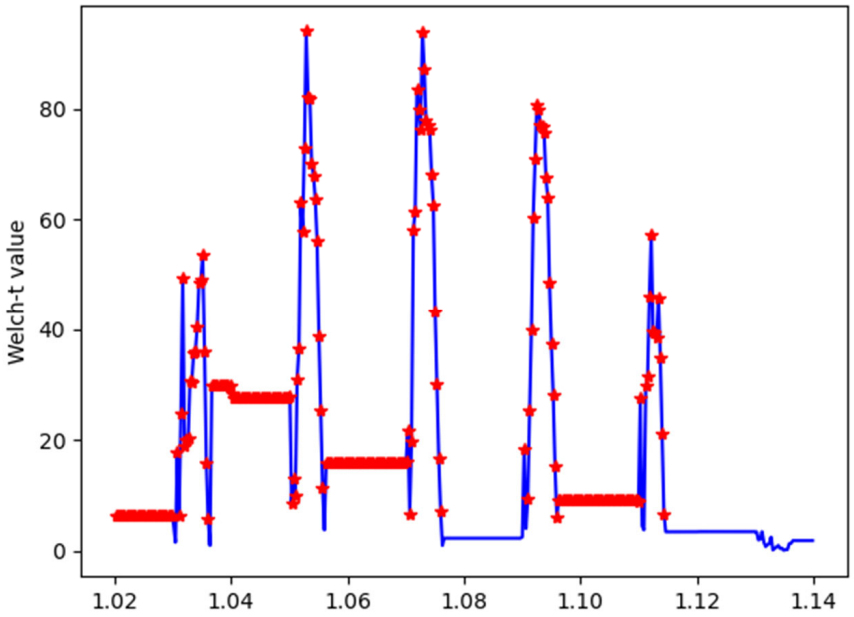


Figure 8.11: Leaky frames in round 6 for a non-specific test on all state bits concurrently.

on round 6, where the bias occurs. The majority of the frames (251 out of 385 in the trace) are flagged as part of LTI. Furthermore, after ranking the cells, we identify 2,812 unique cells as correlated with the round-6 state bias. This is much more than the 412 cells selected using a specific leakage model on the first key byte. However, this result is not unexpected: zero-forcing an entire state word (128 bits) where the expected value would be random is a very significant bias, which has an impact throughout the datapath. Table 8.5 shows the distribution of the 2,812 cells over the design.

Among the flagged leaky cells 2,498 gates were traced back to their RTL design files. Figure 8.12 illustrates the rank of flagged leaky gates from each design file with rank=1 belonging to the leakiest gate in the design.

We caution that a strongly biased test, such as this all-zero round-6 non-specific test, always results in aggressive leaky cell selection. However, a weaker form of the test is easy to define, for example by biasing only a single state byte of round 6. The non-specific ACA test lets a user evaluate the impact of an arbitrarily chosen bias in the cipher.

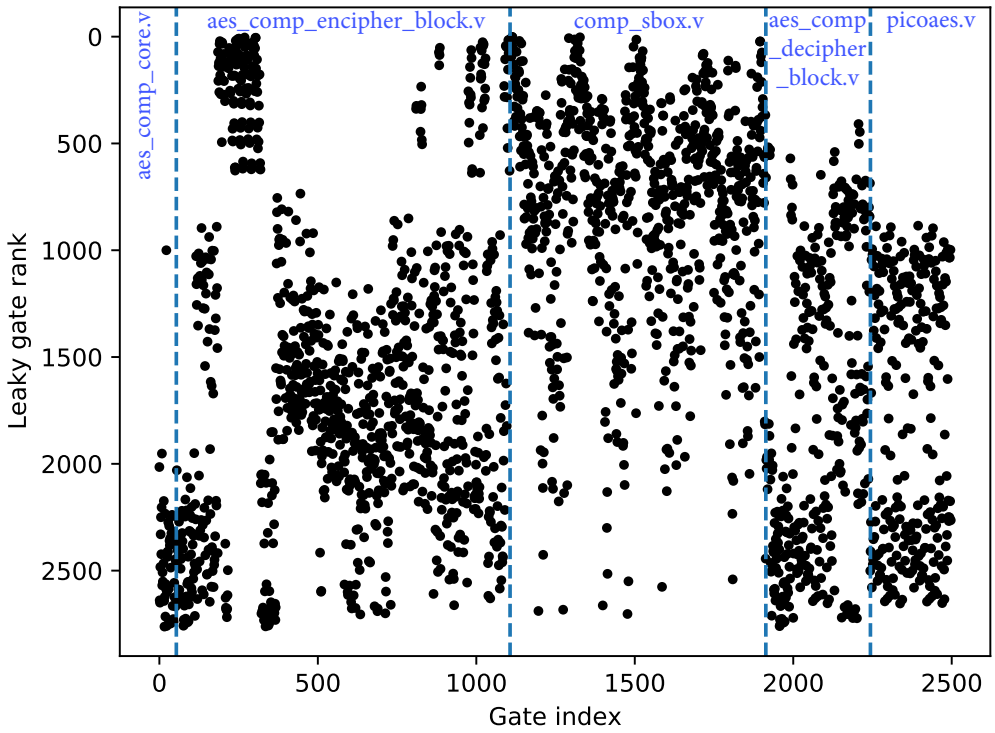


Figure 8.12: Leaky gate ranks identified by non-specific test in ACA on AES coprocessor sectioned into RTL design files.

8.5 ACA on RISC-V based SoC

To investigate the scalability of ACA we also applied the methodology on the SoC shown in Figure 8.13. A 5-stage pipelined RISC-V core (fetch, decode, execute, memory, write-back) integrates a collection of peripherals including the memory-mapped AES coprocessor discussed in Section 8.4. In a typical access sequence, the RISC-V software uploads a key and a block of plaintext to the coprocessor, and then uses the control/status register of the coprocessor to start the encryption and monitor the completion flag. The RISC-V software then retrieves a block of ciphertext. This design is considerably more complicated than the stand-alone AES design, and covers a software and a hardware component. Table 8.6 shows

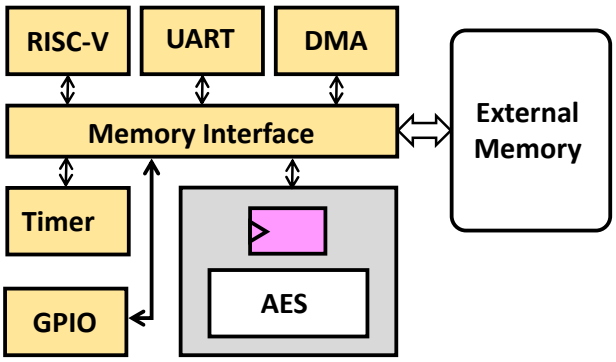


Figure 8.13: Block diagram of the RISC-V based SoC including the AES coprocessor

Table 8.6: Cell type and area for RISC-V based SoC

Type	Cell Count	Area (%)
Sequential	8,091	51.5
Logic	21,484	48.5
Total	29,575	100.0

synthesis results for SkyWater 130nm standard cells at 50MHz clock. The overall design is three times larger than the AES coprocessor by itself.

8.5.1 Architecture Correlation Analysis

In this test, we are investigating the hardware/software interface between the RISC-V software and the AES coprocessor. Therefore, we apply ACA with a specific leakage model using the Hamming Weight on the output of the pre-whitening round. This will enable the monitoring of any interactions between the plaintext and the key on the path from software to the hardware coprocessor. Figure 8.14 shows a portion of the driver software. In a 32-bit architecture, a 128-bit block is loaded using 4 consecutive memory-mapped writes. The driver first loads 4 plaintext words, followed by 4 key words. Next, the coprocessor control register is configured to run a single block encryption. The software then goes into a polling loop waiting for the coprocessor to complete operation, about 50 clock cycles later.

Figure 8.14: RISC-V driver software for AES Coprocessor

```
  li      a4,8
  lw      a3,0(a4)      ; load plaintxt[0]
  sw      a3,4(a5)      ; STALL
  lw      a3,4(a4)      ; load plaintxt[1]
  sw      a3,8(a5)      ; STALL
  lw      a3,8(a4)      ; load plaintxt[2]
  sw      a3,12(a5)     ; STALL
  lw      a4,12(a4)     ; load plaintxt[3]
  sw      a4,16(a5)     ; STALL
  li      a4,24
  lw      a3,0(a4)      ; load key[0]
  sw      a3,20(a5)     ; STALL
  lw      a3,4(a4)      ; load key[1]
  sw      a3,24(a5)     ; STALL
  lw      a3,8(a4)      ; load key[2]
  sw      a3,28(a5)     ; STALL
  lw      a4,12(a4)     ; load key[3]
  sw      a4,32(a5)     ; STALL
  li      a4,6
  sw      a4,0(a5)      ; control
  li      a4,4
  sw      a4,0(a5)      ; start
  li      a3,1
.L121:
  lw      a4,68(a5)
  bne     a4,a3,.L121
```

Results

We selected a set of 1024 vectors under a random plaintext and a constant key. We then ran a gate-level simulation and a gate-level power simulation at 5 frames per clock cycle. Figure 8.15 shows a sample power trace from the simulation. The testbench covers software activity as well as hardware activity. The hardware activity uses considerably more power than software because of the higher parallelism of the hardware coprocessor implementation. However, because of the noiseless simulation, the overall operation is visible with remarkable

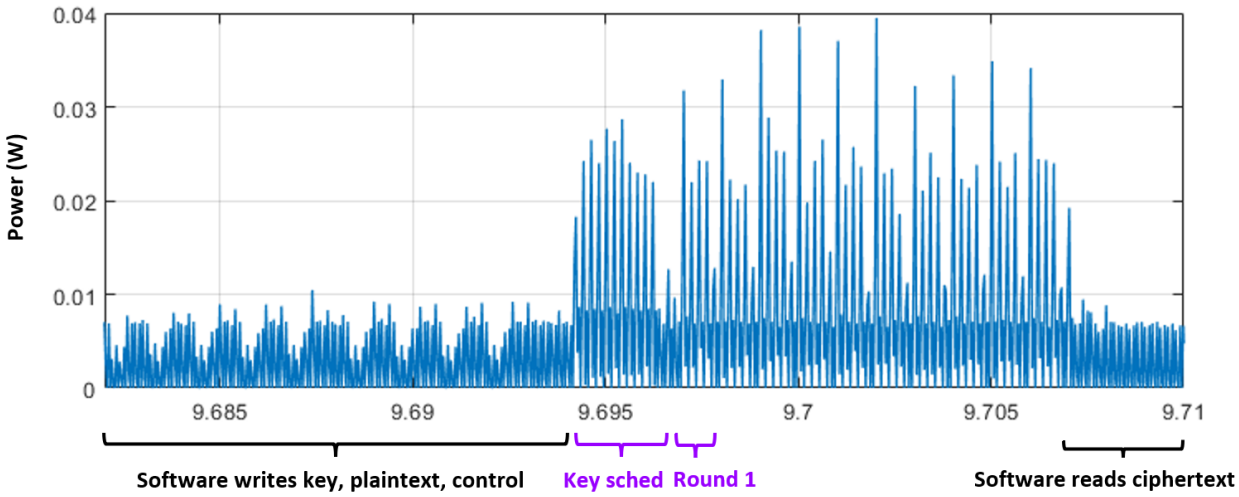


Figure 8.15: Power trace of the RISC-V based SoC

clarity. The trace starts with the software transmitting a key value, followed by a plaintext value. Figure 8.15 shows a series of 8 notches in the power trace which correspond to reduced power consumption. These are caused by pipeline stall operations on the RISC-V processor (Figure 8.14 lines 3, 5, 7, 9, 12, 14, 16, 18). Next, the software triggers the hardware AES execution, which runs the key schedule followed by 10 rounds. Finally, the RISC-V software retrieves the ciphertext.

We next ran ACA using the aforementioned specific leakage model on the Hamming Weight of the input of round 1. We identify the LTI with a $\rho_{threshold}$ of 0.2, which flags 91 frames out of the 710 frames as containing potentially leaky samples. By correlating the transitions by cells within LTI with the specific leakage model, 1,298 cells are then flagged as leaky. This group represents 4.38 % of the total number of 29,575 cells.

8.5.2 Leaky Gate Analysis

Figure 8.16 shows the distribution of leaky frames over the testbench. The leakage model is $HW(key \oplus pt)$. Remarkably, the bulk of the leaky frames occurs during the *software driver* activity, while the key is being loaded into the coprocessor. In addition, there is also leakage

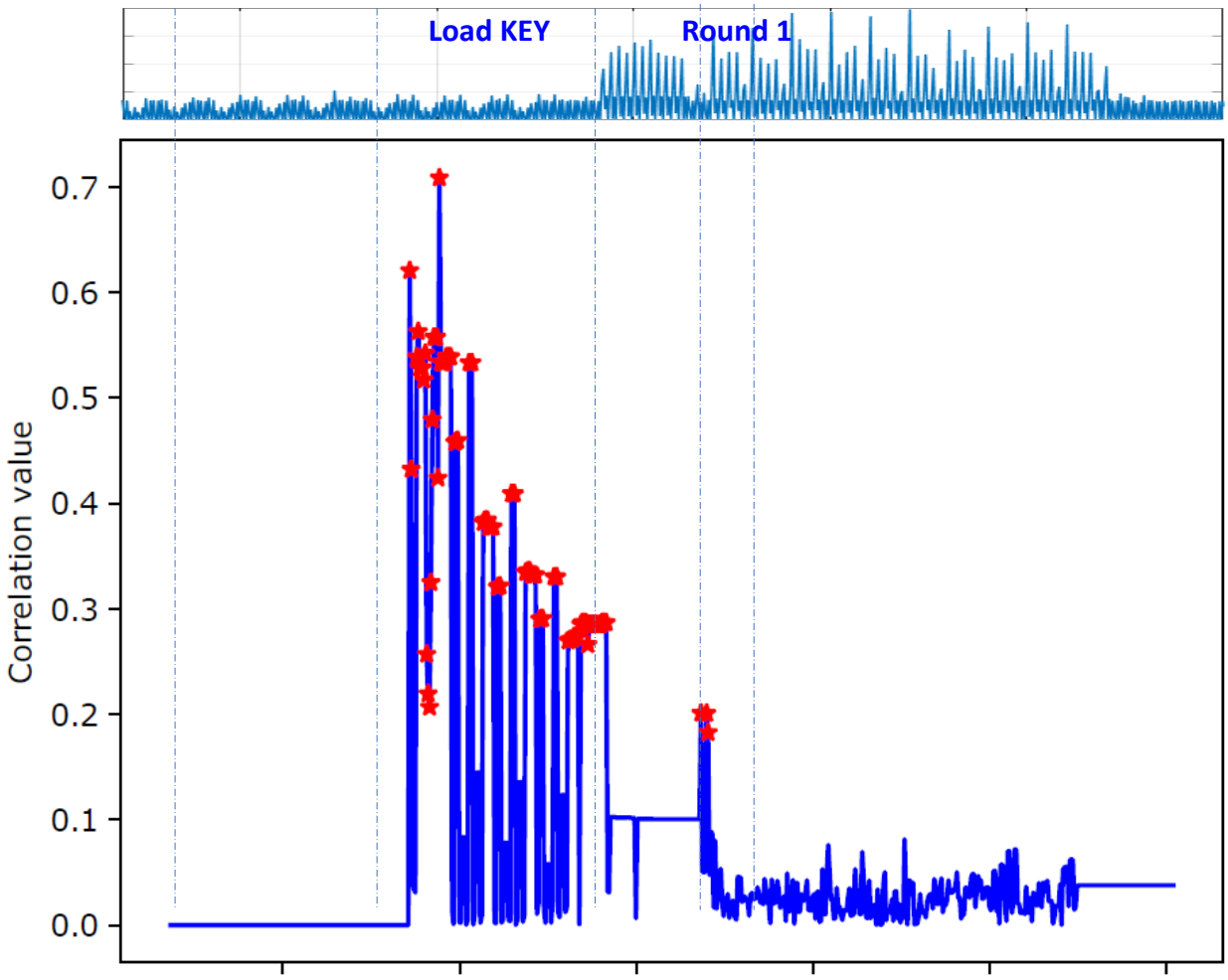


Figure 8.16: Leaky Frame Selection in ACA on RISC-V based SoC

during the first round of the encryption, which is expected.

ACA demonstrates that the RISC-V processor, the memory bus, and the memory-mapped AES coprocessor are all potential contributors to side-channel leakage. The origin of such leakage is caused by the interaction of values over shared storage and interconnect. The leakage occurs in the processor micro-architecture because the driver software writes the key after the plaintext. There is a minor hint of this issue in the software driver itself. The first key byte is loaded in register `a3`, which still contains a portion of plaintext. This leads to distance-based leakage conforming to the leakage model.

Table 8.7: Leaky Gate Identification for RISC-V based SoC

Module	File	# Cells	Seq
AES Coprocessor			
AES top	aes_comp_core.v	2	0
Decryption	aes_comp_decipher_block.v	21	0
Encryption	aes_comp_encipher_block.v	27	14
KeyMem	aes_comp_key_mem.v	1	0
Bus Interface	aes_top.v	130	112
SBOX	comp_sbox.v	108	0
RISC-V			
ALU	ALU.v	332	0
Control	control_unit.v	2	0
Control	controller.sv	10	0
Memory	memory_arbiter.v	11	0
Memory	memory_interface.v	3	0
Pipeline	pipeline_register.v	66	31
Regfile	regFile.v	248	248
Peripherals			
GPIO	gpio_top.v	3	0
DMA Bus Control	s_axi_controller.sv	3	0
UART	simpleuart.v	3	0
DMA	transposer.sv	3	0
DMA	ca_prng.v	4	0
DMA	dma_top.v	6	0
DMA	fifo_dma.sv	3	0
DMA	fifo.v	2	0
DMA	tDMA.sv	2	0

Table 8.7 shows the distribution of the 1,298 leaky cells over the design. There are indeed a large number of leaky gates located within the RISC-V processor. We analyze two examples below.

The highest-ranked leaky gate (with a correlation of 0.463) in the SoC is the pipeline register of the memory stage which, in its data-path components, transfers the contents of the second source register and the result of the ALU from the execute to the memory stage. Even for instructions that do not need ALU operation, the ALU result is written with the

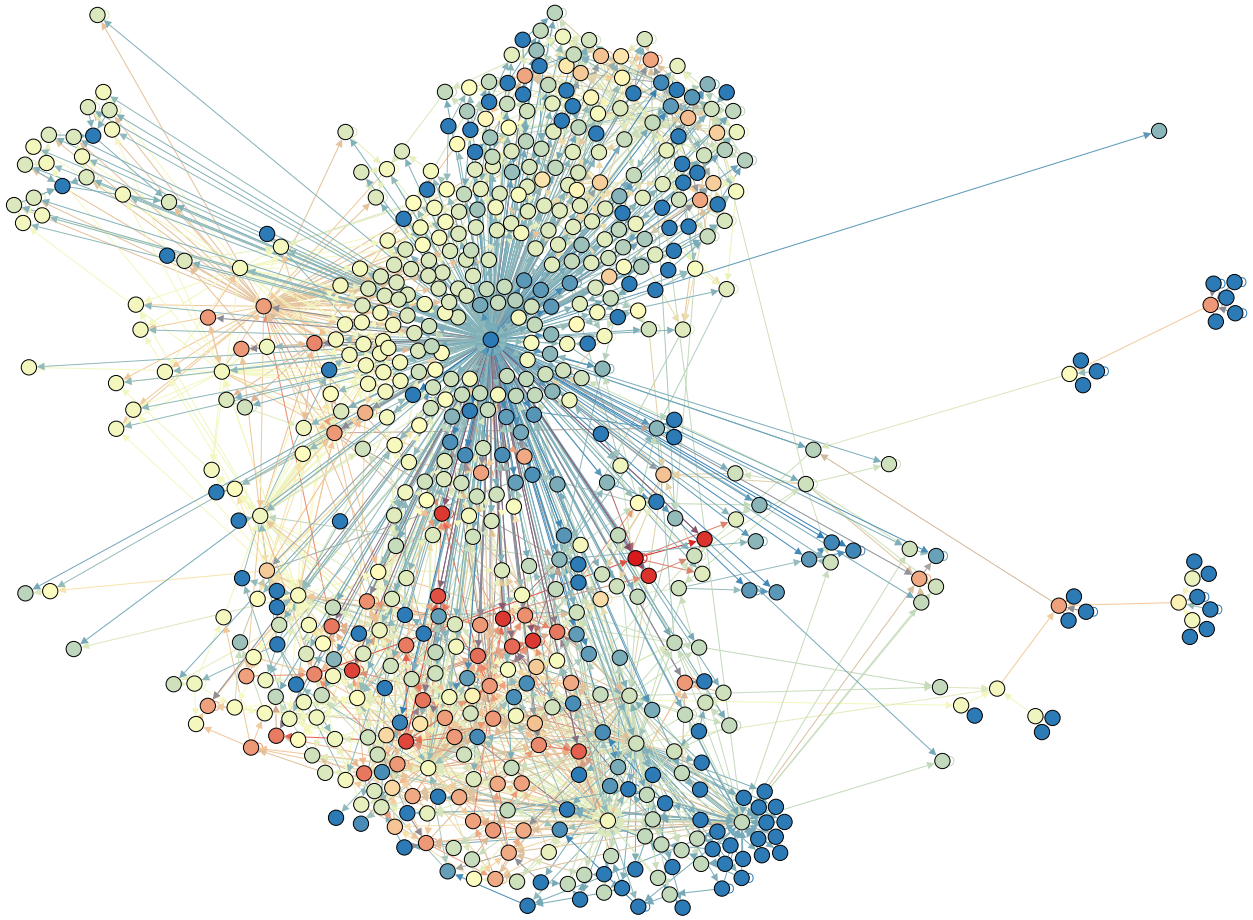


Figure 8.17: Gate-level netlist graph of the RISC-V SoC color coded with leaky gates' correlation. Warmer colors correspond to higher correlations.

addition of the two source operands. Therefore the same transitional leakage discussed for `a3` register can occur for the ALU result register.

The leakage from the peripherals is caused by the transmission of key and plaintext on the memory interface. At the connection point of each peripheral module to the memory interface there are multiplexers to decide whether the transmitted data should be admitted to the current peripheral. Such interconnect logic can manifest Hamming weight leakage of the plaintext and key.

Additionally, we form the gate-level netlist graph (introduced in Section 9.3.2) of the SoC under study. As shown in Figure 8.17, the graph is color coded by the leaky gates' correlation

Table 8.8: ACA Performance for various steps in the flow. Performance numbers in user seconds* for 1024 Vectors.

	AES Coprocessor	**RISC-V based SoC**
Gate-level Synthesis	392	1,201
Simulation	2,436	6,996
Power Estimation		
64 frames/cycle	7,862	
2 frames/cycle	1,557	
5 frames/cycle		31,201
Correlation Analysis	<60	<60

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

with the warmer colors demonstrating higher correlation. The correlations are calculated by ACA while running the aforementioned software driver for the AES coprocessor shown in Figure 8.14. The graph is further reduced by merging the neighboring nodes that have the same correlation value. The final graph has 711 nodes and 2310 edges. The flow of colors in this graph confirms that the ranks of the gates change gradually among the adjacent gates. In other words, sudden jumps in the colors of the connected nodes are rarely encountered.

These examples demonstrate that ACA is a powerful debugging tool, as it can highlight side-channel leakage of the gate-level implementation at the RTL level.

8.6 ACA Performance Considerations

ACA adds a new design step to the overall design flow, and thus the cost of running ACA in comparison to other tools in the design flow must be considered. Table 8.8 summarizes the runtime performance of ACA analysis. There are three major components that consume the bulk of the execution time: logic synthesis, functional gate-level simulation, and gate-level power simulation. The ACA correlation component is minor and typically takes less than a minute to complete. Overall, we observe that gate-level power simulation is a dominant

factor that is more complex than gate-level synthesis and gate-level simulation. The overall runtime is strongly affected by the design size and the total number of frames per trace. On the plus side, the power simulation step is embarrassingly parallel. Each test vector can be run independently from the other. In our experiments, we did not use any parallel execution.

8.7 Conclusions

Gate-level leakage assessment is a tool that supports a designer to identify leaky gates in a pre-silicon design context. Our methodology relies on industry-standard tools including logic synthesis, gate-level simulation, and gate-level power estimation, together with scripting on the intermediate results. Architecture Correlation Analysis, the underlying detection technique to support gate-level leakage assessment, can serve as a verification technique as well as as a basis for countermeasure design. In particular, by moving the leaky gates flagged by ACA into a separate power domain, a low-cost countermeasure may be enabled that requires only selective replacement of cells in a design.

Chapter 9

RootCanal

In this chapter, we introduce a technique to find the cause of an observed power side-channel leakage at gate-level in both software and hardware. This work will appear in IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES) 2022 [104].

9.1 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. 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 9.1 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

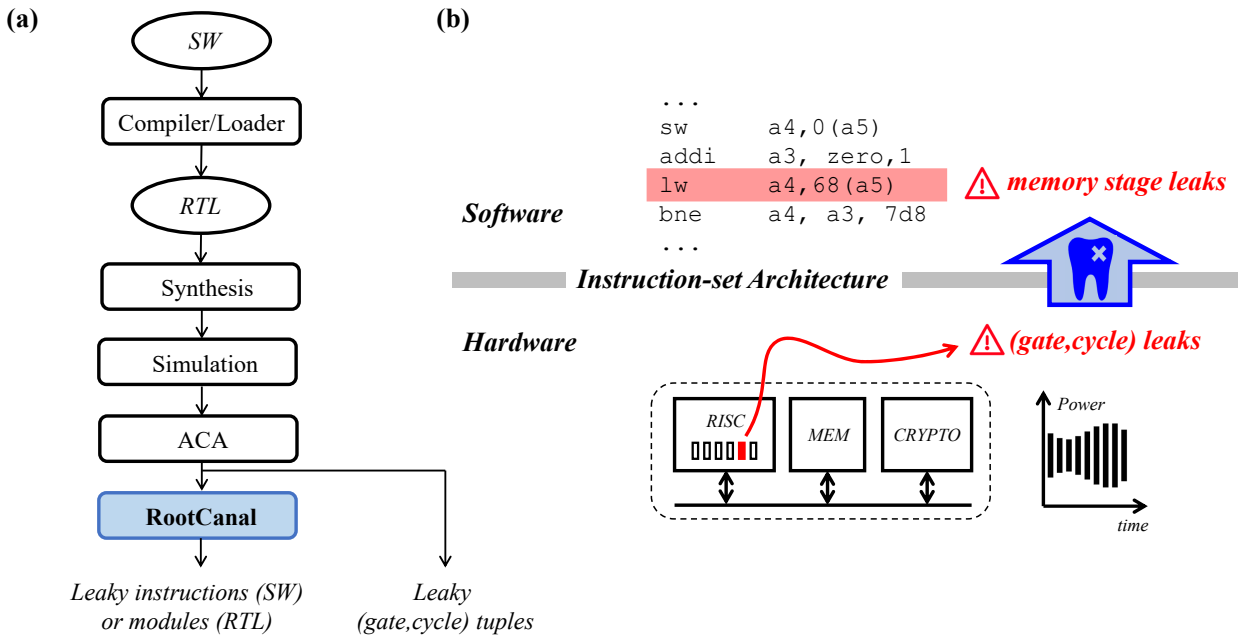


Figure 9.1: RootCanal is a pre-silicon side-channel leakage assessment technique to back-annotate leaky (gate,cycle) tuples in an SoC design to high-level software or hardware information. (a) RootCanal design flow integration (b) Example application.

the 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 [9].

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

[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.

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 [40]. 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 [43]. 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 [59], Architecture Correlation Analysis [168], and KARNA [149]. 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 [31, 16, 108]. 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.* ap-

plied formal verification of masked hardware implementations to analyze micro-architectures that execute masked software implementations [77, 78]. 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 [123, 141, 120, 20]. 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 [72]. 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.

9.2 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) [168], 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 [118]. 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 [42].

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 [106]. 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.

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 [106], 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 9.1 shows the stimuli used for the experimental work of the paper. A *Value Leakage* test evaluates if

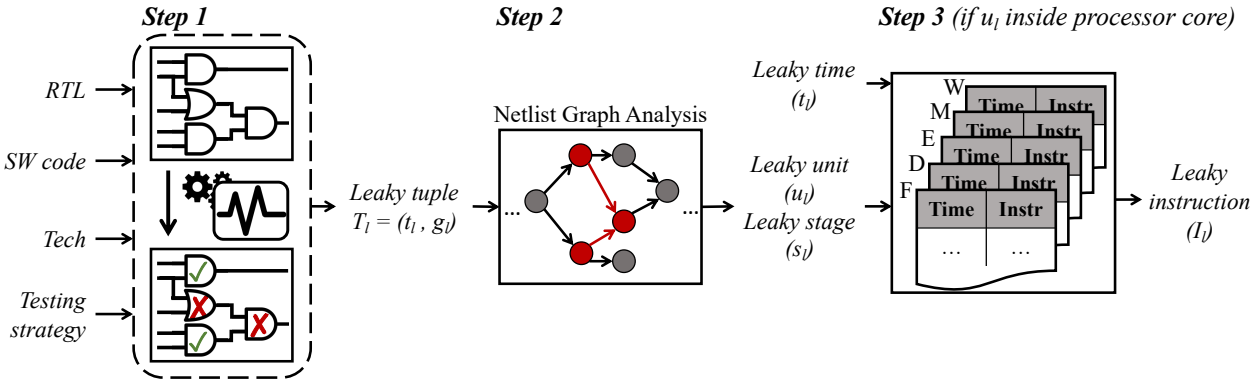


Figure 9.2: Overall RootCanal flow

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 [137].

Table 9.1: 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

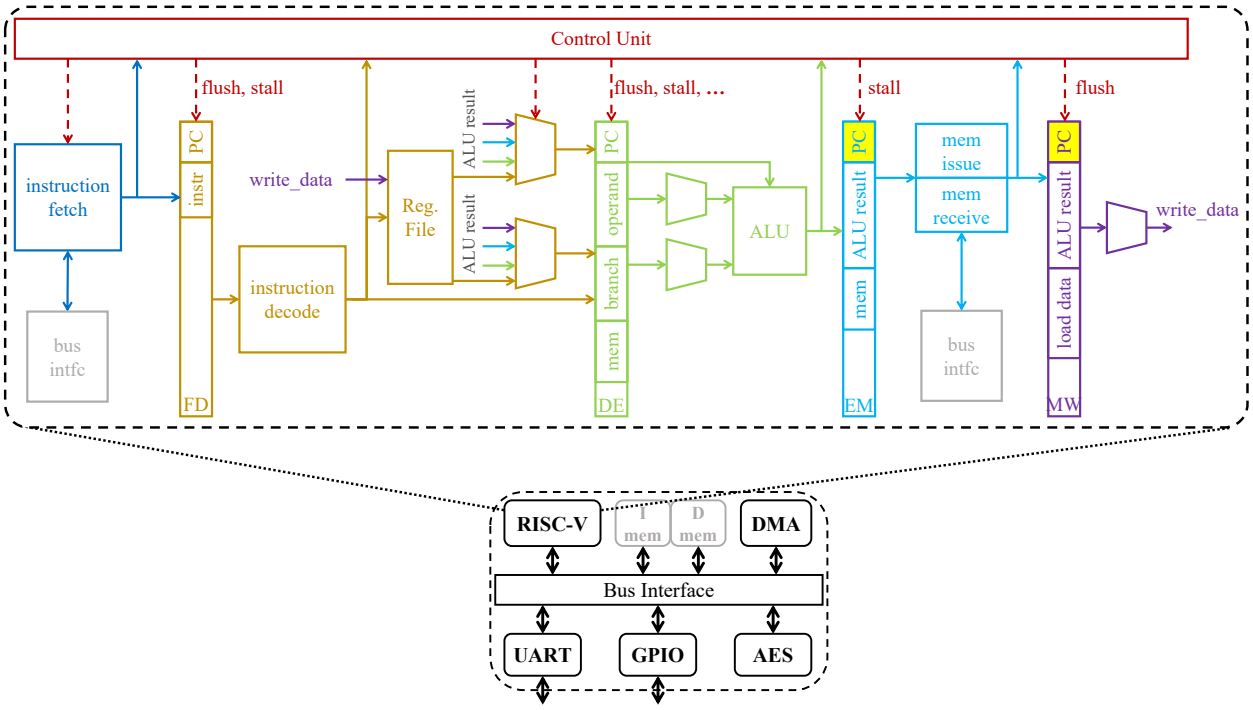


Figure 9.3: Block diagram of RISCV-SoC and its five-stage RISC-V processor. Resources from different pipeline stages are shown in different colors in the processor core. The gray blocks in the SoC (instruction and data memories) are modeled in the testbench (not synthesized).

9.3 Methodology

RootCanal finds the source (in hardware and software) of power side-channel leakage from an SoC by a three-step process. Figure 9.2 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 9.2).

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 9.2)

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 9.2).

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 9.3) 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 9.3)

9.3.1 Step 1: Finding Leaky Time-Gate Tuples

In the first step (Figure 9.4), 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 9.1). 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.

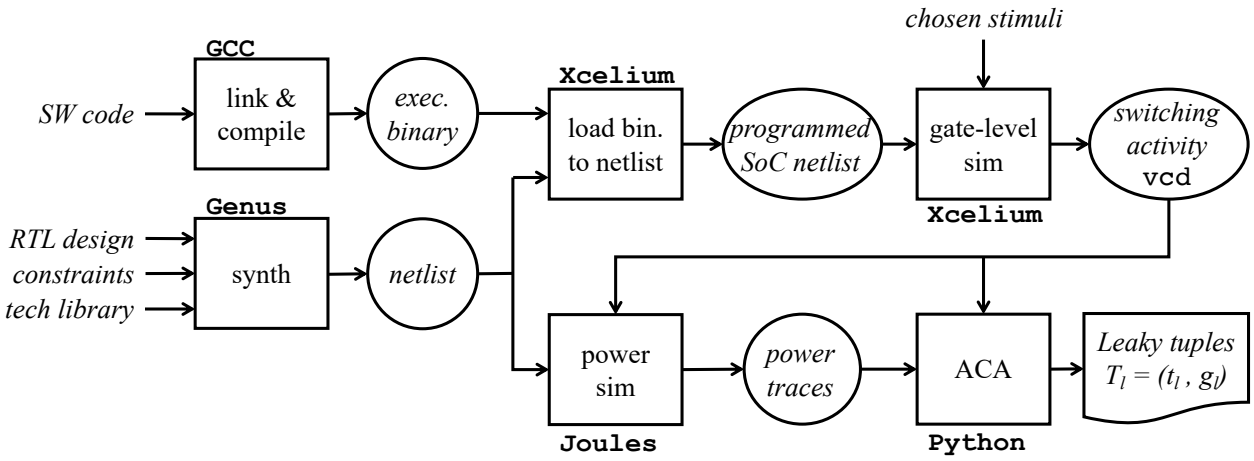
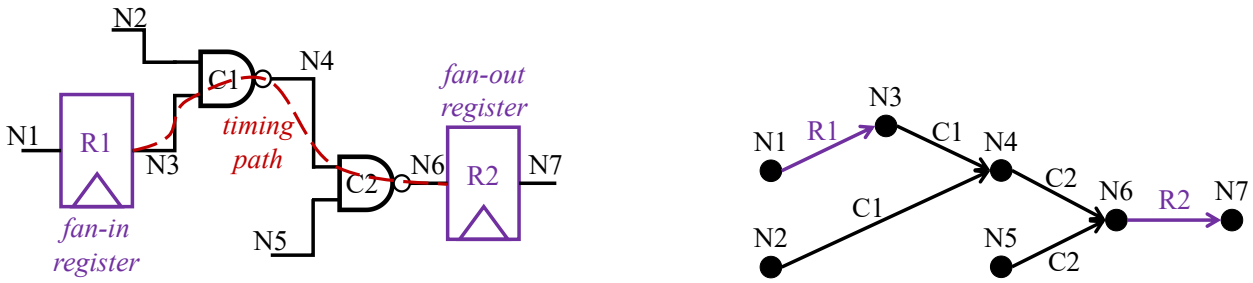


Figure 9.4: Layout of step 1 in RootCanal

9.3.2 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 9.3), 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 9.3 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.



(a) Example circuit (b) Corresponding netlist graph

Figure 9.5: An example for timing path, fan-in register, fan-out register, and gate-level netlist graph.

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.

Definitions

We use the following definitions to describe NGA.

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 9.5a 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 9.5a, $R1$ is a fan-in register for cells $C1$ and $C2$.

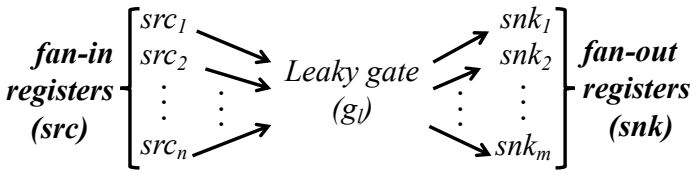


Figure 9.6: NGA uses fan-in and fan-out registers for each gate to determine its approximate location in the design.

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 9.5a, $R2$ 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 9.5b shows the netlist graph for the circuit in Figure 9.5a.

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

Algorithm 2 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

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 9.6, 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 2 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.

Algorithm 3 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

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 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 3 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 3 returns $\varnothing$), we use phase B and phase C of NGA to detect the pipeline stage s_l for a leaky gate g_l.

Algorithm 4 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 ← DIJKSTRA_PATH(netlist_graph, g_l, pipereg_MW)
    r_0 ← FIRST_REG(path_wb)
    if r_0 ∈ pipereg_{k+1} then          ▷ pipereg_{k+1}: pipeline register after stage_k
        return stage_k
    else if r_0 ∈ 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 ∈ path do
        if edge is register then
            return edge
        end if
    end for
end procedure
```

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 9.3). Next, detect possible combination of different pipeline stages from g_l's fan-in and fan-out registers. Algorithm 4 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 5 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 9.3). We follow Algorithm 5 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).

9.3.3 Step 3: Finding Leaky Instructions

Figure 9.7 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 9.3, we instrument the EM and MW stage registers with the PC signal (highlighted in yellow in Figure 9.3) for an RTL simulation of the SoC running the

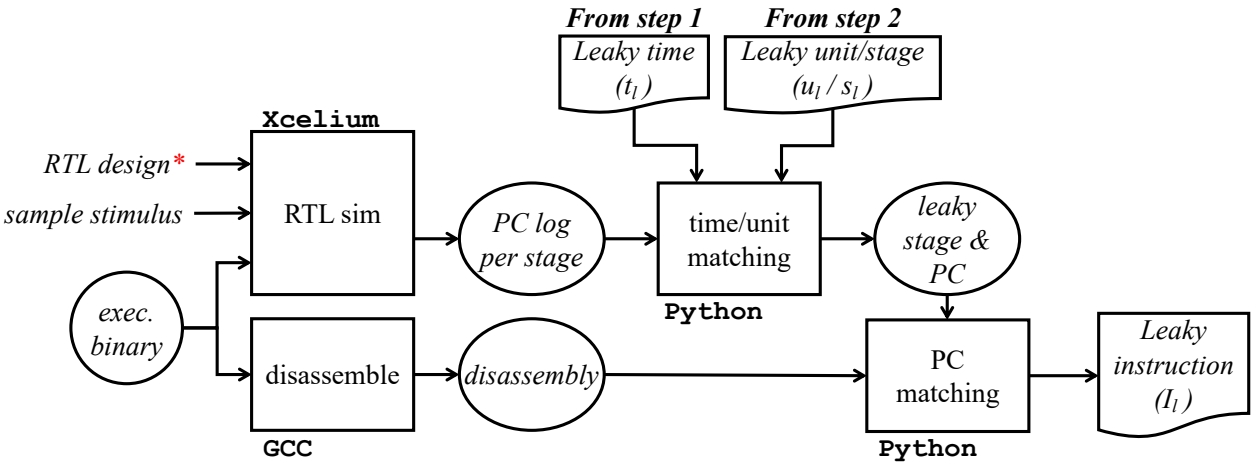


Figure 9.7: Layout of step 3 in RootCanal. The RTL design may need to be modified to pass on the PC signal to all pipeline registers. The executable binary is generated in the same way as in step 1.

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.

9.4 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 9.3). 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 9.2 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.

Table 9.2: Synthesis details for RISCV-SoC using Cadence Genus

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

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

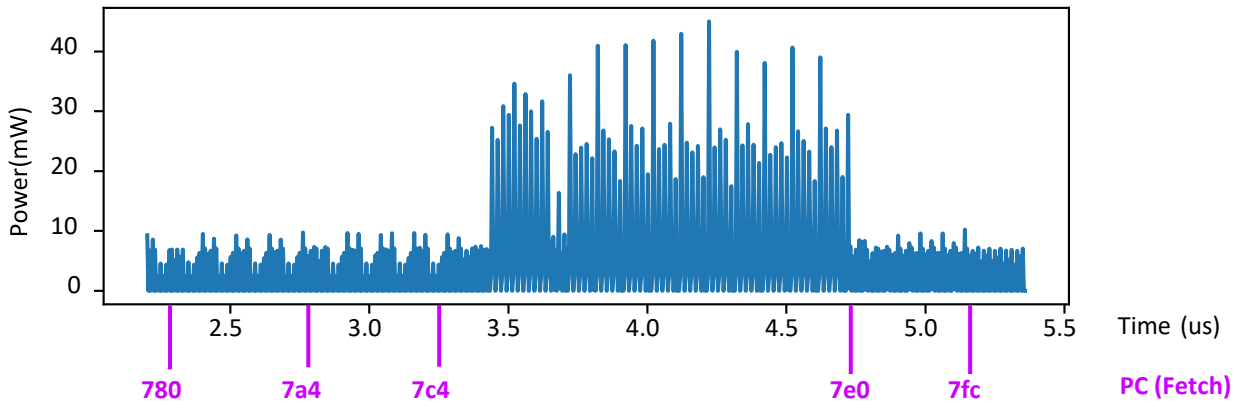


Figure 9.8: (Example 1) Average simulated power trace for SoC programming and running the AES hardware accelerator. The bottom X-scale links the power trace to Listing 9.2 through the value of the program counter (Fetch stage).

9.4.1 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.

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 9.1 and Listing 9.2 show the C code and 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).

```
int main() {
    // fixed expected memory address for key
    int* key = (int*)0x00000008;
    // fixed expected memory address for pt
    int* pt = (int*)0x00000018;
    // fixed expected memory address for ct
    int* ct = (int*)0x00000028;

    trigger_high();

    aes_base_address[1] = key[0];    // send key word 0
    aes_base_address[2] = key[1];    // send key word 1
    aes_base_address[3] = key[2];    // send key word 2
    aes_base_address[4] = key[3];    // send key word 3

    aes_base_address[5] = pt[0];     // send plaintext word 0
    aes_base_address[6] = pt[1];     // send plaintext word 1
    aes_base_address[7] = pt[2];     // send plaintext word 2
    aes_base_address[8] = pt[3];     // send plaintext word 3

    aes_base_address[0] = 0x00000006;        // reset and start AES
    aes_base_address[0] = 0x00000004;        // undo reset

    while(aes_base_address[17] != 0x00000001); // poll for AES completion

    // Read result from coprocessor and write to memory:
    for (i=0;i<4;i++)
        ct[i] = aes_base_address[13+i];

    trigger_low();
}
```

Listing 9.1: Firmware used for running the AES accelerator to simulate power traces

Figure 9.8 shows the average simulated power trace for the activity shown in Listing 9.2. 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.

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 9.2 summarizes the leakage sources identified by RootCanal from the perspective of the software, i.e., based on leakage from cells inside the pro-

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

cessor 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.

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.

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

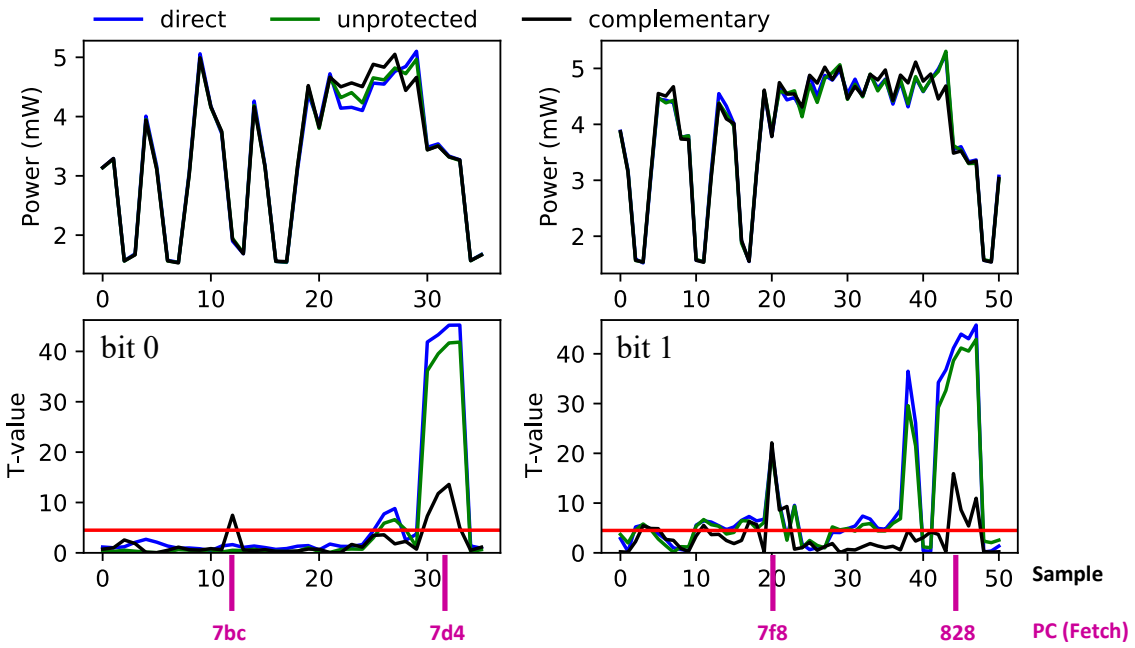


Figure 9.9: (Example 2) Average simulated power traces and TVLA results for redundant encoding schemes on bit-sliced PRESENT SBox

Setup

We use RootCanal to compare the side-channel impact of two fault encoding schemes presented in SKIVA [101]. 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 [33]. 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,

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

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

direct redundancy, and complementary redundancy). We feed the same 1024 random inputs to each scheme and simulate power traces. Figure 9.9 shows the average simulated traces over all inputs.

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 9.9, 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 9.3 and Listing 9.4 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.

9.4.3 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.

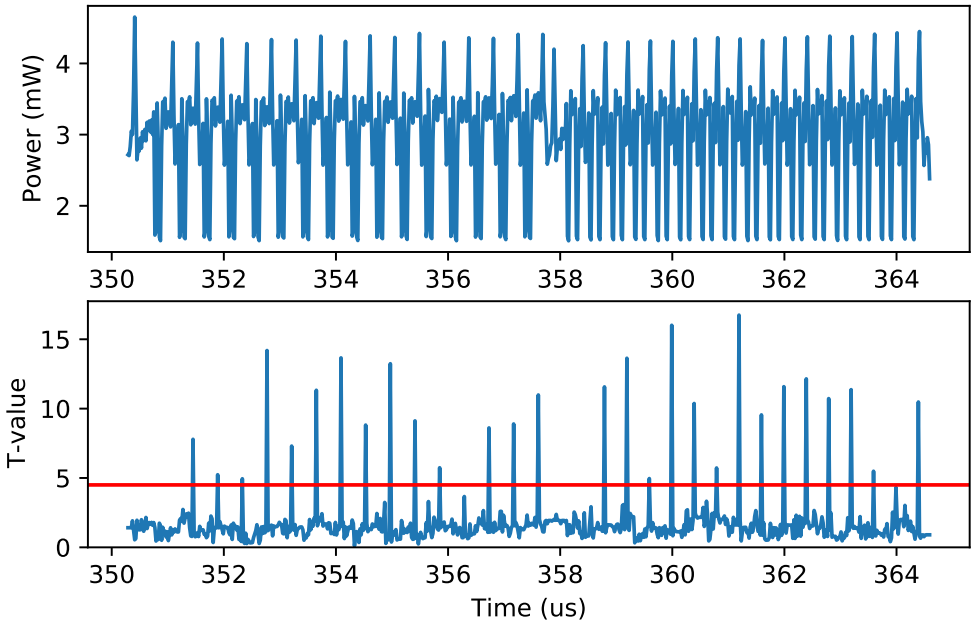


Figure 9.10: (Example 3) Average simulated power trace and TVLA result for the byte-masked AES example

Setup

We analyze an open-source byte-masked software implementation[3] of AES [171] that was previously shown to suffer from transition-based leakage [141, 70]. We analyze the power side-channel leakage of the initial `AddRoundKey` and the first round `SubBytes`. Listing 9.5 shows the assembly code for the body of the loops in `addRoundKey_masked` and `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.

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

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

Results

We perform the fixed vs. random TVLA test three times, each time with a new randomly chosen fixed value. Figure 9.10 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.

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

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

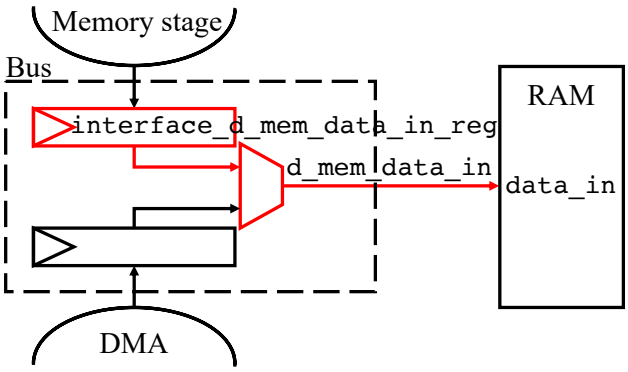


Figure 9.11: Leaking circuit in byte-masked software AES

values S_0 and S_1 are stored in memory, causing transitional leakage proportional to $S_0 \oplus S_1$.

$$S_0 = sbox[state[14] \oplus RoundKey[0][14]] \oplus Mask[5]$$

$$S_1 = sbox[state[15] \oplus RoundKey[0][15]] \oplus Mask[5]$$

$$S_0 \oplus S_1 = sbox[state[14] \oplus RoundKey[0][14]] \oplus sbox[state[15] \oplus RoundKey[0][15]]$$

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 9.11 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`

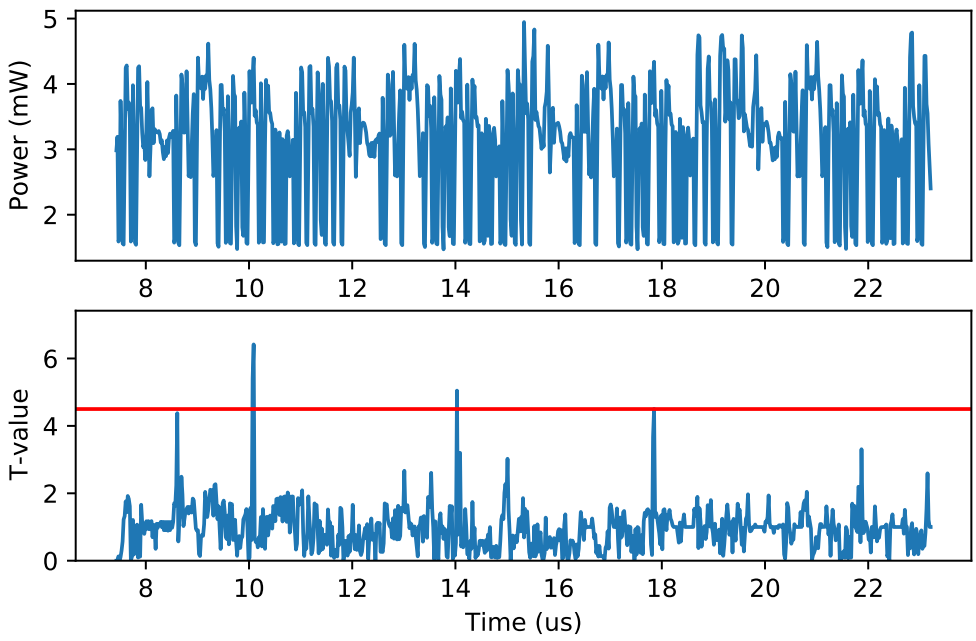


Figure 9.12: (Example 4) Average simulated power trace and TVLA result for bit-sliced masked PRESENT SBox

register should be cleared (zeroized or randomized [71]) 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.

9.4.4 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.

```
static void isw_mult(uint32_t *res,const uint32_t *op1,const uint32_t *op2) {
  int i,j;
  uint32_t rnd;

  for (i=0; i<MASKING_ORDER; i++) {
    res[i] = 0;
  }

  for (i=0; i<MASKING_ORDER; i++) {
    res[i] ^= op1[i] & op2[i];

    for (j=i+1; j<MASKING_ORDER; j++) {
      rnd = get_random();
      res[i] ^= rnd;
      res[j] ^= (rnd ^ (op1[i] & op2[j])) ^ (op1[j] & op2[i]);
    }
  }
}
```

Listing 9.6: (Example 4) ISW multiplication used in non-linear operations in the bit-sliced masked PRESENT SBox

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 [26]. The non-linear operations in the generated code use the masked multiplication introduced by Ishai *et al.* [89] (`isw_mult()` shown in Listing 9.6 and Listing 9.7 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.

Results

Figure 9.12 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

```
...
140:  sw     a5,0(s0)    # FDEMW # res[0] = (op1[0] & op2[0]) ^ rnd
144:  lw     a5,4(s2)    # FDEMW # a5 = op2[1]
148:  lw     a4,0(s1)    # FDEMW # a4 = op1[0]
14c:  and    a5,a5,a4    # FDEMW # a5 = op2[1] & op1[0]
150:  lw     a4,4(s0)    # FDEMW # a4 = res[1] (a4 = 0)
154:  xor    a5,a5,a4    # FDEMW # a5 = (op2[1] & op1[0]) ^ 0
158:  lw     a4,0(s2)    # FDEMW # a4 = op2[0]
15c:  lw     a3,4(s1)    # FDEMW # a3 = op1[1]
160:  and    a4,a4,a3    # FDEMW # a4 = op2[0] & op1[1]
164:  xor    a5,a5,a4    # FDEMW # a5 = (op2[1] & op1[0]) ^ (op2[0] & op1[1])
168:  xor    a5,a5,a0    # FDEMW # a5 = a5 ^ rnd
16c:  sw     a5,4(s0)    # FDEMW # res[1] = a5
170:  lw     a4,4(s2)    # FDEMW # a4 = op2[1]
174:  lw     a3,4(s1)    # FDEMW # a3 = op1[1]
178:  and    a4,a4,a3    # FDEMW # a4 = op2[1] & op1[1]
17c:  xor    a5,a4,a5    # FDEMW
180:  sw     a5,4(s0)    # FDEMW # res[1] = a5
...
```

Listing 9.7: (Example 4) Assembly code of the bit-sliced masked PRESENT SBox. Blue letters indicate leaky pipeline stages.

(leakage not expected at ISA level), being two consecutive ALU instructions, their intermediate results collide in the pipeline registers.

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 9.7). The multiplication operands now depend on different shares of the same variable, which creates side-channel leakage.

9.4.5 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

Table 9.3: 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 9.4: 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

of side-channel leakage, but also to explain it. Table 9.3 illustrates the data reduction we achieved for each design example. The design complexity of our RISC-V SoC 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 9.4. 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.

9.5 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.

Bibliography

[1] Implementation of SIMON and SPECK lightweight block ciphers for the SUPERCOP benchmark toolkit. `https://github.com/nsacyber/simon-speck-supercop/tree/master/crypto_stream/simon128128ctr/neon`. Accessed: 10-12-2019.

[2] Abubakr Abdulgadir, William Diehl, and Jens-Peter Kaps. An open-source platform for evaluation of hardware implementations of lightweight authenticated ciphers. In David Andrews, René Cumplido, Claudia Feregrino, and Marco Platzner, editors, *2019 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2019, Cancun, Mexico, December 9-11, 2019*, pages 1–5. IEEE, 2019.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 53–70. USENIX Association, 2016.

[4] Alric Althoff, Joseph McMahan, Luis Vega, Scott Davidson, Timothy Sherwood, Michael Taylor, and Ryan Kastner. Hiding intermittent information leakage with architectural support for blinking. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 638–649. IEEE, 2018.

[5] Kapil Anand, Matthew Smithson, Aparna Kotha, Khaled Elwazeer, and Rajeev Barua. Decompilation to compiler high ir in a binary rewriter. *University of Maryland, Tech. Rep*, 2010.

[6] Krste Asanovic, David A Patterson, and Christopher Celio. The berkeley out-of-order machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor. Technical report, University of California at Berkeley Berkeley United States, 2015.

[7] Krste Asanovic and Andrew Waterman. The risc-v instruction set manual. In *Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*, volume 2. RISC-V Foundation, 2019.

[8] Aydin Aysu, Ege Gulcan, and Patrick Schaumont. SIMON says: Break area records of block ciphers on FPGAs. *IEEE Embedded Systems Letters*, 6(2):37–40, 2014.

[9] 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.

[10] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*, pages 5–23. Springer, 2004.

[11] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *International Conference on Smart Card Research and Advanced Applications*, pages 64–81. Springer, 2014.

[12] Sahan Bandara, Alan Ehret, Donato Kava, and Michel A Kinsy. Brisc-v: An open-source architecture design space exploration toolbox. *arXiv preprint arXiv:1908.09992*, 2019.

[13] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A block cipher for low energy. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 411–436. Springer, 2015.

[14] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. Gift: a small present. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 321–345. Springer, 2017.

[15] Thierno Barry, Damien Couroussé, and Bruno Robisson. Compilation of a countermeasure against instruction-skip fault attacks. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, pages 1–6, 2016.

[16] 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.

[17] 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 *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 535–566. Springer, 2017.

[18] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 23–39. Springer, 2016.

[19] Omid Bazangani, Alexandre Iooss, Ileana Buhan, and Lejla Batina. Abby: Automating the creation of fine-grained leakage models. *Cryptology ePrint Archive*, 2021.

[20] 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.

[21] Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[22] G. T. Becker et al. Test vector leakage assessment (tvla) methodology in practice. In *International Cryptographic Module Conference*, 2013.

[23] George Becker, J Cooper, Elke DeMulder, Gilbert Goodwill, Joshua Jaffe, G Kenworthy, T Kouzminov, A Leiserson, M Marson, Pankaj Rohatgi, et al. Test vector leakage assessment (TVLA) methodology in practice. In *International Cryptographic Module Conference*, volume 1001, page 13, 2013.

[24] Arthur Beckers, Lennert Wouters, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede. Provable secure software masking in the real-world. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 215–235. Springer, 2022.

[25] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 616–648. Springer, 2016.

[26] Sonia Belaïd, Pierre-Evariste 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.

[27] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[28] Nicolas Belleville, Karine Heydemann, Damien Couroussé, Thierno Barry, Bruno Robisson, Abderrahmane Seriai, and Henri-Pierre Charles. Automatic application of software countermeasures against physical attacks. In *Cyber-Physical Systems Security*, pages 135–155. Springer, 2018.

[29] Eli Biham. A fast new DES implementation in software. In *International Workshop on Fast Software Encryption*, pages 260–272. Springer, 1997.

[30] 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 *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 321–353. Springer, 2018.

[31] 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.

[32] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 450–466, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[33] 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.

[34] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997.

[35] Joan Boyar, Philip Matthews, and René Peralta. Logic minimization techniques with applications to cryptology. *Journal of Cryptology*, 26(2):280–312, 2013.

[36] Jean-Baptiste Bréjon, Karine Heydemann, Emmanuelle Encrenaz, Quentin Meunier, and Son-Tuan Vu. Fault attack vulnerability assessment of binary code. In *Proceedings of the Sixth Workshop on Cryptography and Security in Computing Systems*, pages 13–18, 2019.

[37] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International workshop on cryptographic hardware and embedded systems*, pages 16–29. Springer, 2004.

[38] I. Buhan et al. Sok: Design tools for side-channel-aware implementions. *arXiv preprint arXiv:2104.08593*, 2021.

[39] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. Sok: Design tools for side-channel-aware implementations. Cryptology ePrint Archive, Report 2021/497, 2021. `https://ia.cr/2021/497`.

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

[41] Paul Caspi, Stavros Tripakis, and Pascal Raymond. Synchronous programming., 2007.

[42] 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.

[43] 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.

[44] CRYPTREC Lightweight Cryptography Working Group. *CRYPTREC Cryptographic Technology Guideline (Lightweight Cryptography)*, CRYPTREC Report March 2017.

[45] Joan Daemen, Michael Peeters, and Gilles Van Assche. Bitslice ciphers and power analysis attacks. In *International Workshop on Fast Software Encryption*, pages 134–149. Springer, 2000.

[46] Debayan Das, Mayukh Nath, Baibhab Chatterjee, Santosh Ghosh, and Shreyas Sen. STELLAR: A generic EM side-channel attack protection through ground-up root-cause analysis. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2019, McLean, VA, USA, May 5-10, 2019*, pages 11–20. IEEE, 2019.

[47] Wouter de Groot, Kostas Papagiannopoulos, Antonio de La Piedra, Erik Schneider, and Lejla Batina. Bitsliced masking and arm: Friends or foes? In *International Workshop on Lightweight Cryptography for Security and Privacy*, pages 91–109. Springer, 2016.

[48] Ronald De Keulenaer, Jonas Maebe, Koen De Bosschere, and Bjorn De Sutter. Link-time smart card code hardening. *International Journal of Information Security*, 15(2):111–130, 2016.

[49] Elke De Mulder, Samatha Gummalla, and Michael Hutter. Protecting RISC-V against side-channel attacks. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–4. IEEE, 2019.

[50] Jerry den Hartog, Jan Verschuren, Erik P. de Vink, Jaap de Vos, and W. Wiersma. PINPAS: a tool for power analysis of smartcards. In *SEC*, pages 453–457, 2003.

[51] Siemen Dhooghe and Svetla Nikova. My gadget just cares for me - how nina can prove security against combined attacks. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, pages 35–55, Cham, 2020. Springer International Publishing.

[52] Siemen Dhooghe and Svetla Nikova. My gadget just cares for me - how NINA can prove security against combined attacks. In Stanislaw Jarecki, editor, *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings*, volume 12006 of *Lecture Notes in Computer Science*, pages 35–55. Springer, 2020.

[53] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 131–141, 2017.

[54] Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Léo Perrin, Johann Großschädl, and Alex Biryukov. Triathlon of lightweight block ciphers for the internet of things. *Journal of Cryptographic Engineering*, 9(3):283–302, 2019.

[55] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1. 2. *Submission to the CAESAR Competition*, 2016.

[56] Stephen A. Edwards, Sungjun Kim, Edward A. Lee, Isaac Liu, Hiren D. Patel, and Martin Schoeberl. A disruptive computer design idea: Architectures with repeatable timing. In *27th International Conference on Computer Design, ICCD 2009, Lake Tahoe, CA, USA, October 4-7, 2009*, pages 54–59. IEEE Computer Society, 2009.

[57] T. El Motassadeq. Ccs vs nldm comparison based on a complete automated correlation flow between primetime and hspice. In *2011 Saudi International Electronics, Communications and Photonics Conference (SIECPC)*, pages 1–5. IEEE, 2011.

[58] Goodwill et al. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[59] He et al. 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, 2019.

[60] Mangard et al. Side-channel leakage of masked cmos gates. In *Cryptographers' Track at the RSA Conference*, pages 351–365. Springer, 2005.

[61] SLPSK et al. Karna: A gate-sizing based security aware eda flow for improved power side-channel attack protection. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 04-07, 2019*.

[62] Tiri et al. A logic level design methodology for a secure dpa resistant asic or fpga implementation. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 246–251. IEEE, 2004.

[63] Yao et al. Architecture correlation analysis: Identifying the source of side-channel leakage at gate-level. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST 2020)*. IEEE, 2020.

[64] Muhammad Arsath K. F, Vinod Ganesan, Rahul Bodduna, and Chester Rebeiro. PARAM: A microprocessor hardened for power side-channel attack resistance. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020*, pages 23–34. IEEE, 2020.

[65] Y. Fei et al. A statistics-based fundamental model for side-channel attack analysis. *IACR Cryptol. ePrint Arch.*, 2014:152, 2014.

[66] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[67] Research Institute for Secure Systems (RISCEC/AIST). Evaluation environment for side-channel attacks. `https://www.risec.aist.go.jp/project/sasebo/`. Accessed 7/26/2021.

[68] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *International workshop on cryptographic hardware and embedded systems*, pages 251–261. Springer, 2001.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering*, 8(1):1–27, 2018.

[74] Ilias Giechaskiel, Ken Eguro, and Kasper B. Rasmussen. Leakier wires: Exploiting fpga long wires for covert- and side-channel attacks. *ACM Trans. Reconfigurable Technol. Syst.*, 12(3), aug 2019.

[75] Ilias Giechaskiel, Ken Eguro, and Kasper Bonne Rasmussen. Leakier wires: Exploiting FPGA long wires for covert- and side-channel attacks. *ACM Trans. Reconfigurable Technol. Syst.*, 12(3):11:1–11:29, 2019.

[76] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-design and co-verification of masked software implementations on cpus. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1469–1468. USENIX Association, 2021.

[77] 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.

[78] 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.

[79] Thomas Given-Wilson, Annelie Heuser, Nisrine Jafri, and Axel Legay. An automated and scalable formal process for detecting fault injection vulnerabilities in binaries. *Concurrency and Computation: Practice and Experience*, 31(23):e4794, 2019.

[80] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and K Asanovic. Replicating and mitigating spectre attacks on an open source risc-v microarchitecture. In *Third Workshop on Computer Architecture Research with RISC-V (CARRV 2019)*, 2019.

[81] J. L. Gonzalez et al. Low delta-i noise cmos circuits based on differential logic and current limiters. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 46(7):872–876, 1999.

[82] Dahmun Goudarzi, Anthony Journault, Matthieu Rivain, and François-Xavier Standaert. Secure multiplication for bitslice higher-order masking: Optimisation and comparison. 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 3–22. Springer, 2018.

[83] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. *IACR Cryptology ePrint Archive*, 2016:486, 2016.

[84] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[85] Hendra Guntur, Jun Ishii, and Akashi Satoh. Side-channel attack user reference architecture board sakura-g. In *2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE)*, pages 271–274, 2014.

[86] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The led block cipher. In *International workshop on cryptographic hardware and embedded systems*, pages 326–341. Springer, 2011.

[87] M. He et al. Rtl-psc: Automated power side-channel leakage assessment at register-transfer level. In *2019 IEEE 37th VLSI Test Symposium (VTS)*, pages 1–6. IEEE, 2019.

[88] Miao Tony He, Jungmin Park, Adib Nahiyan, Apostol Vassilev, Yier Jin, and Mark Mohammad Tehranipoor. RTL-PSC: automated power side-channel leakage assessment at register-transfer level. In *VTS*, pages 1–6, 2019.

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

[90] M. A. KF et al. Param: A microprocessor hardened for power side-channel attack resistance. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 23–34. IEEE, 2020.

[91] P. Kiaei et al. Architecture support for bitslicing. Cryptology ePrint Archive, Report 2021/1236, 2021. `https://ia.cr/2021/1236`.

[92] P. Kiaei et al. Saidoyoki: Evaluating side-channel leakage in pre-and post-silicon setting. *IEEE International System-on-Chip Conference (SOCC)*, 2021.

[93] P. Kiaei and P. Schaumont. Synthesis of parallel synchronous software. *IEEE Embedded Systems Letters*, pages 1–1, 2020.

[94] Pantea Kiaei, Cees-Bart Breunesse, Mohsen Ahmadi, Patrick Schaumont, and Jasper Van Woudenberg. Rewrite to reinforce: Rewriting the binary to apply countermeasures against fault injection. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 319–324. IEEE, 2021.

[95] Pantea Kiaei, Tom Conroy, and Patrick Schaumont. Architecture support for bitslicing. Cryptology ePrint Archive, Paper 2021/1236, 2021. `https://eprint.iacr.org/2021/1236`.

[96] Pantea Kiaei, Archanaa S Krishnan, and Patrick Schaumont. Parallel synchronous code generation for second round light weight candidates. *NIST Lightweight Cryptography Workshop*, 2020.

[97] Pantea Kiaei, Zhenyuan Liu, Ramazan Kaan Eren, Yuan Yao, and Patrick Schaumont. Saidoyoki: Evaluating side-channel leakage in pre- and post-silicon setting. Cryptology ePrint Archive, Report 2021/1235, 2021. `https://ia.cr/2021/1235`.

[98] Pantea Kiaei, Zhenyuan Liu, Ramazan Kaan Eren, Yuan Yao, and Patrick Schaumont. Saidoyoki: Evaluating side-channel leakage in pre-and post-silicon setting. *Cryptology ePrint Archive*, 2021.

[99] Pantea Kiaei, Zhenyuan Liu, and Patrick Schaumont. Leverage the average: Averaged sampling in pre-silicon side-channel leakage assessment. In *Proceedings of the 2022 on Great Lakes Symposium on VLSI*, 2022.

[100] 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 *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 221–253. Springer, 2020.

[101] 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.

[102] Pantea Kiaei and Patrick Schaumont. Domain-oriented masked instruction set architecture for risc-v. *IACR Cryptol. ePrint Arch.*, 2020:465, 2020.

[103] Pantea Kiaei and Patrick Schaumont. Synthesis of parallel synchronous software. *IEEE Embedded Systems Letters*, 13(1):17–20, 2021.

[104] Pantea Kiaei and Patrick Schaumont. SoC Root Canal!: Root Cause Analysis of Power Side-Channel Leakage in System-on-Chip Designs. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2022.

[105] 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. *arXiv preprint arXiv:2204.11972*, 2022.

[106] 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`.

[107] Pantea Kiaei, Yuan Yao, and Patrick Schaumont. Real-time detection and adaptive mitigation of power-based side-channel leakage in soc. *arXiv preprint arXiv:2107.01725*, 2021.

[108] 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.

[109] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[110] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[111] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[112] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. Software countermeasures for control flow integrity of smart card c codes. In *European Symposium on Research in Computer Security*, pages 200–218. Springer, 2014.

[113] Edward A. Lee, Jan Reineke, and Michael Zimmer. Abstract PRET machines. In *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*, pages 1–11. IEEE Computer Society, 2017.

[114] Régis Leveugle, A Calvez, Paolo Maistri, and Pierre Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 502–506. IEEE, 2009.

[115] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[116] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[117] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[118] 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.

[119] Stefan Mangard and Kai Schramm. Pinpointing the side-channel leakage of masked AES hardware implementations. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2006.

[120] 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.

[121] Seiichi Matsuda and Shiho Moriai. Lightweight cryptography for the cloud: exploit the power of bitslice implementation. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 408–425. Springer, 2012.

[122] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In *USENIX Security Symposium*, pages 199–216, 2017.

[123] 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.

[124] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering:'grey box'modelling for instruction leakages. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 199–216, 2017.

[125] Amir Moradi. Side-channel leakage through static power - should we care about in practice? In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 562–579. Springer, 2014.

[126] Adib Nahiyan, Jungmin Park, Miao He, Yousef Iskander, Farimah Farahmandi, Domenic Forte, and Mark Tehranipoor. Script: A cad framework for power side-channel vulnerability assessment using information flow tracking and pattern generation. *ACM Trans. Des. Autom. Electron. Syst.*, 25(3), may 2020.

[127] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *International conference on information and communications security*, pages 529–545. Springer, 2006.

[128] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of non-linear functions in the presence of glitches. In *International Conference on Information Security and Cryptology*, pages 218–234. Springer, 2008.

[129] Colin O'Flynn and Zhizhang (David) Chen. Chipwhisperer: An open-source platform for hardware embedded security research. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*, volume 8622 of *Lecture Notes in Computer Science*, pages 243–260. Springer, 2014.

[130] Pádraig O'sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D Keromytis. Retrofitting security in cots software with binary rewriting. In *IFIP International Information Security Conference*, pages 154–172. Springer, 2011.

[131] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against spn structures, with application to the aes and khazad. In *International workshop on cryptographic hardware and embedded systems*, pages 77–88. Springer, 2003.

[132] Chester Rebeiro, David Selvakumar, and ASL Devi. Bitslice implementation of AES. In *International Conference on Cryptology and Network Security*, pages 203–212. Springer, 2006.

[133] Francesco Regazzoni, Stéphane Badel, Thomas Eisenbarth, Johann Großschädl, Axel Poschmann, Zeynep Toprak Deniz, Marco Macchetti, Laura Pozzi, Christof Paar, Yusuf Leblebici, and Paolo Ienne. A simulation-based methodology for evaluating the dpa-resistance of cryptographic functional units with application to CMOS and MCML technologies. In *Proceedings of the 2007 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2007), Samos, Greece, July 16-19, 2007*, pages 209–214, 2007.

[134] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 413–427. Springer, 2010.

[135] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, 1988.

[136] Patrick R Schaumont. *A practical introduction to hardware/software codesign*. Springer Science & Business Media, 2012.

[137] 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.

[138] Eric Schulte, Jonathan Dorn, Antonio Flores-Montoya, Aaron Ballman, and Tom Johnson. Gtirb: intermediate representation for binaries. *arXiv preprint arXiv:1907.02859*, 2019.

[139] Nader Sehatbakhsh, Baki Berkay Yilmaz, Alenka G. Zajic, and Milos Prvulovic. EMSim: A microarchitecture-level simulation tool for modeling electromagnetic side-channel signals. In *HPCA*, pages 71–85, 2020.

[140] Madura A Shelton, Łukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. Rosita++: Automatic higher-order leakage elimination from cryptographic code. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 685–699, 2021.

[141] 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.

[142] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. In *NDSS*, 2021.

[143] D. Šijačić et al. Towards efficient and automated side-channel evaluations at design time. *Journal of Cryptographic Engineering*, 10(4):305–319, 2020.

[144] Danilo Sijacic, Josep Balasch, Bohan Yang, Santosh Ghosh, and Ingrid Verbauwhede. Towards efficient and automated side-channel evaluations at design time. *J. Cryptogr. Eng.*, 10(4):305–319, 2020.

[145] R. Singh, T. Conroy, and P. Schaumont. Variable precision multiplication for software-based neural networks. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2020.

[146] Richa Singh, Saad Islam, Berk Sunar, and Patrick Schaumont. An end-to-end analysis of emfi on bit-sliced post-quantum implementations. *arXiv preprint arXiv:2204.06153*, 2022.

[147] Sergei P Skorobogatov and Ross J Anderson. Optical fault induction attacks. In *International workshop on cryptographic hardware and embedded systems*, pages 2–12. Springer, 2002.

[148] P. Slpsk et al. Karna: A gate-sizing based security aware eda flow for improved power side-channel attack protection. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[149] 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.

[150] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. Systematic classification of side-channel attacks: A case study for mobile devices. *IEEE Communications Surveys & Tutorials*, 20(1):465–488, 2017.

[151] F.-X. Standaert. How (not) to use welch's t-test in side-channel security evaluations. In *International Conference on Smart Card Research and Advanced Applications*, pages 65–79. Springer, 2018.

[152] François-Xavier Standaert. How (not) to use welch's t-test in side-channel security evaluations. *IACR Cryptol. ePrint Arch.*, 2017:138, 2017.

[153] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. Design and evaluation of smallfloat simd extensions to the risc-v isa. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 654–657, 2019.

[154] N. Timmers, A. Spruyt, and M. Witteman. Controlling pc on arm using fault injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35, 2016.

[155] S. Tiran et al. A model of the leakage in the frequency domain and its application to cpa and dpa. *Journal of Cryptographic Engineering*, 4(3):197–212, 2014.

[156] Elena Trichina. Combinational logic design for aes subbyte transformation on masked data. *Cryptology EPrint Archive*, 2003.

[157] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005.*, pages 7–12. IEEE, 2005.

[158] A. Vasselle, H. Thiebeauld, Q. Maouhoub, A. Morisset, and S. Ermeneux. Laser-induced fault injection on smartphone bypassing the secure boot-extended version. *IEEE Transactions on Computers*, 69(10):1449–1459, 2020.

[159] Fish Wang and Yan Shoshitaishvili. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.

[160] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[161] Shuai Wang, Pei Wang, and Dinghao Wu. Uroboros: Instrumenting stripped binaries with static reassembling. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 236–247. IEEE, 2016.

[162] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. The risc-v instruction set manual. volume 1: User-level isa, version 2.0. Technical report, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 2014.

[163] Clifford Wolf. Yosys open synthesis suite. `http://www.clifford.at/yosys/`.

[164] Shixiong Xu and David Gregg. Bitslice vectors: A software approach to customizable data precision on processors with simd extensions. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 442–451. IEEE, 2017.

[165] Shuo Yang, Shubhra Deb Paul, and Swarup Bhunia. Hands-on learning of hardware and systems security. *ASEE*, 9(2):1–25, 2021.

[166] Y. Yao et al. 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)*, pages 188–196. IEEE, 2020.

[167] Y. Yao et al. Verification of power-based side-channel leakage through simulation. In *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1112–1115, 2020.

[168] 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.

[169] Yuan Yao, Pantea Kiaei, Richa Singh, Shahin Tajik, and Patrick Schaumont. Programmable RO (PRO): A multipurpose countermeasure against side-channel and fault injection attack. *CoRR*, abs/2106.13784, 2021.

[170] Yuan Yao, Tuna Tufan, Tarun Kathuria, Baris Ege, Ulkuhan Guler, and Patrick Schaumont. Pre-silicon architecture correlation analysis (paca): Identifying and mitigating the source of side-channel leakage at gate-level. *IACR Cryptol. ePrint Arch.*, 2021:530, 2021.

[171] 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.

[172] Yuval Yarom and Katrina Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[173] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W Fletcher. Data oblivious ISA extensions for side channel-resistant and high performance computing. In *NDSS*, 2019.

[174] Bilgiday Yuce, Chinmay Deshpande, Marjan Ghodrati, Abhishek Bendre, Leyla Nazhandali, and Patrick Schaumont. A secure exception mode for fault-attack-resistant processing. *IEEE Trans. Dependable Secur. Comput.*, 16(3):388–401, 2019.

[175] Alexander Zeh, Andy Glew, Barry Spinney, Ben Marshall, Daniel Page, Derek Atkins, Ken Dockser, Markku-Juhani O Saarinen, Nathan Menhorn, and Richard Newell. Risc-v cryptographic extension proposals.

[176] Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. Flexpret: A processor platform for mixed-criticality systems. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, pages 101–110. IEEE Computer Society, 2014.