Towards Automated Detection of Single-Trace Side-Channel Vulnerabilities in Constant-Time Cryptographic Code

Ferhat Erata
Yale University
ferhat.erata@yale.edu

Ruzica Piskac
Yale Unviersity
ruzica.piskac@yale.edu

Victor Mateu
Technology Innovation Institute
victor.mateu@tii.ae

Jakub Szefer
Yale Unviersity
jakub.szefer@yale.edu

***Abstract*—Although cryptographic algorithms may be mathematically secure, it is often possible to leak secret information from the implementation of the algorithms. Timing and power side-channel vulnerabilities are some of the most widely considered threats to cryptographic algorithm implementations. Timing vulnerabilities may be easier to detect and exploit, and all high-quality cryptographic code today should be written in constant-time style. However, this does not prevent power side-channels from existing. With constant time code, potential attackers can resort to power side-channel attacks to try leaking secrets. Detecting potential power side-channel vulnerabilities is a tedious task, as it requires analyzing code at the assembly level and needs reasoning about which instructions could be leaking information based on their operands and their values. To help make the process of detecting potential power side-channel vulnerabilities easier for cryptographers, this work presents Pascal: Power Analysis Side Channel Attack Locator, a tool that introduces novel symbolic register analysis techniques for binary analysis of constant-time cryptographic algorithms, and verifies locations of potential power side-channel vulnerabilities with high precision. Pascal is evaluated on a number of implementations of post-quantum cryptographic algorithms, and it is able to find dozens of previously reported single-trace power side-channel vulnerabilities in these algorithms, all in an automated manner.**

***Index Terms*—power side-channels, differential program analysis, hamming weight, post-quantum cryptography**

1. Introduction

Although cryptographic algorithms may be formally proven to be secure, often it is the case that there is a gap between a mathematical formalization of the algorithm and its implementation; the implementation might leak the secret information. When one can infer secret information from either observing the running time of the program [1], [16], [28], [124], or its power consumption [57], [58], [63], or electromagnetic emanations (EM) [82], traditionally this is referred to as a side-channel vulnerability. Timing side-channel vulnerabilities have been studied the most, and currently we have numerous powerful analysis techniques and tool that can detect if code is resistant against timing side-channel attacks or to help write so-called constant-time code [7], [9], [22], [24], [26], [32], [35], [52], [118], [120].

However, even if the programmers follow the constant-time paradigm, this still does not prevent power side-channels attacks. Computing and manipulating different data causes different power consumption. An attacker with physical access to sampling power consumption [73], [74], or with remote access to power-related performance counters [62], can observe power variations as an algorithm executes. This can leak secure data.

If the code is non-constant time, the attackers already have an easier (timing) attack that they can leverage. Therefore, in this paper, we consider only constant-time code. The power side-channels that still remains in constant-time code are often difficult to detect, even for experts. Therefore, the focus of our work is on developing automated reasoning methods to find potential locations of power side-channel leaks in cryptographic code. The existing work mostly relies on hypothesis testing [49], [50], [64], [91] and known power side-channel attacks are found experimentally by collecting code traces and conducting various statistical analyzes or using target specific leakage simulators [66], [95], [96]. Without collecting traces, our approach analyzes the binaries and pinpoint location(s) of potential single-trace power side-channel vulnerabilities, which can then be confirmed by using methods such as TVLA, and then finally fixed. We built a tool, Pascal, and empirically evaluated it on 30 publicly disclosed power-side channel vulnerabilities mainly in post-quantum cryptographic algorithms; Pascal found all previously reported single-trace vulnerabilities in these algorithms.

Pascal combines differential program analysis [44], [72], [75] with optimization queries [21], [61], [68], [94] in order to identify the most vulnerable locations in the binaries of a constant-time code. We use Hamming weight and Hamming distance leakage models [27], [57], [58]. The Hamming weight, defined on a binary string, is the number of 1's in the string. The Hamming weight leakage model assumes that the Hamming weight of the operands is strongly correlated with the power consumption.

In symbolic analysis, some or all variables (or in the context of binary analysis, registers or memory locations) are represented by a symbolical variable. This symbolical variable represents all possible values that the variable could take. As the execution proceeds over each instruction, symbolic execution derives logical formulas defined using these symbolical variables. These formulas, defined in symbolical variables, represent a summary of

This work was supported in part by NSF grant 2245344.

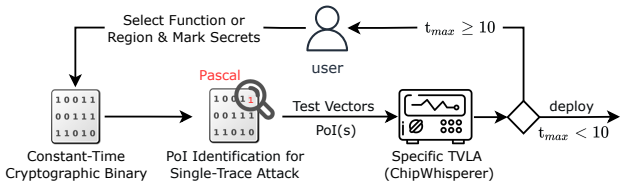

Figure 1: Pascal reports addresses of Point of Interests (PoI) (vulnerable instructions) in the binary and generates Test Vectors for TVLA analysis if wanted.

a program execution. We use these formulas to derive limits for the range of values the symbolical variable can represent. In differential part of the analysis, we create optimization queries to find the minimum and maximum Hamming weight differences for operands for each of the instructions. At the binary code level, if the Hamming weight difference is typically large in destination registers, large power measurement difference is assumed, and the code location can be vulnerable to practical power analysis attacks in that the secret can be guessed with one or few traces.

Our tool Pascal, at high-level, formally locates these vulnerable instructions in the binaries where there is a large separation between the Hamming weights of *register writes*. For example, through the symbolic execution and the optimization queries the tool may find that some instruction only can have secret-related inputs of either $0x00\ldots0$ or $0xFF\ldots F$, then this instruction may lead to potential power side-channel vulnerability since power consumption of computation related to $0x00\ldots0$ is much different from for $0xFF\ldots F$. Automatically analyzing the code to find all the possible ranges of register values, and their Hamming weights, is a non-trivial task and one of the main contributions of this paper.

The methodology of Pascal is shown in Figure 1. Pascal is designed to be used in a standalone manner, allowing developers to easily check for single-trace vulnerabilities in their code. However, once a point of interest is found, developers can then use TVLA or other statistical analysis methods to confirm the vulnerability empirically. To aid in this process, Pascal also generates test vectors.

With our tool, we can analyze cryptographic code to get the differential Hamming weights of all the possible operands for all instructions in a target function. The input code can be in a variety of programming languages, as it is first compiled into binary before analysis is done. Pascal can naturally explore impacts of different target architectures and optimizations by compiling the code with the different flags or optimization levels and then running the analysis on the different binaries.

To empirically evaluate Pascal, we examined, to the best of our knowledge, all known power single-trace side-channel attacks against the publicly available constant-time implementations of post-quantum cryptographic algorithms submitted to NIST's post-quantum cryptography standardization process. Our method successfully pinpointed more than 30 known power-side channel vulnerabilities in constant-time implementations in a variety of cryptographic schemes, from Elliptic Curve [25] to Lattice-based [70] and to Lightweight [113] cryptography. Pascal is the first tool that can find the locations of potential vulnerabilities in an automated way with limited programmer input: they only need to specify a function of interest and which secret inputs are related to it, and the tool finds if and where there are instructions that could be sources of power side-channel leakage.

Detecting potential power side-channel vulnerabilities is a tedious task, as they require analyzing code at the *assembly level* and need reasoning about which instructions could be leaking information based on their operands. Our tool is designed to automatize this procedure by *formally* analyzing given binaries. It is developed on top of `angr` [99], [100], [110] binary analysis framework, `Z3` [36] and `CVC5` [12] SMT solvers, and single-objective linear optimization algorithms over bitvector terms [21], [61], [93] of `Z3` [20] and `OptiMathSAT` [94] solvers.

Our work is one of the first to approach the problem of detecting power side-channel vulnerabilities using a *white-box* technique that considers the semantics of the tainted instruction flow and the symbolic states of the registers. Our work is also the first to adapt the relatively new concept of *Relational Symbolic Execution* [44], [72] for power side-channel analysis.

Contributions. In summary, this work makes the following contributions: **(1)** Definition of a robustness measure of cryptographic code against power side-channel attacks, which reveals and characterizes the Hamming weight patterns of instruction operands. From an attacker's perspective, it allows them to identify the most vulnerable code regions for an attack, whereas, from a cryptographic library developer's perspective, allows them to pinpoint the most vulnerable code regions for hardening them (see §3). **(2)** Development of the novel prototype analysis tool for automatic detection of potential power side-channel vulnerabilities in constant-time implementations of cryptographic algorithms at the binary code level (see §4). **(3)** Systematic analysis of the power side-channel attacks known in the literature to validate that Pascal can identify the most vulnerable binary code locations of a constant-time code under Hamming weight or Hamming distance leakage model. We successfully identified 30 different vulnerabilities in constant-time implementations in a variety of cryptographic schemes (see §5).

Availability. https://github.com/ferhaterata/pascal

2. Motivating Example

In this section, we illustrate a power side-channel leakage in a constant-time code and its detection.

A Constant-Time Code. It is a well-known fact that the power consumption during certain stages of a cryptographic algorithm exhibits a strong correlation with the Hamming weight of its underlying variables, i.e., Hamming weight leakage model [27], [62], [63], [83], [107], [121]. This phenomenon has been widely exploited in the cryptographic literature in various attacks targeting a broad range of schemes [4], [5], [8], [19], [47], [54], [56], [69], [84]–[86], [101]–[105], [109], [123]. Due to the intrinsic connection between the Hamming weight of

constant-time code		lifted binary code
$f(sum_{[8]S},\ x_{[8]S})$:		$f(sum_{0[8]S},\ x_{0[8]S})$:
$mask := (x - 64) \gg 7$	$\rightsquigarrow$	$r_0 := x_0 - 64$
$sum := sum + (\sim mask \wedge x)$		$r_1 := r_0 \gg 7$ ⋆
		$r_2 := \sim r_1$
		$r_3 := r_2 \wedge x_0$
		$r_4 := sum_0 + r_3$

Figure 2: Example of conditional addition written in a constant-time style using masking.

Arm32, O0	O3, cortex-m4, armv7e-m
`mov r3, r0`	`sub.w r3, r1, 0x40`
`...`	`bic.w r1, r1, r3, asr 7` ⋆
`sub r3, r3, 0x40`	`add r0, r1`
`asr r3, r3, 7` ⋆	`uxtb r0, r0`
`...`	`bx lr`
`bx lr`	

Figure 3: Disassembly of conditional addition from Figure 2 based on different compiler flags.

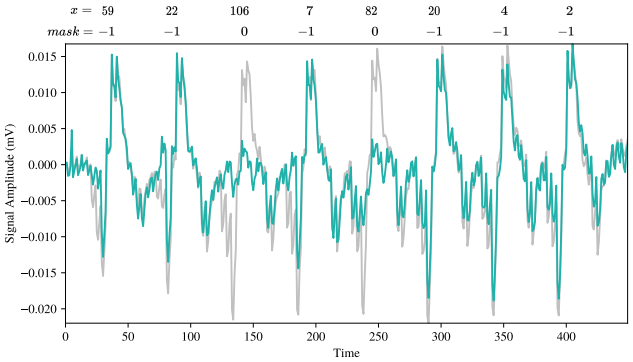

Figure 4: A *single-trace template-based attack*. The trace in green exposes the differences between the mask value being either 0 or -1. The reference trace in gray is a template having the max. *Hamming weight difference*.

intermediate cipher variables and the power consumption of software implementations of cryptographic algorithms, the most vulnerable binary locations of a constant-time code are those that are most likely to be executed on a set of few intermediate values whose Hamming weight differences are considerably large. For example, consider the conditional addition function, $if(x \geqslant 64)\ then\ sum := sum + x$, where sum is the running sum and x is the next secret byte. In this simple example, the tainted intermediate variables related to the secret input x are colored red to help visualize the propagation of the secret information (aka. forward tainting). This code has obvious timing vulnerability, since execution of the addition operation depends on the secret value x. However, this code can be converted to constant-time code using *masking*, as shown in Figure 2.

In the constant-time version of the conditional addition function, a *mask*, containing 0 or -1 (= $0xFF$), replaces the if-condition. The mask calculation is shown in a simplified machine code in Figure 2. An arithmetic right shift operation (indicated by a ⋆) extracts the sign bit of the 8-bit intermediate value, $m_{0[8]S}$ (S means signed two's complement). The processed message bit is leaked **(i)** neither in a branch, **(ii)** nor in an address-index look-up, **(iii)** nor through a loop bound, **(iv)** nor in differences in execution time, but through power consumption which differs between processing operand of all zeros vs. all ones; and here the *mask* either contains exactly either ones or zeroes only. Chances that these two different processed values can be detected by analyzing the power consumption of the device are very high. This will give attackers a good attack point (aka. Point of Interest (PoI) or Point of Leakage) for the analysis of the power consumption of a constant-time code under hamming weight leakage model, since in particular an attacker will infer a statistical property of the secret value, which is in our case to be either in the range of $[-128, 63]$ or $[64, 127]$. The PoIs are identified in the binaries of the code.

In Figure 3, the binary outputs of `arm-none-eabi-gcc` compiler toolchain targeting 32-bit Arm architecture with the different compiler flags generates two different vulnerable locations. `asr` instruction performs arithmetic shift right operation over operands; and `bic.w`, Bitwise Bit Clear, performs a bitwise AND of a register value and the complement of an immediate value, and writes the result to the destination register. As we can see that different compiler flags generate different code and different vulnerable locations, it is necessary to focus on the assembly code level for the power side-channel checking.

Test Vector Leakage Assessment. The Test Vector leakage Assessment (TVLA) [50] identifies differences between two sets of side channel measurements by computing the t-test for the two sets of measurements. It is being used in the literature to confirm the *presence* or *absence* of side leakages for power traces [39], [64], [83], [88], [91], [108].

To empirically show that the vulnerability exists in the conditional addition function, we perform a *single-trace template-based attack* using differential power analysis [31], [57] on ChipWhisperer UFO STM32F3 target board [73]. STM32F303 is a microcontroller based on the 32-bit ARM Cortex-M4 processor core, which is commonly used in embedded systems such as IoT/Edge devices. Based on the *Point of Interest* that we identified at line 2 in Figure 2, we fix the $sum = 0$ and randomly draw eight sensitive s values from two sets, $[-128, 63]$ or $[64, 127]$, to take conditionally running sum of those eight values. We repeat this procedure to obtain 100 traces from each. In Figure 4, we show averaged differential power traces of the two sets to create the *template* in gray color. The green one is a *single-trace*, having obtained by triggering the function f eight times. Evidently, the difference between a mask value being 0 or -1 is immediately visible in the respective trace when f is compiled at -O3.

3. Overview

Section §3.1 explains the data-dependent power consumption phenomenon in CMOS circuits, and Section

§3.2 formalizes the Hamming weight and Hamming distance leakage models. Section §3.3 introduces the notion of differential Hamming weights and Hamming distances and relates it to differential program analysis. Finally, in Section §3.4, we introduce ω-class sampling model to quantify self-information content based on possible Hamming weight classes at a register.

3.1. Data Dependent Power Consumption

Power analysis attacks are built upon the observation that the power consumption of CMOS digital circuits is data-dependent *by design*. Each bit flip requires one or more voltage transitions from 0 to high (or vice versa). Different data values typically entail differing numbers of bit flips and therefore produce distinct power traces [30]. Therefore, any circuit not explicitly designed to be resistant to power attacks has data-dependent power consumption. However, in a complex circuit, the differences can be so slight that they are difficult to distinguish from a *single trace*, particularly if an attacker's sampling rate is limited [62], [117]. Therefore, it is necessary to use statistical techniques such as Differential Power Analysis [57] and Correlation Power Analysis [27] across multiple power traces [62]. In these analysis methods the *Hamming weight* or *Hamming distance* leakage models are preferred to quantify the power consumption of a CMOS circuit.

3.2. Hamming Weight and Hamming Distance

Leakage Models. In the value-based leakage model, the leakage correlates with the Hamming Weight of values at registers leaked by instruction execution, that differs from the transition-based model where the value leaked by an instruction correlates with the Hamming distance between the result of its new value and the previous old value in the register. Pascal supports both models. Additionally, Pascal incorporates an information theoretical, value-based measure, namely Hamming weight class sampling model, to quantify leakages.

Definition 1 (Hamming weight)**.** Given an n-bit element $v \in \mathbb{F}_n$, let $0 \leq \omega(v) \leq n$ be its Hamming weight, i.e., the number of bits that are set to one in the binary representation of v.

The leakage model considered in this paper are the *Hamming weight* and *Hamming distance* leakage models [27], [57], [58]. We use $\omega(v)$ to refer to the Hamming weight of a value v and use $d(v, v')$ to refer the Hamming distance of values v and v'.

Definition 2 (Hamming distance)**.** Hamming distance is a metric for measuring the edit distance between two sequences. Given two n-bit elements $v, v' \in \mathbb{F}_n$, let $0 \leq d(v, v') \leq n$ be their Hamming distance, i.e., $d(v, v') \triangleq \omega(x \oplus y)$. $\uparrow d(v, v')$ and $\downarrow d(v, v')$ represents *maximum* and *minimum* Hamming distance between v, v', respectively.

Due to noise in power measurements, it is not possible to directly deduce the Hamming weight for a particular value of interest, that is being updated inside the central processing unit of the microcontroller or a microprocessor. It is nevertheless possible to find out the Hamming weight of a particular value by the means of a differential power analysis.

Definition 3 (Power Consumption & Power Trace)**.** Given a function $F : \mathbb{F}_m \mapsto \mathbb{F}_n$ that executes a sequence of instructions for an arbitrary $x \in \mathbb{F}_m$, we denote by $P(F(x))$ the power consumption of this function and by $\overline{P}(F(x))$ its average over multiple repetitions. A power trace $P_{T_n}(F(x))$ is a vector of n values of power samples, where n is the number of samples taken at each time instant over the execution of function F.

We experimentally verified that this property holds on a number of microcontrollers. In Figure 10 (Appendix) on the left graph, the function F corresponds to execution of $R_d := R_d \gg x$ including an `asr` instruction. Note that knowledge of $\omega(F(x))$ does not necessarily mean that $F(x)$ is uniquely recovered. However, 0 and $2^n - 1$ within $\mathbb{F}_n$ are the only elements with Hamming weight 0 and n respectively.

Definition 4 (Monotonic relation between Power Consumption and Hamming weight)**.** If $\omega(F(x)) > \omega(F(y))$ then $\overline{P}(F(x)) > \overline{P}(F(y))$ for two different $x, y \in \mathbb{F}_n$.

The power consumed by the attacked device has to be a monotonic function of the Hamming weight of the processed data for the Hamming weight leakage model to work. This observation is validated experimentally in the literature such as on Intel and ARM's mobile, desktop, and server CPUs [62]. We also confirmed that 8-bit ATXmega128D4-AU (AVR instruction set), 32-bit STM32F3/STM32F4 (ARM Cortex-M4), STM32F0 (ARM Cortex-M0) microcontrollers exhibits a monotonic relation between power consumption and Hamming weights of data, and we have also found that the energy consumption of ultra-low powered devices (MSP430 instruction set of the MSP430FR5994 microcontroller) is monotonic with respect to the Hamming weight of the processed data. Almost all the power and EM side-channel attack papers use the Hamming weight as a metric and accept monotonicty assumptions for the power consumption behavior of the attacked device.

3.3. Differential Power Analysis

Different code execution paths may leave different power traces, and this may lead to different instructions for each path and affect power consumption, this can be also classified as a timing vulnerability. This occurs based-on secret dependent branches. Since constant-time code always generates the same length of power traces, we omit this type of behavior in our analysis. In fact, there is no secret-dependent branching and secret-dependent loop bounds among the Post-Quantum Cryptographic (PQC) implementations [55] nor in reference implementations among finalists of NIST's PQC standardization process [71] which we analyze later in this work. However, even with the same control flow, different data being manipulated in the microcontroller or the microprocessor affects power consumption.

Definition 5 (Differential Hamming weight)**.** Given two n-bit elements $v, v' \in \mathbb{F}_n$, let $0 \leq \Delta_\omega(v, v') \leq n$ be

their differential Hamming weight or Hamming weight difference, i.e., $\Delta_\omega(v, v') \triangleq |\omega(v) - \omega(v')|$. $\uparrow \Delta_\omega(v, v')$ and $\downarrow \Delta_\omega(v, v')$ represents *maximum* and *minimum* differential Hamming weights between v, v', respectively.

The notion of differential Hamming weight that we have coined in this work is subtly different from that of Hamming distance, which is a metric for measuring the edit distance between two sequences, $d(x, y) \triangleq \omega(x \oplus y)$. For instance, $d(\langle 11110000 \rangle_U, \langle 00001111 \rangle_U) = 8$ whereas their Δ_ω is 0.

Cryptographic implementations in hardware, such as an AES accelerator in a microcontroller where the algorithm is not running as a software process on a dedicated hardware accelerator, are much more likely to be vulnerable to the Hamming distance leakage. Since they typically have only a few interconnections between registers, this leads to a detectable Hamming distance as opposed to a Hamming weight. Therefore, Hamming distance leakage model is commonly used in power or EM attacks on hardware implementations [74] of cryptographic algorithms, while Hamming weight leakage model targets software implementations running on a microcontroller or a microprocessor. Nevertheless, Pascal additionally employs Hamming distance leakage model in its symbolic analysis.

A standard *safety* property states that nothing bad can happen along one execution trace; however, information flow properties relate two execution traces and called 2-hypersafety properties in literature [34]. Differential Hamming weight and Hamming distance will be used in detection of *leakage points* (cf. §4.2) and can be thought as an interpretation of *2-hypersafety property* in power side-channel analysis. If the Hamming weight of two different values is the same or their difference is close to zero, then it is very hard for an attacker to differentiate them from the power traces and there is likely no vulnerability. On the other hand, if the Hamming weight of two different values has large difference, then it is much more likely an attacker can detect these different values from the power traces.

Let's assume that k_1 and k_2 are sensitive 8-bit inputs to a function $c \leftarrow F(k)$ of a cryptographic code and that their Hamming weights are $\omega(k_1) = 8$ and $\omega(k_2) = 0$. In this case, their Δ_ω is maximum possible since for 8-bit vectors ($\mathbb{F}_8$) there cannot be bigger difference than 8. If we aligned averaged power traces of two executions, $F(k_1)$ and $F(k_2)$, we would expect to see a distinguishing spike at the point where $F(k_1)$ and $F(k_2)$ are executed, from Definition 4, i.e. $|\overline{P}(F(k_1)) - \overline{P}(F(k_2))| > 0$. In Figure 10 on the right graph, such spikes can be seen. Those power traces are collected from an ARM Cortex-M4 microcontroller's 32-bit target architecture to profile data-oriented differences on *asr* instructions on the same cryptographic implementation. The most distinguishable Hamming weight differences are those closer to 8 and the least distinguishable ones are those closer to 0.

Definition 6 (Formal Leakage Definition)**.** Let P be a program with a function $f(k, x) = c$, where k is the sensitive (secret) input, x is the public input, and c is the output. We define a leakage in the binary of f if there exist two sensitive input vectors k_1 and k_2, that slice the secret input domain of k into two subsets at the destination register of location i with a distinguishable Hamming weight difference. Therefore, P has a leakage if $\forall x, k_1 \in \vec{k_1}, k_2 \in \vec{k_2}, |\omega(r_i(k_1, x)) - \omega(r_i(k_2, x))| \leq \nu$ where ω is a function that returns Hamming weight of the register r and ν is a threshold to distinguish hamming weight difference. ν is automatically set with the maximum width that the register r can take at location i. Those points can be easily exploited by the attacker through a single-trace attack.

We conducted a literature review on the recent power/EM side-channel attacks (cf. Table 6 in Appendix A). We observe some commonalities among those *single-trace side-channel attacks* [4], [5], [8], [19], [47], [56], [69], [84], [85], [101]–[104], [109] and have made two observations:

Observation 1 (Case splits are *points of interest*)**.** Attackers identify the most vulnerable instruction locations in the binaries as specific *points of interest (PoIs)*, where the domain of an intermediate value is significantly shrunk into a set of values (*cases*). Thus, they only need to observe the power traces at those PoIs to detect the leakage; in fact, they can create templates for each Hamming weight *classes* where those values belong to and then compare the power traces to the templates to detect the leakage using statistical techniques.

Observation 2 (Differential Hamming weights at *points of interests* reflect distinguisable observations)**.** Similarly, intermediate values where differential Hamming weight among them become high are of interest since they are highly distinguishable on a power or EM trace.

Existing work introduces the notion of *determiner* in their single-trace attacks on Lattice-based key encapsulation [104].

Definition 7 (Determiner)**.** The determiner is an intermediate value that is defined according to a sensitive bit value, and the difference between the Hamming weights of the elements of the determiner domain is greater than or equal to 2. The cardinality of the determiner domain is 2.

3.4. Hamming Weight Classes

For deterministic systems, the Shannon entropy can also be used to give a measure of the leakage of the side-channel, corresponding to the observation gain (on the secret) after one round of observation [67], [77].

If an adversary learns the Hamming weight ω of an 8 bit-width intermediate value v, then this reduces the uncertainty about v as only $\binom{8}{w}$ out of 256 values satisfy the observed Hamming weight. For example, observing an ω of 4, it gives an attacker 70 possible secret keys, whereas observing an ω of 0 or 8 leads to only one possible secret key. Accordingly, the occurrence of Hamming weight *classes* close to 4 are more likely, but bring less information about the secret key [78].

Since Pascal relies on the program's compiled semantics and data encoding at intermediate values (destination registers), it naturally identifies those interesting points where their *Shannon's entropy* [23] becomes significantly low. Therefore, we propose an

TABLE 1: ω classes and probabilities for $\mathbb{F}_8$

ω_i	0	1	2	3	4	5	6	7	8
$\lvert\omega_i\rvert$	1	8	28	56	70	56	28	8	1
$\mathbb{P}_{\omega_i}$	.004	.031	.109	.219	.273	.219	.109	.031	.004

approximate model to obtain the entropy of destination registers to quantify the leakage over *single-run* (or *single-trace*) in the attack.

Definition 8 (ω-class sampling model)**.** It samples each *Hamming weight class* ω_i of a destination register r, and if we obtain a value, we include the probability $\mathbb{P}_{\omega_i}$ of that class (see Table 1) in entropy calculation.

$$\tilde{\eta}(r) = -\sum_{i=0}^{n} \mathbb{P}_{\omega_i}(r) \cdot \log_2 \mathbb{P}_{\omega_i}(r), \text{ where } r \in \mathbb{F}_n$$

For instance, a register having only all-ones or all-zeros has $\tilde{\eta}$ of 1.00. The maximum entropy for an intermediate value in $\mathbb{F}_n$ is 2.54.

4. Approach

In our formal analysis of binaries, we will be using the notion of differential Hamming weight, Hamming distance, and Hamming weight classes.

Threat Model. Our work mainly focuses on *single-trace side-channel attacks* against constant-time implementations. Single-trace side-channel attacks aim to extract a secret value from one side-channel measurement. First, the attacker has physical access to the identical device and can configure it with known code and secrets to capture power traces. Therefore, the adversary can capture multiple measurements to create *templates* corresponding to known keys or intermediate values of the target cryptographic algorithm implementation. Second, the adversary has sufficient knowledge about the details of the target software implementation, i.e., the adversary downloads and inspects the publicly available software packages, e.g., candidates submitted to NIST PQC standardization process. When running the attack, the adversary is limited to a single measurement. With single-trace attacks, key generation and encapsulation can be targeted since they use one-time values (*ephemeral secrets*).

4.1. Tooling Workflow

In this section, we discuss the general workflow of the proof-of-concept implementation of our approach.

A user first needs to specify a function of interest or a starting address in the binary at which secret inputs or registers are defined as symbolic values. At the stage ❶ in Figure 5, the user wants to analyze an ARM binary starting from the `x103a8` address. This location starts with a `sbfx` (Signed Bit Field Extract) instruction which copies adjacent bits from one register into the least significant bits of a second register, and sign extends to 32 bits. Register `r0`, which holds the first operand, is marked as *sensitive* and is defined with a symbolic bit-vector in $\mathbb{F}_{16}$.

Binary analysis tools typically consists of a disassembler and a lifter to disassemble a given binary and translate it into what is known as an Intermediate Representation (IR). In Pascal, we preferred to use `angr` binary analysis framework (see stage ❸ in Figure 5) which operates on the lifted VEX IR, and also provides a convenient API to perform symbolic execution, expression annotation, and static analyses such as *loop finder*. In Figure 5, the `sbfx` instruction (stage ❶) is lifted into the corresponding set of VEX statements in an `irsb` block by PyVEX [98] (stage ❸). The VEX is structured into instruction superblocks (`irsb`). Each `irsb` contains a list of statements which may modify the program state and ends with an exit statement describing the succeeding `irsb` location. The most common statements are `WrTmp`, `Put`, and `St` to write temporary variables, registers, and memory respectively. Statements use expressions such as `RdTmp`, `Get`, and `Ld` to read temporary variables, registers, and memory [90]. Unary and binary operations such as `Shr`, `And`, and `Xor` have variants based on different bit-widths. Although, the focus of our research is on analysis of constant-time cryptographic implementations, in our tooling workflow, our method first checks any violations of constant-time practices using `ctgrind` tool developed by [59] (stage ❷) and Pascal rejects to continue if any violations are found.

Pascal uses forward symbolic execution [92] to find min/max Hamming weight differences and approximate entropy of registers in tandem with satisfiability modulo theory and optimization solvers [21], [36], and employs dynamic taint tracking over bit-vector expressions using a dynamic annotation propagation mechanism on bitvector expressions (provided by `angr`'s `claripy` module). For each instruction that the symbolic execution engine steps over, Pascal identifies secret-tainted, logical and arithmetic instructions. For each of them, it lazily extracts the destination register's symbolic bit-vector expression from the program's symbolic store (cf. §4.2). Pascal then creates a self-composed version of the expression by introducing a fresh variable for each symbolic variable (cf. §4.2). In this example, `r2′` is introduced after the stage ❸. Pascal later creates two optimization queries over the conjunction of those two bit-vector expressions, with the objective of maximizing and minimizing differential Hamming weights or Hamming Distance, and analyzed them over SMT-based *single-objective, linear optimization* algorithms (after stage ❹). Pascal records the results of the optimization queries and SMT queries as annotations at the respective address locations in the binary using a popular reverse engineering framework, `Radare2` [3]. Additionally, if Pascal detects a *determiner* (cf. Definition 7) then it adds a vulnerability flag to the binary (after stage ❺) at the respective location as an indication of *Point of Interest*. It also returns a function to generate a pair of *Test Vectors* to be used as an input for specific *t-test* (TVLA). For the case of symbolic register analysis (stages ❻ and ❼), we calculate the approximate entropy using ω-class sampling method (see §3.4 and §23). We developed three SMT-based techniques, parallel, pseudo-boolean equalities, and incremental using `z3` and `CVC5` solvers for bit-vector arithmetic.

In this example, `sbfx r2, r0, #0xf, #1` extracts the 16th bit position from register r_0, and store it in register r_2 by sign extending to 32 bits. Consequently, r_2 can only take two values, 0 or -1 (= $0xFFF\ldots F$). Therefore, the

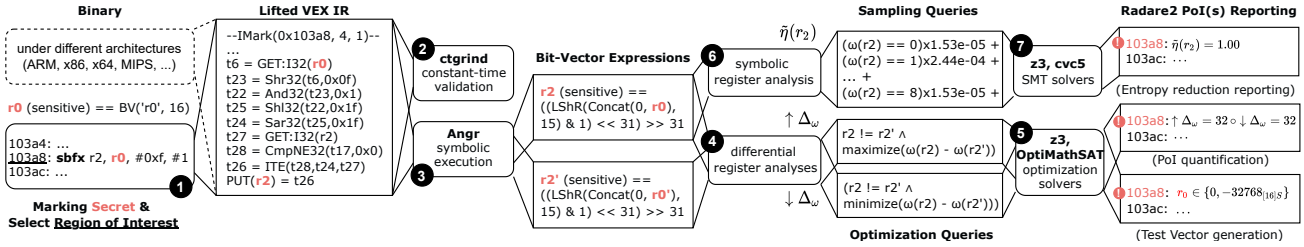

Figure 5: Pascal's simplified tooling workflow under *differential Hamming weight* and *ω-class sampling* models.

power side-channel analysis at address `x103a8` results in $\uparrow \Delta_\omega = 32$ and $\downarrow \Delta_\omega = 32$ with $\tilde{\eta} = 1.00$, and critical intermediate values for r_0 are found as $-32768_{[16]S}$ and 0.

4.2. Register Analysis Techniques

The principle of Pascal's verification scheme depends on analyzing each register that is tainted by the secret input(s) at every point of execution of the code.

Pascal uses *forward symbolic execution* and *dynamic taint analysis* to perform *symbolic register analysis* at binary level: dynamic taint analysis runs a program and observes which computations are affected by predefined taint sources such as secret function inputs; and forward symbolic execution automatically builds a logical formula describing registers at each state of the execution, which reduces the problem of reasoning about the execution to the domain of bit-vector logic. The two analyses can be used in conjunction to build formulas representing only the parts of an execution that depend upon tainted values [92].

Another purpose of using these techniques together is to quickly identify potential Point-of-Interest (PoI) candidates within the code, and to determine which functions have these candidates. The dynamic taint analysis is used to narrow down the number of potential PoIs without invoking a solver, by identifying which registers are tainted by the secret input. However, dynamic analysis alone is not sufficient to prove the existence of vulnerabilities, as it is not able to show that there are always only two classes of data whose Hamming weight difference is distinguishable. This is where symbolic execution comes into play, by providing symbolic representations of destination registers, extracted from the symbolic states of the computation, to further verify the existence of vulnerabilities.

Symbolic execution operates on symbolic values that represent any possible concrete value. Some or all variables (or in the context of binary analysis, registers or memory locations) are represented by a symbol that stands in for any possible value the variable could take. As the execution proceeds, symbolic execution computes logical formulas over these symbols. These formulas represent the operations performed on the symbols during execution and describe limits for the range of values the symbols can represent [6].

The `angr`'s symbolic execution engine computes two different kinds of formulas over these symbolic values: a set of symbolic expressions and a path constraint. A *symbolic state* S_n consists of the path constraint and a *symbolic store* σ_n that maintains a mapping of all registers, temporaries, and memory locations to symbolic expressions.

state	lifted binary code	symbolic store σ
S_0	$f(sum_{0[8]}, x_{0[8]})$	$\sigma_0 := \{x_0 = \beta_{[8]} \wedge sum_0 = \lambda_{[8]}\}$
S_1	$r_0 := x_0 - 64$	$\sigma_1 := \{\sigma_0 \wedge r_0 = x_0 - 64\}$
S_2	$r_1 := r_0 \gg 7$	$\sigma_2 := \{\sigma_1 \wedge r_1 = f_{asr}(r_0, 7)\}$
S_3	$r_2 := \sim r_1$	$\sigma_3 := \{\sigma_2 \wedge r_2 = \sim r_1\}$
S_4	$r_3 := r_2 \wedge x_0$	$\sigma_4 := \{\sigma_3 \wedge r_3 = r_2 \wedge x_0\}$
S_5	$r_4 := sum_0 + r_3$	$\sigma_5 := \{\sigma_4 \wedge r_4 = sum_0 \wedge r_3\}$

Figure 6: Symbolic execution of the example code.

In Pascal, the user introduces taints by annotating the initial symbolic bit-vector values (registers or function parameters) as *sensitive*; the rest of registers, temporaries, and memory locations are initialized as untainted. In Figure 6, symbolic bit-vector $\beta_{[8]}$ denotes the secret (high) parameter x_0, and symbolic bit-vector $\lambda_{[8]}$ denotes the untainted (low) parameter $sum_{0[8]}$. Taint propagation is performed dynamically over symbolic bit-vector expressions by relocating taint annotations from one symbolic expression to another while symbolic execution progresses to construct the symbolic store.

As an illustration, consider the program in Figure 6, for the sake of clarity its static single assignment form will be discussed in this section as a working example instead of a target specific machine code. Our analysis is being performed on disassembled binaries where intermediate values are represented by *registers* and *temporary* locations based on the target instruction set architecture. 'Closed quantifier-free formulas over the theory of bit-vectors' (QF_BV) [13] is used to represent symbolic expression and formulas. In Figure 6, first, Pascal assigns symbolic values (unconstrained bit-vector values) to β and λ to initiate a *call state* and uses standard forward symbolic execution to update the symbolic store until the first secret-dependent critical instructions is met at state S_2. Just after the symbolic store updates destination register r_1, Pascal queries the symbolic store for the value of the temporary r_1 to fetch the corresponding symbolic bit-vector expression and apply variable elimination and simplification: $\sigma_2[r_1] \mapsto (\beta_{[8]S} - 64) \gg 7$. At this point the bit-vector expression can be discharged to an `SMT` solver to get a satisfying model for symbolic value r_1: $\exists x_{[8]}.bv_{asr}(bv_{sub}(x_{[8]}, 64), 7)$. However, Pascal does not create a solver instance, just caches the query and its address location in the binary at this stage.

Differential Symbolic Register Analysis. Differential analysis aims to find different behaviors in programs or verify k-safety (i.e., properties that concern interactions between k program runs) [72]. Pascal needs to reason

about pairs of execution traces (2-safety) to identify different power side-channel behavior for different inputs on the same program; therefore, we are required to model two execution traces in the same symbolic execution instance.

In principle, 2-safety properties can be reduced to standard safety properties of a self-composed program [106]. Similarly, symbolic execution can be adapted to the case of constant-time code following the self-composition principle. Instead of self-composing the program, we rather self-compose the formula with a fresh version of its symbolic variables plus a precondition stating that the low inputs are equal:

$$\lambda = \lambda' \wedge \begin{pmatrix} x_0 = \beta \wedge sum_0 = \lambda \wedge r_0 = x_0 - 64 \wedge r_1 = r_0 \gg 7 \wedge \\ x_0' = \beta' \wedge sum_0' = \lambda' \wedge r_0' = x_0' - 64 \wedge r_1' = r_0' \gg 7 \end{pmatrix}$$

However, this can be achieved in a lazy manner by querying the symbolic store to get the symbolic expression of r_1 and then self-compose it with a version of itself (r_1') in which all symbolic values are fresh. Since we aim to find a data-oriented difference in a pair of executions, we assume r_1 and r_1' are different. In this way, the side-channel formula $\varphi_{SC_{S_2}}$ in state S_1 for the destination register r_1 can be then simplified to the following:

$$\varphi_{SC_{S_2}} \triangleq \overbrace{r_1 \neq r_1'}^{\text{disjoint 2-secrets}} \wedge \underbrace{r_1 = (\beta - 64) \gg 7}_{\text{symbolic register } r_1} \wedge \overbrace{r_1' = (\beta' - 64) \gg 7}^{\text{self-composition of } r_1}$$

Once the symbolic execution reaches at state S_4, a tainted *and* instruction is also detected and Pascal caches this symbolic expression of m_3 and self-compose it as well.

In literature, there is yet no definitive way of maintaining *constant power-consumption* in software implementations similar to that of *constant-time* due to the data dependent power consumption of digital circuits (see §3.1). Therefore, it is not possible to reduce this problem to a *decision problem*. Nevertheless, we can still use differential program analysis to model the power side-channel vulnerabilities by introducing an ω constraint to check if there are two intermediate values r_1 and r_1' whose differential Hamming weight is greater than a certain threshold (e.g., $|\omega(r_1) - \omega(r_1')| > v_{asr}$ for `asr` instruction). However, this query would be only useful for an attacker to craft low inputs to gain an observable difference in power traces, and can be used in *chosen-ciphertext* attacks, but it is not useful to find a definitive weak point such as a *determiner* (see Definition 7) in the binary and does not explain much in terms of quantification of power side-channel vulnerability. Therefore, we introduce minimum and maximum differential Hamming weight (see Definition 5) inspired from differential power analysis (see §3.3) to pinpoint the large *Hamming weight swings* happened among a constrained set of values.

Since those binary locations where small number of large Hamming weight swings happen are likely to be the locations where attackers can easily gain observable differences in power traces, we introduce a measure to quantify the power side-channel vulnerabilities using differential Hamming weight (see Definition 5), differential Hamming distance (see Definition 2) and

	machine code		$\tilde{\eta}(entropy)$	$\Delta_\omega(weight)$	$d(distance)$
S_1	$r_0 := x_0 - 64$		$\tilde{\eta}(r_0) = 2.54$	$\Delta_{\uparrow\downarrow\Delta_\omega} = 8$	$\Delta_{\uparrow\downarrow\ d} = 7$
S_2	$r_1 := r_0 \gg 7$	★	$\tilde{\eta}(r_1) = 1.00$	$\Delta_{\uparrow\downarrow\Delta_\omega} = 0$	$\Delta_{\uparrow\downarrow\ d} = 0$
S_3	$r_2 := \sim r_1$	★	$\tilde{\eta}(r_2) = 1.00$	$\Delta_{\uparrow\downarrow\Delta_\omega} = 0$	$\Delta_{\uparrow\downarrow\ d} = 0$
S_4	$r_3 := r_2 \wedge x_0$		$\tilde{\eta}(r_3) = 2.54$	$\Delta_{\uparrow\downarrow\Delta_\omega} = 8$	$\Delta_{\uparrow\downarrow\ d} = 7$
S_5	$r_4 := sum_0 + r_3$		$\tilde{\eta}(r_4) = 2.54$	$\Delta_{\uparrow\downarrow\Delta_\omega} = 8$	$\Delta_{\uparrow\downarrow\ d} = 7$

Figure 7: Three different leakage analysis results —Point-of-Interests are marked with stars.

Hamming weight class sampling (see Definition 8). If the maximum Δ_ω is equal to minimum Δ_ω or their difference ($|\uparrow \Delta_\omega - \downarrow \Delta_\omega|$) is significantly low considering the bitwidth of the intermediate values, then the implementation can be considered as not robust against power side-channel attacks. Because an attacker can easily generate a few templates for respective ω classes of those intermediate values (e.g. Table 1 for $\mathbb{F}_8$).

To use in bit-vector expressions, we introduce a Hamming weight function ω in the theory of bit-vectors in the `SMT` solver as $\omega : \mathbb{F}_n \mapsto \mathbb{F}_m$ where n is the width of the input vector and $m = \lfloor \log_2 n \rfloor$ is the width of its Hamming weight. SMT solvers perform better at bit-vector arithmetic in smaller bit-widths due to bit-blasting, therefore, we aim here to have a smaller bit-width for ω function. The following shows the objective functions for the optimization problems constructed by Pascal at state S_2 (other states are similar).

state	Δ_ω objective function	optimization query
S_2	maximize $\Delta_\omega(r_1, r_1')$	$\varphi_{SC_{S_2}} \wedge max(\lvert\omega(r_1) - \omega(r_1')\rvert)$
	minimize $\Delta_\omega(r_1, r_1')$	$\varphi_{SC_{S_2}} \wedge min(\lvert\omega(r_1) - \omega(r_1')\rvert)$

After discharging those queries into the optimization solver, we can see the quantification results in Figure 7. At state S_2, the difference of differential Hamming weight is minimized, and at state S_4 difference of differential Hamming weight is maximized. The former is marked as vulnerable, and the latter is not. In addition, a pair of witness is generated for the vulnerable state.

After the first vulnerability is identified at state S_2, the second vulnerability would have been also identified at state S_3. However, Pascal does not trigger another query at this point, instead it uses the S_2's solver instance to evaluate the min/max Δ_ω at S_3. The reason is the first vulnerability that happens at state S_2 affects the state S_3 since the symbolic expression that represents r_2 is a logical implication of r_1. We call this as *leakage continuity effect*, and it occurs when consecutive instructions continue operating on the reduced domain, which leads to more observable signal in the power trace.

Since optimization queries are more expensive than standard SMT queries and Pascal also aims at providing solutions for critical intermediate values (at least two *witnesses*) to generate Test Vector at each PoI for further TVLA analysis, we use Algorithm 1 to reduce the number of queries once dynamic taint propagation encounters an instruction whose VEX interpretation uses one of the critical instructions in a symbolic state.

In Algorithm 1, r is the bit-vector expression of the register (or temporary) under analysis and r' is its self-composed version. The algorithm first initializes the *objective* function at Line 1, sets minimization objective

Algorithm 1: Simplified version of differential symbolic register analysis at stage ❹ in Figure 5

Ensure: for each secret-tainted VEX instruction
Data: r, r': bit-vector expressions $\in \mathbb{F}_n$; *solver*: solver *context*
Result: $(\uparrow \Delta_\omega, \downarrow \Delta_\omega)$; ($w_1$: witness$_1$, w_2: witness$_2$)

```
1  (↑ Δω, ↓ Δω) ← (none, none);
2  objective ← | ω(r) − ω(r′) |;
3  solver.push(solver.minimize(objective));
4  if solve(solver) = SAT then ↓ Δω ← solver.eval(objective);
6  solver.pop();
7  if ↓ Δω = n then      /* when ↓ Δω equals to width of r */
8  |  ↑ Δω ← ↓ Δω; flag(vulnerable);
9  else
10 |  (w1, w2) ← (none, none);
11 |  solver.push(solver.maximize(objective));
12 |  if solve(solver) = SAT then
13 |  |  ↑ Δω ← solver.eval (objective);
14 |  |  (w1, w2) ← solver.eval(r, r′);
15 |  solver.pop();
16 |  if ↓ Δω = i ∧ ↑ Δω = i, where i is an ω-class then
17 |  |  solver.push(solver.add(r[n] ⊕ r′[n]));
18 |  |  if solve(solver) = UNSAT then flag(vulnerable);
19 |  |  solver.pop();
20 |  |  if w1 ≠ none ∧ w2 ≠ none ∧ ω(w1) = ω(w2) then
21 |  |  |  solver.push(solver.add(r ≠ w1 ∧ r′ ≠ w2));
22 |  |  |  if solve(solver) = UNSAT then flag(vulnerable);
23 |  |  |  solver.pop();
```

at Line 2, and finds the maximum $\Delta_\omega(r, r')$ at Line 4. Function `push` and `pop` are to retain solver instance's internal state in successive queries. `model` checks the satisfiability of a (optimal) solution, and if it is satisfiable (`SAT`), it returns a *finite model*. Function `eval` evaluates the given expression based on the *model* found. If the $\downarrow \Delta_\omega$ equals the maximum Hamming weight difference possible considering the bit-width of the expression m (Line 6) then there is no need to find $\uparrow \Delta_\omega$ since they are both equal; otherwise, it discharges maximization objective at Line 10. Function `flag` marks the address of the instruction under analysis as vulnerable in the binary. At line 13, solver assigns solutions to the two witnesses. We call the constraint constructed at line 16 as *discriminant*, and it is to enforce values of two witnesses at bit position n to be disjoint (1 or 0). It is used to check if there are other solutions when $\uparrow \Delta_\omega$ equals to $\downarrow \Delta_\omega$ at a Hamming weight class n at line 17. If the result is unsatisfiable (`UNSAT`) then there are no other solutions in this ω class, and the instruction address is flagged as vulnerable. The case analysis at line 19 is to capture edge scenarios such as a mask is crafted on purpose to have the same Hamming weight (e.g. $w_1 = 0x5A3C$ and $w_2 = 0xA5C3$) but having maximum *Hamming distance* to make it difficult for an attacker to change one valid value to a different valid value through *fault-injection* attacks. It is a defense used by smart card industry [119]. However, this case hasn't showed up yet in our analysis of PQC candidates.

Entropy-based Symbolic Register Analysis. *Hamming weight-class sampling model* calculates the approximate entropy at destination registers for each secret-tainted arithmetic and logical instruction under analysis. It is a good indicator of Point of Interests in *single-trace side-channel attacks*. However, it is computationally expensive and therefore is used as an alternative model. In Algorithm 2, r is the symbolic bitvector expression, for

Algorithm 2: Simplified version of symbolic register analysis for *ω-class sampling model* at stage ❻ in Figure 5

Ensure: for each secret-tainted VEX instruction
Data: r: bit-vector expression $\in \mathbb{F}_n$; *solver*: solver *context*
Result: $\tilde{\eta}$ of r

```
1  η̃ ← none; distribution ← ∅;
2  for i ← 0 to n do          /* for each ωi in ω-classes */
3  |  solver.push(solver.add(ω(r) = i));
4  |  if solve(solver) = SAT then      /* sample ω-class */
5  |  |  distribution.add(ℙωi);
6  |  solver.pop();
7  |  solver.add(ω(r) ≠ i) ;            /* block formula */
8  normalize(distribution) ; /* may resample at most twice */
9  η̃ ← entropy(distribution) ;        /* see Definition 8 */
10 if η̃ ≤ 1.00 then flag(vulnerable);
```

instance, for r_1 in Figure 6 it is $\sigma_2[r_1] \mapsto (\beta_{[8]S} - 64) \gg 7$. Based on Definition 8, `normalize` normalizes the discrete distribution of given probability values; additionally, if there is only one value in the `distribution`, the `normalize` function takes two more samples in that class to check if the entropy should be normalized to 1.00 or not. Finally, function `entropy` calculates the entropy of a distribution for given probability values (`distribution`). We added *blocking constraints* at Line 7 to increase the performance of the solver.

4.3. Limitations

We currently individually analyze functions or regions. Pascal's API allows for performing an analysis between given start and end addresses in a given binary file thus any code between start and end address can be analyzed, this will typically be the body of a function, including loops, branches, etc. However, due to the computational complexity of solving symbolic optimization queries for large operations, we generally divide the analysis into sub parts and analyze each sub part (usually functions).

`Angr` framework provides the capability to replace the actual code with function summaries. This feature was primarily used to skip analysis on certain portions of the code where we can be certain that there are no leakages. However, it is important to note that our approach does not solely rely on function summaries and they were only used sparingly in our empirical evaluation. A well-known drawback of using function summaries is that it requires careful verification of the correctness of the written summaries, as it may result in false negatives if not properly checked.

Pascal performs bounded analysis of loops. Specific loops can be chosen, and loop count can be set by the user through `angr`'s `LoopSeer` symbolic execution exploration technique.

5. Evaluation

Our evaluation is focused on answering the following research questions: **RQ1**: Is the proposed approach able to identify existing vulnerabilities? **RQ2**: Is the proposed approach able to detect new vulnerabilities? **RQ3**: What is the performance of Pascal's register analysis methods under different algorithms? **RQ4**: What is the precision of Pascal in analyzing protected implementations with

randomization and shuffling? **RQ5**: Show Pascal works in a variety of microarchitectures.

Detection of Known Vulnerabilities. We performed a literature review of recent power side-channel attacks against publicly available constant-time implementations of post-quantum cryptographic algorithms submitted to NIST's post-quantum cryptography standardization process, mbedTLS, and the NIST lightweight crypto competition. In the literature we found 16 known power-side channel vulnerabilities in constant-time implementations in a variety of cryptographic schemes, from Elliptic Curve [25] and Lattice-based [70] to Lightweight [113] cryptography. Our method successfully identifies all of them. We present the results in Table 6 in Appendix A and include some of them in our benchmarks (see Table 2).

In our review, we exclude some attack papers that fall under the following categories: (1) classic attacks to block ciphers such as AES's sbox [11], [18]. One of the reason is that employing table look-ups indexed by secret data is not a recommended constant-time practice and in those attacks although Hamming weight or Hamming distance leakage models are being used, the attacker needs to profile all the Hamming weight classes of intermediate values from Galois Field, $\mathbb{GF}(2^8)$, to the sbox function; (2) timing side-channel attacks using power traces such as [43], [79] since there are many formal constant-time analysis tools that is able to detect secret-dependent branching; (3) primitives protected with perfect masking such as order-d secret-sharing masking scheme [81] considering the fact that the vulnerability is already eliminated by the masking technique.

The experiments are conducted on ARM Cortex-M4 since popular PQC projects such as pqm4 [55] chooses this target. Pascal was able to detect all known vulnerabilities listed in Table 2. All vulnerabilities are experimentally shown to be exploitable in the literature and are successfully detected and quantified by Pascal. Here, we discuss three power side-channel attacks against a lightweight crypto [123] and a post-quantum crypto [104], [109] implementations. More examples have been shown in Appendix A.

SPECK [14] and SPARX [40] are lightweight block ciphers that use add-rotate-xor (ARX) constructions. Other examples for ARX constructions include stream cipher Chacha20 [60] and the SKEIN [45] hash function. [123] examines the intuition that ARX ciphers [40] have intrinsic resilience against side channel attacks because of the absence of strong S-Boxes. They performed a correlation power analysis attack to recover the secret. Consider the function A in Listing 1. When compiled with ARM toolchain with -O3 and cortex-m4 options, it is translated into the assembly code shown in Listing 2.

```
1 // Rotate left for 16 bit registers.
2 #define ROTL(x, n) (((x) << n) | ((x) >> (16-(n))))
3 // Rotation and Addition.
4 void A(uint16_t* l, uint16_t* r) {
5   (*l) = ROTL((*l), 9);
6   (*l) += (*r);
7   (*r) = ROTL((*r), 2);
8   (*r) ^= (*l);
9 }
```

Listing 1: SPECK's ARX-box Implementation

Pascal detects maximum $\Delta_\omega = 2$ and minimum $\Delta_\omega = 0$. Over the set of 2^{16} number, the register r3 at line 13 can take only one of $\{0, 1, 2, 3\}$ and therefore this reduces the number of traces required for the attack.

```
1  ldrh r2, [r0]            ; arg1;
2  lsrs r3, r2, 7           ; ↑Δω = 9 ∘ ↓Δω = 0
3  orr.w r3, r3, r2, lsl 9  ; ↑Δω = 25 ∘ ↓Δω = 0
4  uxth r3, r3              ; ↑Δω = 16 ∘ ↓Δω = 0
5  strh r3, [r0]            ; arg1
6  ldrh r2, [r1]            ; arg2
7  add r3, r2
8  strh r3, [r0]            ; arg1
9  ldrh r2, [r1]            ; arg2
10 lsrs r3, r2, 0xe         ; ↑Δω = 2 ∘ ↓Δω = 0
11 orr.w r3, r3, r2, lsl 2  ; ↑Δω = 18 ∘ ↓Δω = 0
12 uxth r3, r3              ; ↑Δω = 18 ∘ ↓Δω = 0
13 strh r3, [r1]            ; arg2
14 ldrh r2, [r0]            ; arg1
15 eors r3, r2              ; ↑Δω = 18 ∘ ↓Δω = 0
16 strh r3, [r1]            ; arg2
17 bx lr
```

Listing 2: Full disassembly of Listing 1

NewHope [2] is a key encapsulation method proposed in the NIST post-quantum project. In Listing 3, a straightforward implementation might use a for-loop over all message bits containing an if-condition which sets the polynomial coefficients to either 0 or $q/2$. Since such an implementation would be susceptible to timing attacks, the message encoding is implemented in a way that the code inside the for-loop always runs in constant time.

```
1 void poly_frommsg(poly *r, unsigned char *msg){
2  unsigned int i, j, mask;
3  for (i = 0; i < 32; i++) {
4    for (j = 0; j < 8; j++) {
5      mask = -((msg[i] >> j) & 1);
6      r->coeffs[8*i+j+0] = mask & (NEWHOPE_Q/2);
7      r->coeffs[8*i+j+256] = mask & (NEWHOPE_Q/2);
8      ...
```

Listing 3: Attack to NewHope's message encoding

A mask, containing 0 or -1 (= $0xFFFF\ldots$), replaces the if-condition. The mask calculation is shown in Listing 3 at line 5. However, power consumption might differ between processing a logical zero or logical one, especially because the mask either contains ones or zeroes only. Chances that processed values can be detected by analyzing the power consumption of the device are high.

Based on power measurement, [4] extracts the complete shared secret from one single trace only. The impact of different compiler directives are additionally investigated: when the code is compiled with optimization turned off (-O0 shown in Listing 4), the shared secret can be read from an oscilloscope display directly with the naked eye; when optimizations are enabled (-O3 shown in Listing 5), the attack requires template-based attack, but the attack still works on single power traces.

```
1 ldr   r2, [r7,#0]   ;r2 = memory[r7]
2 ldr   r3, [r7,#20]  ;r3 = memory[r7+20]
3 add   r3, r2        ;↑Δω = 32 ∘ ↓Δω = 0  ‖ η̃ = 3.53
4 ldrb  r3, [r3,#0]   ;r3 = memory[r3]
5 mov   r2, r3        ;r2 = r3
6 ldr   r3, [r7,#16]  ;r3 = memory[r3+16]
7 asr.w r3, r2        ;↑Δω = 32 ∘ ↓Δω = 32 ‖ η̃ = 1.00  ★
8 and.w r3, r3, #1    ;↑Δω = 1  ∘ ↓Δω = 1  ‖ η̃ = 1.00  ★
9 negs  r3, r3        ;↑Δω = 1  ∘ ↓Δω = 1  ‖ η̃ = 1.00  ★
```

Listing 4: Partial disassembly at -O0 of Listing 3

TABLE 2: Known and New Vulnerabilities. All known attacks in literature levereges Hamming weight leakage model.

	Crypto Algorithm	Crypto Type	Crypto Function	Operation Description	Implementation	Attack	Leakage Model(s)
P_1	NTRU	Lattice-based	`mod3`	Modular reduction	NIST PQC Round3	[8]	ω
P_2	NTRU	Lattice-based	`mod3_alt`	Alternative modular reduction	NIST PQC Round3	[8]	ω
P_3	NTRU	Lattice-based	`poly_Z3_to_Zq`	Map from $\mathbb{Z}_3$ to $\mathbb{Z}_q$	NIST PQC Round3	[104]	ω
P_4	NTRU	Lattice-based	`int32_MINMAX`	Sorting/Comparison	NIST PQC Round3	[56]	ω
P_5	NTRU	Lattice-based	`int32_MINMAX'`	Sorting/Comparison (inline assembly)	pqm4 [55] library	—	ω
P_6	Kyber	Lattice-based	`poly_frommsg`	Message Encoding (NIST standard)	NIST PQC Round3	[84], [109]	ω
P_7	Kyber	Lattice-based	`poly_frommsg`^{1}	Message encoding with multiplication	NIST PQC Round3	[4]	ω
P_8	Kyber	Lattice-based	`poly_frommsg`^{2}	Data independent poly. generation	NIST PQC Round3	[109]	ω
P_9	Kyber	Lattice-based	`poly_frommsg`^{3}	Balanced data independent poly. gen.	NIST PQC Round3	[109]	ω
P_{10}	Kyber	Lattice-based	`poly_frommsg`^{4}	Polynomial randomization	NIST PQC Round3	[109]	ω
P_{11}	Kyber	Lattice-based	`poly_frommsg`^{5}	Byte and bit level random ordering	NIST PQC Round3	[109]	ω
P_{12}	Kyber	Lattice-based	`poly_frommsg'`	Alternative Message Encoding	pqm4 [55] library	[104], [122]	ω
P_{13}	Kyber	Lattice-based	`poly_tomsg`	Convert polynomial to 32-byte message	NIST PQC Round2	[85]	ω
P_{14}	Kyber	Lattice-based	`mont_reduce`	Montgomery reduction	NIST PQC Round3	—	ω, d
P_{15}	Kyber	Lattice-based	`poly_csubq`	csubq of each coefficient a polynomial	NIST PQC Round3	New	ω
P_{16}	NewHope	Lattice-based	`poly_tomsg`	Convert polynomial to 32-byte message	NIST PQC Round2	[85]	ω
P_{17}	NewHope	Lattice-based	`poly_tobytes`	Serialization of a polynomial	NIST PQC Round2	New	ω, d
P_{18}	NewHope	Lattice-based	`poly_from_msg`	Convert 32-byte message to polynomial	NIST PQC Round1	[4], [84]	ω
P_{19}	FrodoKEM	Lattice-based	`key_decode`	Message Decoding	NIST PQC Round2	[85]	ω
P_{20}	FrodoKEM	Lattice-based	`key_encode`	Message Encoding	NIST PQC Round3	[104]	ω
P_{21}	mbedTLS	Elliptic Curve	`ct_mpi_uint_lt`	Constant-time less-than comparison	mbedTLS v3.1.0	New	ω, d
P_{22}	mbedTLS	Elliptic Curve	`mpi_lt_mpi_ct`	Constant-time, signed comparison	mbedTLS v3.1.0	New	ω, d
P_{23}	Sparx	Lightweight	`sparx_encrypt`	Sparx's ARX-box Assembly	NSA Reference	[123]	ω
P_{24}	Sparkle	Lightweight	`ARX`	Sparkle's ARX-box Assembly	NIST LWC Finalist	—	ω, d

P_7, P_8, P_9, P_{10}, and P_{11} are protected implementations of CRYSTALS-Kyber's vulnureable message encoding function (P_6).

In Listing 5 at line 2, Arm's *signed bit field extract* instruction (`sbfx`) is generated by the compiler for extraction of zero bit (1 bit) of $r2$ and sign-extend it to 32 bits (if $bit_0(r2) == 0$, then $r2 = 0x0000\cdots$, else $r2 = 0xFFFF\cdots$). This instruction's semantics is also similar to right shifting, and actually the VEX IR lifter uses an `Asr32` instruction in the intermediate representation.

```
1 ldrb   r2, [r3,#0]     ;r2 = memory[r3]
2 sbfx   r2, r2, #0, #1  ;↑Δω = 32 ∘ ↓Δω = 32 ‖ η̃ = 1.00 ★
3 strh   r2, r2, #6144   ;r2 = r2 & 6144
4 and.w  r2, r2, [r0,#0];↑Δω = 32 ∘ ↓Δω = 0  ‖ η̃ = 3.53
5 strh.w r2, [r0,#512]   ;memory[r0 + 512]  = r2
6 strh.w r2, [r0,#1024]  ;memory[r0 + 1024] = r2
7 strh.w r2, [r0,#1536]  ;memory[r0 + 1536] = r2
```

Listing 5: Partial disassembly at `-O3` of Listing 3

Detection of New Vulnerabilities. We want to highlight that the objective of Pascal is to find Points of Interest, i.e. potentially vulnerable instruction addresses, so developers can fix them. In our work, we used known vulnerabilities to show that Pascal can find real vulnerabilities. It can also detect new potential Points of Interest. For example, here we report some PoIs in the Post Quantum and Lightweight Crypto libraries as well as constant-time libraries of mbedTLS, which can be found in Table 2. There are Points of Interest in P_{15}, P_{17}, P_{21}, and P_{22}. They are all confirmed with specific t-tests (TVLA) by attacking those PoIs reported by Pascal and using test vectors that Pascal generated. We used ChipWhisperer UFO [73] with STM32F3 target board having 32-bit ARM Cortex-M4 processor core.

As an example, Listing 6 and Listing 7 shows the detected PoI in P_{21} of mbedTLS [112]. The value of `ret` at line 7 in C code is defined as a sensitive bit-dependent determiner. Its state corresponds to the register `r0` in the assembly at line 7. Power consumption traces can be classified into two sets depending on the Hamming weight of its value, as the number of cases is 2.

```
1 unsigned mbedtls_ct_mpi_uint_lt(const uint64_t x,
2                                 const uint64_t y) {
3     uint64_t ret, cond;
4     cond = (x ^ y);
5     ret = (x - y) & ~cond;
6     ret |= y & cond;
7     ret = ret >> (sizeof(uint64_t) * 8 - 1);
8     return (unsigned)ret;
9 }
```

Listing 6: P_{21}: Constant-flow LT comparison.

```
1 cmp r0, r2
2 sbc r2, r1, r3   ; ↑Δω = 32 ∘ ↓Δω = 0  ‖ η̃ = 3.547
3 eor r0, r1, r3   ; ↑Δω = 32 ∘ ↓Δω = 0  ‖ η̃ = 3.547
4 eor r3, r3, r2   ; ↑Δω = 32 ∘ ↓Δω = 0  ‖ η̃ = 3.547
5 and r0, r0, r3   ; ↑Δω = 32 ∘ ↓Δω = 0  ‖ η̃ = 3.547
6 eor r0, r0, r2   ; ↑Δω = 32 ∘ ↓Δω = 0  ‖ η̃ = 3.547
7 lsr r0, r0, 0x1f; ↑Δω = 1  ∘ ↓Δω = 0  ‖ η̃ = 0.195 ★
8 bx lr
```

Listing 7: Disassembly at `-O3` of Listing 6

Performance Evaluation. The evaluation is performed on 11th Gen Intel Core™ i7-1185G7 @ 3.00GHz x 8 cores with 32 GB RAM on Ubuntu 20.04.4 LTS OS. Pascal's implementation is sequential, however, in the future, we aim to discharge optimization queries concurrently. In our evaluation we evaluated 24 functions. If a benchmark has loops, we selectively unroll them 8 times. Analyzing benchmarks with 6 different methods take about 2.5 hours to run on the machine. Table 2 presents the results of the analysis. All times are given in seconds.

The SMT problems that require the capability of finding models that are optimal with regard to some objective functions are grouped under the umbrella term of Optimization Modulo Theories [93].

TABLE 3: Pascal's Performance Analysis on known and some new vulnerabilities.

	# of Inst.	Hamming weight ($\uparrow\downarrow \Delta_\omega$)‡				Hamming distance ($\uparrow\downarrow d$)‡				Approximate Entropy ($\tilde{\eta}$)*				Pascal
		basic	symba	obvbs	# PoI	basic	symba	obvbs	# PoI	par.	pb.	inc.	# PoI	
P_1	77	2.13	2.15	4.49	4	2.01	1.98	4.58	4	5.13	4.86	9.55	30	✓ TP
P_2	73	2.03	2.02	3.89	4	1.89	1.87	4.11	4	4.34	4.14	8.67	31	✓ TP
P_3	52	1.22	1.19	1.60	0	0.64	0.66	1.96	0	3.01	1.98	1.78	0	✓ FN
P_4	46	2.00	1.96	2.00	1	1.40	1.39	2.06	1	5.35	4.41	2.21	1	✓ TP
P_5	24	2.17	2.21	2.21	1	2.18	2.19	2.20	1	2.22	2.25	2.31	1	✓ TP
P_6	64	0.46	0.48	1.03	11	0.40	0.42	1.11	11	1.31	1.32	1.03	11	✓ TP
P_7	60	0.51	0.51	0.91	12	0.44	0.45	1.23	12	1.43	1.45	1.19	12	✓ TP
P_8	75	14.07	13.83	15.95	3	13.81	13.77	15.28	3	16.26	16.08	15.00	3	✓ TP
P_9	104	23.88	23.61	27.45	4	23.17	23.27	26.80	4	27.09	26.71	25.32	4	✓ TP
P_{10}	160	68.61	70.54	81.49	4	75.38	65.37	78.29	4	90.57	91.89	84.24	9	✓ TP
P_{11}	184	129.03	127.12	146.66	7	123.56	119.94	150.91	7	159.51	153.11	151.64	10	— FP
P_{12}	59	2.51	2.50	7.27	8	2.06	2.11	6.67	8	7.35	7.51	6.05	8	✓ TP
P_{13}	139	5.40	5.25	10.81	8	3.60	3.58	9.88	8	21.81	11.49	9.58	8	✓ TP
P_{14}	48	5.35	6.09	4.82	0	72.59^{TO}	71.79^{TO}	69.96^{TO}	0	48.13	5.81	7.45	0	— TN
P_{15}	63	3.83	3.75	8.04	8	2.61	2.53	7.63	8	6.38	8.31	7.04	8	✓ TP
P_{16}	149	369.77	382.49	139.70	8	372.65	391.06	127.10	8	600^{TO}	600^{TO}	314.5	15	✓ TP
P_{17}	223	5.16	5.56	11.03	8	3.78	3.64	10.17	8	21.48	11.26	10.47	8	✓ TP
P_{18}	67	24.77	24.48	48.76	16	22.72	22.77	43.89	16	54.52	59.14	45.44	16	✓ TP
P_{19}	110	161.23	158.77	165.96	3	163.04	160.45	172.02	3	177.01	177.32	170.36	8	✓ TP
P_{20}	98	34.10	34.80	55.36	4	33.41	31.63	56.15	4	67.59	68.90	162.02	14	✓ TP
P_{21}	66	0.54	0.65	0.64	1	0.32	0.29	0.55	1	1.02	0.80	0.77	1	✓ TP
P_{22}	31	0.47	0.52	0.48	1	0.21	0.22	0.50	1	0.87	0.66	0.53	4	✓ TP
P_{23}	31	1.64	1.64	2.26	0	1.23	1.26	2.65	0	4.68	2.69	2.22	3	✓ TP
P_{24}	80	12.09	11.44	11.93	0	10.68	10.33	11.86	0	29.25	19.44	16.38	0	— TN
		872.97	883.56	754.74		933.78	932.97	807.56		1,354.48	1,281.42	1,055.85		

All times are in seconds. ‡ `basic` [20] and `symba` [61] algorithms are part of `Z3` whereas `obvbs` is provided by `OptiMathSAT` [94]. * `par.`: parallel bitvector solving with a pool of 8 cores in `Z3`; `bp`: encoding ω-classes as Pseudo Boolean equalities in `Z3`; `inc.` incremental bitvector solving via `CVC5`. TO: Time-out. TP: True Positive. TN: True Negative. FP: False Positive. FN: False Negative. ✓: TVLA detects leakage (t-test ≥ 10). —: TVLA does not detect any leakage (t-test < 10).

In Pascal, differential Hamming weight and Hamming distance models specifically requires *single-objective linear optimization* over bitvector terms.

There are efficient SMT-based optimization algorithms in the theory of fixed-size bitvectors: *vZ* [21] and `Symba` [61] integrated with `Z3` SMT solver, and `OptiMathSAT` [94] that extends `MathSAT5` [33] SMT solver. We incrementally use *bit-vector optimization with binary search* (`obvbs`) [93] algorithm in `OptiMathSAT`. In *Z*3, we employ both `basic` [20] and `symba` [61] optimization methods that locally enumerate optimal assignments until fixed point. Those methods are able to find intermediate solution along the way. Therefore, in our experiments, Pascal retrieves the latest suboptimal solution in case it runs out of time. In our work, we compare those three algorithms and found that `OptiMathSAT`'s `obvbs` optimization algorithm outperforms `Z3`'s `basic` and `symba` algorithms over our benchmarks (see Figure 8). The analysis time varies based on the loop and the complexity of the symbolic expression that represents the destination registers and the number of discharges on solvers. For instance, in P_{16}, the analysis takes 382.49 seconds with Hamming weight model using `symba` while the instruction count is 149, whereas, in P_{17}, the analysis takes 5.56 seconds with the same method while the instruction count is 223.

With a soft timeout of 60 seconds given per SMT query, `Z3` and `OptiMathSAT` solvers run out of time and returns *suboptimal* solutions for a minimization objective of program P_{14} under Hamming distance model. We couldn't identify what makes this specific query hard for solvers.

In ω-class sampling method, we used `Z3` solver to solve bitvector queries in parallel mode with a pool of 8 cores (see `par.` in Table 2) and compare the performance with two other methods: encoding ω-class constraints as Pseudo-Boolean equalities in `Z3` (`pb.`) and incremental encoding using `CVC5` [12] (`inc.`). We also set full timeout for algorithms, which is 600 seconds per benchmark and method. `par.` and `pb.` timed out for benchmark P_{16} and thus `inc.` method on `CVC5` solver outperforms `par.` and `pb.` methods, however, if we remove this benchmark, method `pb.` outperforms others.

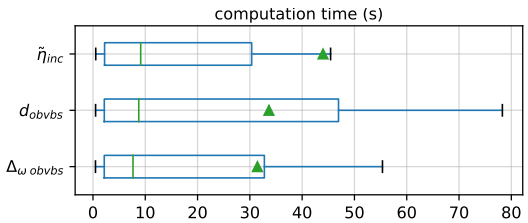

Figure 8: Distributions of computational times show that although median of those methods are close, in average (▲) Δ_ω analysis with `obvbs` performs slightly better.

Randomized and Shuffled Implementations. P_7, P_8, P_9, P_{10}, and P_{11} are different protected implementations of benchmark P_6 which is NIST's PQC Standardization winner CRYSTALS-Kyber's vulnerable message encoding function. The vulnerability that leads to the attack path is quite similar to that of New Hope's message encoding. The message *msg* is processed in bitwise manner during the message encoding as shown in Listing 8. The mask value can be either $0x0000$ or $0xFFFF$; therefore, the

number of cases of the mask value is 2 (Definition 7). Moreover, the differential Hamming weight is equal to $\Delta_\omega = 16$. [104] exercised single-trace attacks at this PoI. More recently [51] and [41] demonstrated attacks on its masked version [53].

```
void poly_frommsg(poly *r,
      const uint8_t msg[KYBER_INDCPA_MSGBYTES]) {
  unsigned int i, j;
  int16_t mask;
  for (i = 0; i < KYBER_N / 8; i++) {
   for (j = 0; j < 8; j++) {
     mask = -(int16_t)((msg[i] >> j) & 1);
     /*↑ Δω = 16 ∘ ↓ Δω = 16 ‖ ↑ Δd = 16 ∘ ↓ Δd = 16 ‖ η̃ = 1.00*/
     r->coeffs[8*i + j] = mask & ((KYBER_Q+1)/2);
    }
  }
}
```

Listing 8: P_6: CRYSTALS-Kyber's message encoding

TABLE 4: Pascal's precision of vulnerability detection for Shuffling and Randomization countermeasures [109] applied to NIST's PQC Standardization winner CRYSTALS-Kyber's message encoding function.

	CRYSTALS-Kyber Message Encoding converts 32-byte message to polynomial	t-test t_{max}	Attack Success	Pascal
P_6	Unprotected NIST Submission [84], [109]	437	100%	TP
P_7	Message encoding with multiplication [4]	177.0	100%	TP
P_8	Data independent polynomial gen. [109]	24.8	68.6%	TP
P_9	Balanced data independent poly. gen. [109]	19.6	67.9%	TP
P_{10}	Polynomial randomization [109]	13.8	64.0%	TP
P_{11}	Byte and bit level random ordering [109]	5.2	50.1%	FP

Table 4 shows Pascal's precision in comparison with t-test measurements and success rates of power analysis attacks based on the work of [109]. A t-test result below 10 indicates absence of a leakage. Individual shuffling and masking countermeasures were shown to be vulnerable against simpler power attacks, and Pascal successfully confirmed vulnerabilities in P_7, P_8, P_9 and P_{10}. A combination of masking and shuffling increases the trace requirement for the attack, and benchmark P_{11} shuffles message encoding using byte and bit level random ordering. If we compare it with the reference implementation, P_5, this countermeasure significantly reduces the success rate of attacks from 100% to 50.1% while introducing 2.49X overhead [109]. In this benchmark, Pascal marked an instruction address as vulnerable (False Positive). However, once the generated report is investigated, one can realize that the vulnerability manifests itself randomly at some loop iterations, and therefore, it can be corrected by the user as True Negative. P_{11} is detailed in Listing 9 and shows 'Data independent polynomial generation with balanced byte look-up' countermeasure proposed by [109] for Kyber's message encoding function (Listing 8). The strategy is to shift the pointer array `p_r` and the balancing array `xorMask` by 0 or 1, depending on the most significant bit of the first message byte (MSB). As the message `msg` is randomly chosen, evaluating the MSB serves as a source of randomness without introducing an additional fetch from a random number generator.

```
void masked_poly_frommsg(poly *r,
    const uint8_t msg[KYBER_INDCPA_MSGBYTES]) {
  unsigned int i, j;
  poly r_d;
  poly *p_r[256 + 1];
  uint32_t xorMasks[3] = {0xaaaaaaaa, 0x55555555,
    0xaaaaaaaa};
  for (i = 0; i < 256; i += 2) {
    p_r[i] = r;
    p_r[i + 1] = &r_d;
  }
  for (i = 0; i < KYBER_N; i++) {
    r->coeffs[i] = (KYBER_Q + 1)/2;
    r_d.coeffs[i] = (KYBER_Q + 1)/2;
  }
  uint32_t rand = (0xaaaa00aa ^ msg[0]);
  uint8_t i_m = rand & 0x1f;
  uint8_t j_m = (rand >> 5) & 0xff;
  uint32_t b_inv = (rand >> 7) & 0xff;
  for (i = 0; i < 255; i++) {
    *(p_r + i) = *(p_r + i + b_inv); }
  for (i = 0; i < 2; i++) {
    *(xorMasks + i) = *(xorMasks + i + b_inv);
  }
  uint8_t i_r, j_r;
  for (i = 0; i < KYBER_N/8; i++) {
    i_r = i ^ i_m;
    for (j = 0; j < 8; j++) {
      j_r = j ^ j_m;
      p_r[((xorMasks[(j_r & 1)]^msg[i_r]) >> j_r)
        & 0xff]->coeffs[8 * i + j] = 0; }
  }
}
```

Listing 9: P_{11}: Protected version of Listing 8.

We expect no False Negatives since if there is some Hamming weight difference or significant entropy loss, Pascal will detect such instructions and flag their addresses as Point of Interests. Pascal only reports a False Negative in `poly_Z3_to_Zq` (benchmark P_3) unless the precondition on input variable is given i.e. the input coefficients must be one of $\{0, 1, 2\}$.

In benchmarks P_5, P_{15}, and P_{24}, neither of the models detect any vulnerabilities and in fact we weren't able to find a significant t-test value above 10. Benchmark P_{23}, ARX box construction of SPARX [40] (see Listing 2), is vulnerable while benchmark P_{24}, ARX box of NIST's LWC finalist Sparkle [15] is secure; Pascal did not detect any PoI, and we did not detect any presence of power leakage via TVLA.

Target Architectures. We have confirmed the presence of those vulnerable PoIs listed in Table 3 using TVLA on ARM Cortex-M4 since it is widely used in the PQC community [55], [65]. We also separately confirmed the applicability of the Hamming weight leakage model to single-trace side-channels on different architectures: ARM Cortex-M4, ARM Cortex-M0, Atmel AVR XMEGA, and MSP430 using STM32F3/F4, STM32F0, ATXmega128D4-AU, and MSP430FR5994 targets respectively.

Any circuit not explicitly designed to be resistant to power attacks exhibits data-dependent power consumption phenomenon (cf. Section 3.1), and Hamming weight leakage model is a well-established method in the literature to model this behavior. Instruction-level information that is obtained from the disassembler retains enough symbolic information to track data-dependent Hamming-weight characteristics.

The Hamming weight leakage model has been used in recent power analysis attacks on modern Intel (and

AMD) x86 CPUs, as demonstrated in the Platypus [62] and Hertzbleed [117] attacks. In the Hertzbleed attack, the vulnerability on SIKE [10]'s *three point ladder* is investigated using the Hamming weight leakage model and it is also detected by Pascal with symbolical register analysis. Additionally, in the Platypus work, the performance counters of x86 are used and attacks are performed using the Hamming weight leakage model.

Discussions. We argue that our work highlights the need for single-trace side-channel aware constant-time cryptographic coding. The conflict between a constant-time implementations of cryptographic algorithms and its single-trace power or EM leakages manifested in hardware makes this a non-trivial task and should be investigated.

Experimenting with power side-channel attacks becomes affordable [74] thanks to special power side-channel analysis circuitry such as ChipWhisperer [73]. Additionally, protecting against power analysis attacks is less straightforward and usually more expensive than countermeasures for timing attacks. Simple countermeasures such as introducing jitter adds horizontal noise leading to non-alignment of PoI across measurements, and it increases the attack effort.

Our current technique is able to detect multi-trace vulnerabilities, as shown in the Listing 1 where the attacker needs to differentiate four different values. However, in this paper, we have primarily focused on single-trace attacks as they are the simplest form of attack for an attacker to perform on unprotected implementations.

6. Related Work

Pascal's approach to detection to power/EM side-channels vulnerabilities has a unique position in the literature since it is the first to devise a leakage detection technique using automated reasoning according to the SoK study conducted in [29] (see Table 5 for a summary). Pascal does not use any simulated, estimated, or measured trace and covers both power and EM side-channels; it does not require any microarchitectural information and therefore it is not tailored to any specific architecture, and it is publicly available. Furthermore, it approximately quantifies side-channel vulnerabilities. However, it is not the only method to detect power side-channel vulnerabilities. In this section, we discuss the most relevant tools in the literature.

The Test Vector Leakage Assessment (TVLA) [50] identifies differences between two sets of side channel measurements by computing the t-test for the two sets of measurements. It is being used in the literature to confirm the *presence* or *absence* of side leakages for power traces. TVLA does not pinpoint code locations, only statistically says if there may be some leakage. It also requires partitioning of the traces based on the value of a particular bit of an intermediate state in the targeted algorithm, and therefore to comprehensively evaluate an implementation, single bit of every single intermediate state must be tested [64], but this is impractical, the analysis are usually restricted to measure not more than one million traces [39]. We would like to clarify that the goal of Pascal is to automatically find potentially vulnerable instructions. Pascal is not a substitute for, or an alternative to, methods such as TVLA. Once a Point of Interest is found automatically, a developer can then investigate it, possibly using TVLA to confirm if this is indeed a vulnerability and fix it.

TABLE 5: Existing Power/EM Side Channel Analysis tools and methods, adapted from the SoK paper of [29]. Other than Pascal, all tools are leakage simulators.

Work	Year	LM‡	Target Device(s)	SC	Avail.‖
Pascal	2022	○	Any Target†	Power/EM	✓
Rosita++ [97]	2021	◑	ARM Cortex M0/M4	Power	✓
Emsim [95]	2020	◑	Risc-V	EM	✗
Rosita [96]	2019	◑	ARM Cortex M0	Power	✓
Elmo [66]	2017	◑	ARM Cortex M0	Power	✓
Ascold [76]	2017	◑	ATMega163	ILA	✓
Savrasca [115]	2017	●	ATMega163	Power	✓
[87]	2016	●	not relevant	Power	✗
Sleak [116]	2014	●	ARM Cortex A8	Register	✗
Silk [114]	2014	●	ATmega328P	Power	✓
[46]	2013	●	Risc-V	Power	✗
[37]	2012	◑	not specified	Power	✗
[111]	2009	●	AT90XX, ATmegaXX	Power	✗
[89]	2002	●	not relevant	Power	✗
Pinpas [38]	2003	●	smartcards	Power	✗

† Hamming weight Leakage assumption is verified on these targets: ATXmega128D4-AU, STM32F3/F4 (ARM Cortex-M4), STM32F0 (ARM Cortex-M0), MSP430FR5994. ‡ Leakage Model: ●: a black-box model (ISA level information needed); ◑: a gray-box model (microarchitectural information needed); and ○: a white-box model (formal instruction semantics needed). ‖: publicly available or not.

There are leakage emulators in the literature such as Elm0 [66], Rosita [96], and Rosita++ [97] for ARM Cortext-M0/M4, Ascold [76] for ATMega163 targets. These tools are able to simulate instruction-based power variations on specific target architectures based on profiling or reverse engineering a target device, and generate emulated traces. However, creating such a model is prohibitively effort-intensive, even for relatively simple processors (in-order, no cache) [29]. They are not able to formally guarantee their existence as Pascal does. On the other hand, those leakage simulators may identify power variations due to the underlying microarchitecture that might not be explained by Hamming weight or Hamming distance leakage models. However, all the attacks we investigated so far can be accounted for by Hamming weight models.

7. Conclusions

To help make the process of detecting potential power side-channel vulnerabilities easier for cryptographers, this work presented Pascal that introduces a number of novel symbolic register analysis techniques in binary analysis of constant-time cryptographic implementations, and pinpoints locations of potential power side-channel vulnerabilities with high precision. It also generates test vectors for TVLA analysis. Our tool was able to locate all currently reported single-trace power side-channel vulnerabilities in constant-time code of post-quantum cryptographic algorithms. The analysis of target functions can be done automatically with Pascal, and significantly reduces the burden on the programmer for finding potentially vulnerable code locations.

References

[1] Onur Aciicmez, Werner Schindler, and Cetin K Koç. Improving brumley and boneh timing attack on unprotected ssl implementations. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 139–146, 2005.

[2] Erdem Alkim, Roberto Avanzi, Joppe W Bos, Leo Ducas, Antonio de la Piedra, Thomas Poppelmann, Peter Schwabe, and Douglas Stebila. Newhope (version 1.1): Algorithm specifications and supporting documentation (april 10, 2020). Technical report, Submission to the NIST post-quantum project, 2020.

[3] Sergi Alvarez. Radare2: Libre and portable reverse engineering framework, 2021. https://github.com/radare/radare2.

[4] Dorian Amiet, Andreas Curiger, Lukas Leuenberger, and Paul Zbinden. Defeating newhope with a single trace. Cryptology ePrint Archive, Report 2020/368, 2020. https://ia.cr/2020/368.

[5] Soojung An, Suhri Kim, Sunghyun Jin, HanBit Kim, and HeeSeok Kim. Single trace side channel analysis on ntru implementation. *Applied Sciences*, 8(11):2014, 2018.

[6] Dennis Andriesse. *Practical Binary Analysis*. No Starch Press, San Francisco, December 2018.

[7] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. *ACM SIGPLAN Notices*, 52(6):362–375, 2017.

[8] Amund Askeland and Sondre Rønjom. A side-channel assisted attack on ntru. Cryptology ePrint Archive, Report 2021/790, 2021. https://ia.cr/2021/790.

[9] Konstantinos Athanasiou, Byron Cook, Michael Emmi, Colm MacCarthaigh, Daniel Schwartz-Narbonne, and Serdar Tasiran. Sidetrail: Verifying time-balancing of cryptosystems. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 215–228. Springer, 2018.

[10] Reza Azarderakhsh, Matthew Campagna, Craig Costello, LD Feo, Basil Hess, Amir Jalali, David Jao, Brian Koziel, Brian LaMacchia, Patrick Longa, et al. Supersingular isogeny key encapsulation. *Submission to the NIST Post-Quantum Standardization project*, 152:154–155, 2017.

[11] Fatih Balli, Andrea Caforio, and Subhadeep Banik. Some applications of hamming weight correlations. Cryptology ePrint Archive, Report 2021/611, 2021. https://ia.cr/2021/611.

[12] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[13] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, volume 13, page 14, 2010.

[14] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The simon and speck families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404, 2013. https://ia.cr/2013/404.

[15] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. Lightweight aead and hashing using the sparkle permutation family. *IACR Transactions on Symmetric Cryptology*, 2020(S1):208–261, Jun. 2020.

[16] Daniel J. Bernstein. Cache-timing attacks on aes, 2005. http://cr.yp.to/papers.html#cachetiming.

[17] Daniel J Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. Ntru prime: reducing attack surface at low cost. In *International Conference on Selected Areas in Cryptography*, pages 235–260. Springer, 2017.

[18] Shivam Bhasin, Jakub Breier, Xiaolu Hou, Dirmanto Jap, Romain Poussier, and Siang Meng Sim. Sitm: See-in-the-middle side-channel assisted middle round differential cryptanalysis on spn block ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(1):95–122, Nov. 2019.

[19] Shivam Bhasin, Jan-Pieter D'Anvers, Daniel Heinz, Thomas Pöppelmann, and Michiel Van Beirendonck. Attacking and defending masked polynomial comparison for lattice-based cryptography. Cryptology ePrint Archive, Report 2021/104, 2021. https://ia.cr/2021/104.

[20] Nikolaj Bjørner and Phan Anh Dung. vz-maximal satisfaction with z3. In *6th International Symposium on Symbolic Computation in Software Science (SCSS 2014)*, pages 1–9, 2014.

[21] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. *vz*-an optimizing smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199. Springer, 2015.

[22] Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying constant-time implementations by abstract interpretation. *Journal of Computer Security*, 27(1):137–163, 2019.

[23] Monica Borda. *Statistical and Informational Model of an ITS*, pages 7–52. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[24] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 715–733. Association for Computing Machinery, 2021.

[25] Joppe W Bos, J Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. In *International Conference on Financial Cryptography and Data Security*, pages 157–175. Springer, 2014.

[26] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S Păsăreanu. Symbolic path cost analysis for side-channel detection. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 27–37, 2018.

[27] 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.

[28] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *European Symposium on Research in Computer Security*, pages 355–371. Springer, 2011.

[29] 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.

[30] Anantha P Chandrakasan and Robert W Brodersen. Minimizing power consumption in digital cmos circuits. *Proceedings of the IEEE*, 83(4):498–523, 1995.

[31] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 13–28. Springer, 2002.

[32] Sudipta Chattopadhyay and Abhik Roychoudhury. Symbolic verification of cache side-channel freedom. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2812–2823, 2018.

[33] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

[34] Michael R Clarkson and Fred B Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[35] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038, 2020.

[36] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[37] Nicolas Debande, Maël Berthier, Yves Bocktaels, and Thanh-Ha Le. Profiled model based power simulator for side channel evaluation. Cryptology ePrint Archive, Report 2012/703, 2012. https://ia.cr/2012/703.

[38] Jerry den Hartog, Jan Verschuren, E de Vink, Jaap de Vos, and W Wiersma. Pinpas: a tool for power analysis of smartcards. In *IFIP International Information Security Conference*, pages 453–457. Springer, 2003.

[39] A. Adam Ding, Cong Chen, and Thomas Eisenbarth. Simpler, faster, and more robust t-test based leakage detection. Cryptology ePrint Archive, Report 2015/1215, 2015. https://ia.cr/2015/1215.

[40] Daniel Dinu, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Johann Großschädl, and Alex Biryukov. Design strategies for arx with provable bounds: Sparx and lax. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 484–513. Springer, 2016.

[41] Elena Dubrova, Kalle Ngo, and Joel Gärtner. Breaking a fifth-order masked implementation of crystals-kyber by copy-paste. Cryptology ePrint Archive, Paper 2022/1713, 2022. https://eprint.iacr.org/2022/1713.

[42] Leo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehle. Crystals – dilithium: Digital signatures from module lattices. Cryptology ePrint Archive, Report 2017/633, 2017. https://ia.cr/2017/633.

[43] Thomas Espitau, Pierre-Alain Fouque, Benoit Gerard, and Mehdi Tibouchi. Side-channel attacks on bliss lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. Cryptology ePrint Archive, Report 2017/505, 2017. https://ia.cr/2017/505.

[44] Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. Relational symbolic execution. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, pages 1–14, 2019.

[45] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The skein hash function family. *Submission to NIST (round 3)*, 7(7.5):3, 2010.

[46] Georges Gagnerot. *Étude des attaques et des contre-mesures associées sur composants embarqués.* PhD thesis, Universite de Limoges, 2013.

[47] Aymeric Genêt, Natacha Linard de Guertechin, and Novak Kaluđerović. Full key recovery side-channel attack against ephemeral sike on the cortex-m4. Cryptology ePrint Archive, Report 2021/858, 2021. https://ia.cr/2021/858.

[48] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 426–442. Springer, 2008.

[49] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. Templates vs. stochastic methods. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 15–29. Springer, 2006.

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

[51] Qian Guo, Denis Nabokov, Alexander Nilsson, and Thomas Johansson. Sca-ldpc: A code-based framework for key-recovery side-channel attacks on post-quantum encryption schemes. Cryptology ePrint Archive, Paper 2023/294, 2023. https://eprint.iacr.org/2023/294.

[52] Shaobo He, Michael Emmi, and Gabriela Ciocarlie. ct-fuzz: Fuzzing for timing leaks. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 466–471. IEEE, 2020.

[53] Daniel Heinz, Matthias J. Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Daan Sprenkels. First-order masked kyber on arm cortex-m4. Cryptology ePrint Archive, Paper 2022/058, 2022. https://eprint.iacr.org/2022/058.

[54] Daniel Heinz and Thomas Pöppelmann. Combined fault and dpa protection for lattice-based cryptography. Cryptology ePrint Archive, Report 2021/101, 2021. https://ia.cr/2021/101.

[55] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking nist pqc on arm cortex-m4. Cryptology ePrint Archive, Report 2019/844, 2019. https://ia.cr/2019/844.

[56] Emre Karabulut, Erdem Alkim, and Aydin Aysu. Single-trace side-channel attacks on ω-small polynomial sampling: With applications to ntru, ntru prime, and crystals-dilithium. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 35–45. IEEE, 2021. https://ia.cr/2022/494.

[57] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[58] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

[59] Adam Langley. Ctgrind: checking that functions are constant time with valgrind, 2010. https://github.com/agl/ctgrind.

[60] Adam Langley, W Chang, Nikos Mavrogiannopoulos, Joachim Strombergson, and Simon Josefsson. Chacha20-poly1305 cipher suites for transport layer security (tls), 2016.

[61] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. Symbolic optimization with smt solvers. *ACM SIGPLAN Notices*, 49(1):607–618, 2014.

[62] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. Platypus: Software-based power side-channel attacks on x86. In *IEEE Symposium on Security and Privacy (SP)*, 2021.

[63] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, Berlin, Germany, 2008.

[64] Luke Mather, Elisabeth Oswald, Joe Bandenburg, and Marcin Wojcik. Does my device leak information? an a priori statistical power analysis of leakage detection tests. Cryptology ePrint Archive, Report 2013/298, 2013. https://ia.cr/2013/298.

[65] matthias j. kannwischer, joost rijneveld, peter schwabe, and ko stoffelen. pqm4: post-quantum crypto library for the arm cortex-m4. https://github.com/mupq/pqm4, 2019.

[66] 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.

[67] S Alvim M'rio, Kostas Chatzikokolakis, Catuscia Palamidessi, and Geoffrey Smith. Measuring information leakage using generalized gain functions. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 265–279. IEEE, 2012.

[68] Alexander Nadel and Vadim Ryvchin. Bit-vector optimization. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 851–867. Springer, 2016.

[69] Erick Nascimento, Lukasz Chmielewski, David Oswald, and Peter Schwabe. Attacking embedded ecc implementations through cmov side channels. Cryptology ePrint Archive, Report 2016/923, 2016. https://ia.cr/2016/923.

[70] Hamid Nejatollahi, Nikil Dutt, Sandip Ray, Francesco Regazzoni, Indranil Banerjee, and Rosario Cammarota. Post-quantum lattice-based cryptography implementations: A survey. *ACM Comput. Surv.*, 51(6), jan 2019.

[71] NIST. Pqc standardization process: Third round candidate announcement. Computer Security Resource Center (CSRC), 2020. https://csrc.nist.gov/News/2020/pqc-third-round-candidate-announcement.

[72] Yannic Noller. Differential program analysis with fuzzing and symbolic execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 944–947, 2018.

[73] Colin O'Flynn and Zhizhang (David) Chen. Chipwhisperer: An open-source platform for hardware embedded security research. Cryptology ePrint Archive, Report 2014/204, 2014. https://ia.cr/2014/204.

[74] Colin O'Flynn and Jasper van Woudenberg. *The Hardware Hacking Handbook: Breaking Embedded Security with Hardware Attacks*. No Starch Press, San Francisco, November 2021.

[75] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. Shadow of a doubt: testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1181–1192, 2016.

[76] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. Cryptology ePrint Archive, Report 2017/345, 2017. https://ia.cr/2017/345.

[77] Quoc-Sang Phan, Lucas Bang, Corina S Pasareanu, Pasquale Malacaria, and Tevfik Bultan. Synthesis of adaptive side-channel attacks. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 328–342. IEEE, 2017.

[78] Stjepan Picek, Annelie Heuser, Alan Jovic, Shivam Bhasin, and Francesco Regazzoni. The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations. Cryptology ePrint Archive, Report 2018/476, 2018. https://ia.cr/2018/476.

[79] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. Cryptology ePrint Archive, Report 2017/594, 2017. https://ia.cr/2017/594.

[80] The OpenSSL Project. Openssl: Tls/ssl and crypto library. https://github.com/openssl/openssl, 2021.

[81] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 142–159. Springer, 2013.

[82] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *International Conference on Research in Smart Cards*, pages 200–210. Springer, 2001.

[83] Mark Randolph and William Diehl. Power side-channel attack analysis: A review of 20 years of study for the layman. *Cryptography*, 4(2), 2020.

[84] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. Drop by drop you break the rock - exploiting generic vulnerabilities in lattice-based pke/kems using em-based physical attacks. Cryptology ePrint Archive, Report 2020/549, 2020. https://ia.cr/2020/549.

[85] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. On exploiting message leakage in (few) nist pqc candidates for practical message recovery and key recovery attacks. Cryptology ePrint Archive, Report 2020/1559, 2020. https://ia.cr/2020/1559.

[86] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on cca-secure lattice-based pke and kem schemes. Cryptology ePrint Archive, Report 2019/948, 2019. https://ia.cr/2019/948.

[87] Oscar Reparaz. Detecting flawed masking schemes with leakage detection tests. In *International Conference on Fast Software Encryption*, pages 204–222. Springer, 2016.

[88] Oscar Reparaz, Benedikt Gierlichs, and Ingrid Verbauwhede. Fast leakage assessment. Cryptology ePrint Archive, Report 2017/624, 2017. https://ia.cr/2017/624.

[89] Riscure. Inspector side channel analysis. Riscure company website, 2002. https://www.riscure.com/security-tools/inspector-sca.

[90] Konstantin Scherer, Tobias Pfeffer, and Sabine Glesner. I/o interaction analysis of binary code. In *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 225–230. IEEE, 2019.

[91] Tobias Schneider and Amir Moradi. Leakage assessment methodology - a clear roadmap for side-channel evaluations. Cryptology ePrint Archive, Report 2015/207, 2015. https://ia.cr/2015/207.

[92] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.

[93] Roberto Sebastiani and Patrick Trentin. OptiMathSAT: A Tool for Optimization Modulo Theories. In *International Conference on Computer Aided Verification (CAV)*, pages 447–454. Springer, 2015.

[94] Roberto Sebastiani and Patrick Trentin. Optimathsat: A tool for optimization modulo theories. *Journal of Automated Reasoning*, 64(3):423–460, 2020.

[95] Nader Sehatbakhsh, Baki Berkay Yilmaz, Alenka Zajic, and Milos Prvulovic. Emsim: A microarchitecture-level simulation tool for modeling electromagnetic side-channel signals. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 71–85. IEEE, 2020.

[96] Madura A Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. Cryptology ePrint Archive, Report 2019/1445, 2019. https://ia.cr/2019/1445.

[97] Madura A. Shelton, Łukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. Rosita++: Automatic higher-order leakage elimination from cryptographic code. Cryptology ePrint Archive, Report 2021/1181, 2021. https://ia.cr/2021/1181.

[98] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice – automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, volume 1, pages 1–1, 2015.

[99] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, volume 1, pages 1–1, 2015.

[100] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[101] Bo-Yeon Sim and Dong-Guk Han. Key bit-dependent attack on protected pkc using a single trace. In *International Conference on Information Security Practice and Experience (ISPEC)*, pages 168–185. Springer, 2017.

[102] Bo-Yeon Sim and Dong-Guk Han. Single-trace vulnerability of countermeasures against instruction-related timing attack. Cryptology ePrint Archive, Report 2019/1236, 2019. https://ia.cr/2019/1236.

[103] Bo-Yeon Sim, Junki Kang, and Dong-Guk Han. Key bit-dependent side-channel attacks on protected binary scalar multiplication. *Applied Sciences*, 8(11):2168, 2018.

[104] Bo-Yeon Sim, Jihoon Kwon, Joohee Lee, Il-Ju Kim, Taeho Lee, Jaeseung Han, Hyojin Yoon, Jihoon Cho, and Dong-Guk Han. Single-trace attacks on the message encoding of lattice-based kems. Cryptology ePrint Archive, Report 2020/992, 2020. https://ia.cr/2020/992.

[105] Bo-Yeon Sim, Aesun Park, and Dong-Guk Han. Chosen-ciphertext clustering attack on crystals-kyber using the side-channel leakage of barrett reduction. Cryptology ePrint Archive, Report 2021/874, 2021. https://ia.cr/2021/874.

[106] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, 2016.

[107] Francois-Xavier Standaert, Nicolas Veyrat-Charvillon, Elisabeth Oswald, Benedikt Gierlichs, Marcel Medwed, Markus Kasper, and Stefan Mangard. The world is not enough: Another look on second-order dpa. Cryptology ePrint Archive, Report 2010/180, 2010. https://ia.cr/2010/180.

[108] François-Xavier Standaert. How (not) to use welch's t-test in side-channel security evaluations. Cryptology ePrint Archive, Report 2017/138, 2017. https://ia.cr/2017/138.

[109] Hauke Malte Steffen, Lucie Johanna Kogelheide, and Timo Bartkewitz. In-depth analysis of side-channel countermeasures for crystals-kyber message encoding on arm cortex-m4. Cryptology ePrint Archive, Report 2021/1307, 2021. https://ia.cr/2021/1307.

[110] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[111] Céline Thuillet, Philippe Andouard, and Olivier Ly. A smart card power analysis simulator. In *2009 International Conference on Computational Science and Engineering*, volume 2, pages 847–852. IEEE, 2009.

[112] TrustedFirmware. mbedtls: An open source, portable, easy to use, readable and flexible ssl library. https://github.com/ARMmbed/mbedtls, 2021.

[113] Meltem Sönmez Turan, Kerry McKay, Donghoon Chang, Cgdas a Calık, Lawrence Bassham, Jinkeon Kang, and John Kelsey. Status report on the second round of the nist lightweight cryptography standardization process, 2021.

[114] Nikita Veshchikov. Silk: high level of abstraction leakage simulator for side channel analysis. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, pages 1–11, 2014.

[115] Nikita Veshchikov and Sylvain Guilley. Use of simulators for side-channel analysis. In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 104–112. IEEE, 2017.

[116] Dan Walters, Andrew Hagen, and Eric Kedaigle. Sleak: A side-channel leakage evaluator and analysis kit. Technical report, MITRE, Bedford, United States, 2014.

[117] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W Fletcher, and David Kohlbrenner. Hertzbleed: Turning power {Side-Channel} attacks into remote timing attacks on x86. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 679–697, 2022.

[118] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Microwalk: A framework for finding side channels in binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 161–173, 2018.

[119] Marc Witteman. Secure application programming in the presence of side channel attacks. Riscure Whitepaper, 2018. https://www.riscure.com/uploads/2018/11/201708_Riscure_Whitepaper_Side_Channel_Patterns.pdf.

[120] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–26, 2018.

[121] Lu Xiao and Howard M. Heys. An improved power analysis attack against camellia's key schedule. Cryptology ePrint Archive, Report 2005/338, 2005. https://ia.cr/2005/338.

[122] Zhuang Xu, Owen Pemberton, Sujoy Sinha Roy, David Oswald, Wang Yao, and Zhiming Zheng. Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of kyber. Cryptology ePrint Archive, Report 2020/912, 2020. https://ia.cr/2020/912.

[123] Yan Yan and Elisabeth Oswald. Examining the practical side channel resilience of arx-boxes. Cryptology ePrint Archive, Report 2019/335, 2019. https://ia.cr/2019/335.

[124] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

Appendix

1. Discussion of Existing Vulnerabilities

[103] present a single trace attack based on the power consumption properties of the key bit check of scalar multiplication used in Elliptic Curve Cryptography. Scalar multiplication and modular exponentiation consist of iterative operations associated with the private key bit k_i value. Accordingly, at the beginning of each iteration, the key bit value is extracted from an n-bit key string, $k = (k_{n-1}, k_{n-2}, \cdots, k_0)_2$ and stored in a k_i variable. Thus, power consumption is related to the k_i value. In software implementation, power consumption in the key bit check phase is associated with the Hamming weight of k_i $(0 \leq i \leq n-1)$, i.e., if $k_i = 0$, power consumption related to 0 occurs. Otherwise, power consumption is associated with 1 (power consumption differs when leakage is zero or one, i.e. $P(\omega(0)) \neq P(\omega(1))$).

Listing 10 shows the key identification function from OpenSSL [80]. The PoI comes immediately after the "`& ((BN_ULONG)1)`" operation is performed at the destination register.

```
1 int BN_is_bit_set(const BIGNUM *a, int n) {
2   int i, j;
3   bn_check_top(a);
4   if (n < 0) return (0);
5   i = n / BN_BITS2;
6   j = n % BN_BITS2;
7   if (a->top <= i) return (0);
8   return (int)(((a->d[i]) >> j) & ((BN_ULONG)1));
9 }
```

Listing 10: Key bit identification function of OpenSSL

Digital signature schemes generate a valid signature on a message using a secret key and the signature's authenticity can be verified with the associated public key. CRYSTALS-Dilithium [42] is a lattice-based digital signature scheme and one of the three finalists running for the NIST's post-quantum digital signature standard. An adversary can target the random challenge sampling during the signature generation at line 17 in Listing 11 where the implementation determines the sign of the non-zero coefficients.

```
1  void poly_challenge(poly *c,
2              const uint8_t seed[SEEDBYTES]) {
3    ...
4    for (i = 0; i < 8; ++i)
5      signs |= (uint64_t)buf[i] << 8 * i;
6    pos = 8;
7    for (i = 0; i < N; ++i) c->coeffs[i] = 0;
8    for (i = N - TAU; i < N; ++i) {
9      do {
10       if (pos >= SHAKE256_RATE) {
11         shake256_squeezeblocks(buf, 1, &state);
12         pos = 0;}
13       b = buf[pos++];
14     } while (b > i);
15     c->coeffs[i] = c->coeffs[b];
16     /* ↑ Δω = 31 ∘ ↓ Δω = 31 */
17     c->coeffs[b] = 1 - 2 * (signs & 1);
18     signs >>= 1;
19   }
```

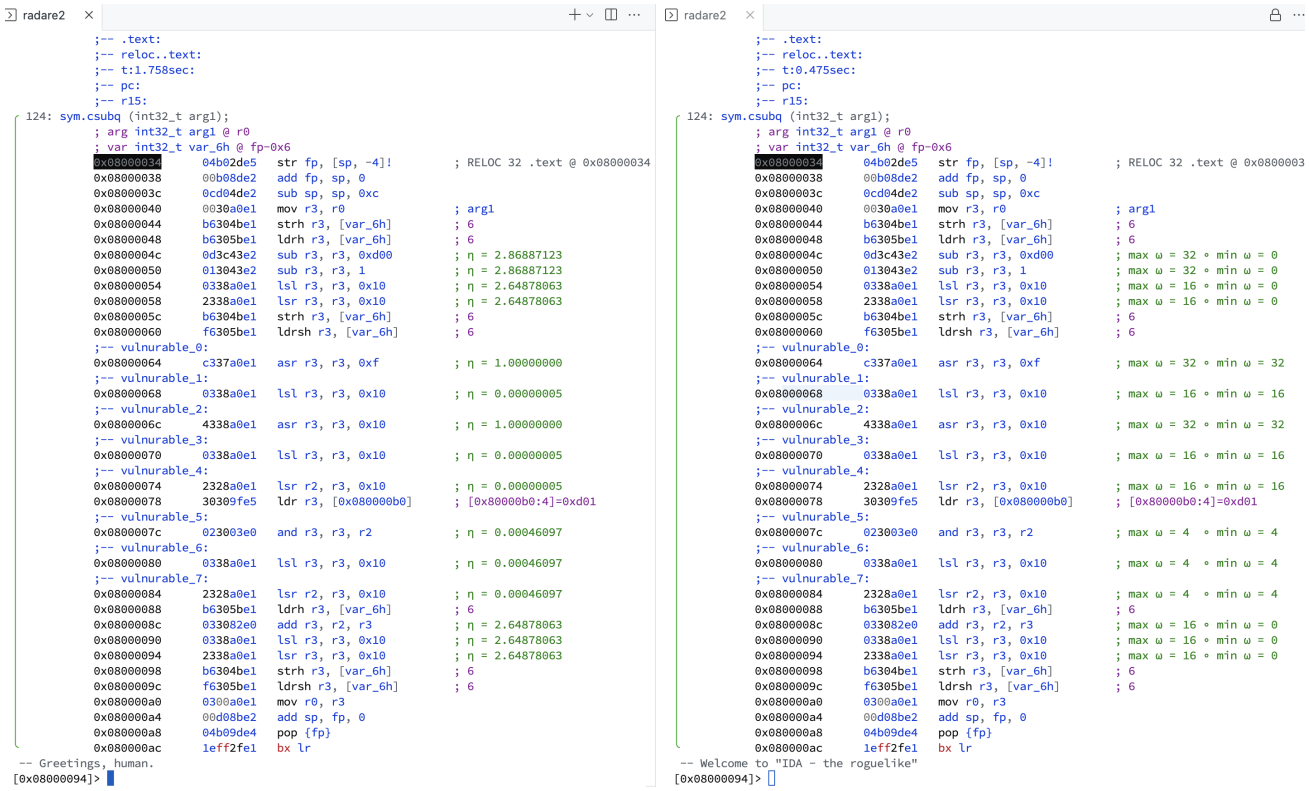

Figure 9: Pascal's Radare2 output for Kyber's conditional subtraction function —csubq— each secret-tainted arithmetic/logical instructions are annotated with Min and Max Hamming weights and approximate entropy. Point of Interests in Single-Trace side-channel attacks are flagged as *vulnerable*.

TABLE 6: Single-trace Power/EM side-channel attacks that our tool confirms

Work	Year	Type	Crypto Algorithm	Operation	Method	Classifier	Side-Channel Attack Type
[56]	2021	Power	NTRU, Dilithium	Polynomial Sampling	SOSD	NPDF	Single-Trace Template-based Attack
[109]	2021	Power	Crystals-Kyber	Message Encoding	SOST	t-test	Single-Trace Simple Power Attack
[8]	2021	EM	NTRU	Modular Reduction	N/A	RMS	Single-Trace Simple Power Attack
[47]	2021	Power	SIKE	Three Point Ladder	N/A	PCC	Single-Trace Correlation Power Analysis
[19]	2021	Power	Lattice-based	Masked Comparisons	TVLA	t-test	Single-Trace Simple Power Attack
[85]	2020	EM	LWE/LWR based	Message Decoding	TVLA	t-test	Single-Trace Template-based Attack
[104]	2020	Power	LWE/LWR based	Message Encoding	SOST	ML-based	Single-Trace Template-based Attack
[84]	2020	EM	Lattice-based	Message Encoding	TVLA	t-test	Single-Trace Template-based Attack
[4]	2020	Power	NewHope	Message Encoding	N/A	S	Single-Trace Simple Power Attack
[102]	2019	Power	HQC	Error Correction	N/A	k-means	Single-Trace Template-based Attack
[86]	2019	EM	Round5/LAC	Error Correction	TVLA	t-test	Chosen-Ciphertext Clustering Attack
[86]	2019	EM	Lattice-based	FO Transform	TVLA	t-test	Chosen-Ciphertext Clustering Attack
[123]	2019	Power	Sparx	ARX-box Assembly	N/A	PCC	Correlation Power Analysis Attack
[5]	2018	Power	NTRU	Polynomial Multiplication	N/A	Averaging	Single-Trace Simple Power Attack
[103]	2018	EM	Scalar Multiplication	Key bit check phase	SOST	k-means	Single-Trace Template-based Attack
[101]	2017	Power	Scalar Multiplication	Key bit check phase	SOST	k-means	Single-Trace Template-based Attack
[69]	2016	Power	Curve25519	Montgomery Ladder	N/A	NPDF	Single-Trace Template-based Attack

Classifier: Classification Method (Statistical Analysis for Clustering), EM: Electromagnetic Emanation, Power: Power-related side-channel attacks, PoI: Interesting points in time on the traces, TVLA: Test Vector Leakage Assessment [50], SOSD: Sum of Squared pairwise Differences of the average signals [48], SOST: Sum of Squared pairwise t-differences [49], S: Sum of Squared differences, k-means: k-means clustering algorithm, NPDF: Normal Probability Density Function, PCC: Pearson Correlation Coefficient [27], RMS: Root Mean Square.

```
}
```

Listing 11: Dilithium polynomial generation

This operation can leak information about how many negative and positive coefficients the private polynomial has, which gives a hint about the secret challenge. [56] target this coefficient assignment operation in Listing 11 through a single-trace template attack. The assignment possible outcomes are -1 ($0xFFF\ldots F$) with $\omega = 32$ and 1 ($0x000\ldots1$) with $\omega = 1$; hence, two significantly different power measurements due to the high Hamming weight difference.

```
#define int32_MINMAX(a, b)
do {
  int32_t ab = (b)^(a);
  int32_t c = (int32_t)((int64_t)(b)-(int64_t)(a));
  c ^= ab & (c ^(b));
  c >>= 31;  // ↑Δω = 32 ∘ ↓Δω = 32
  c &= ab;
  (a) ^= c;
  (b) ^= c;
```

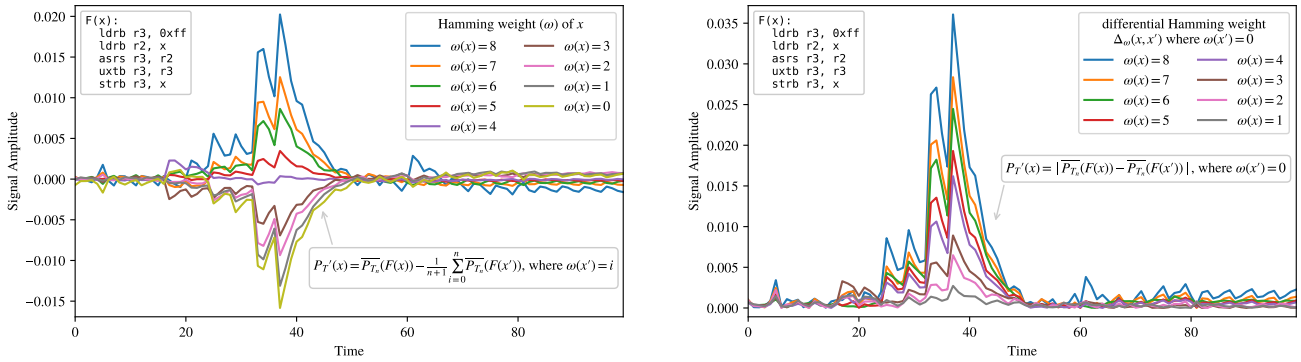

Figure 10: Observable difference in power traces. Execution of $R_d := R_d \gg x$ with all ω classes of $\mathbb{F}_8$. 1000 samples per ω class collected from a 32-bit ARM Cortex-M4 (STM32F3) target.

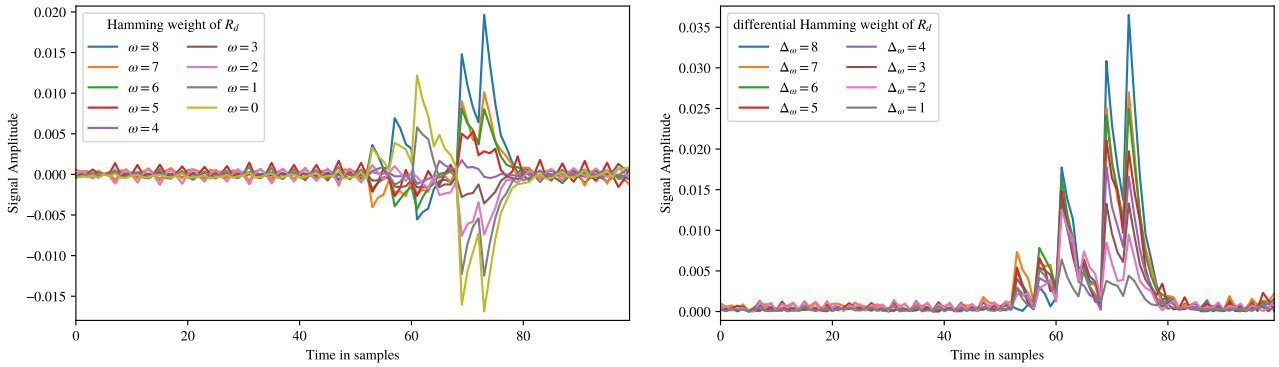

Figure 11: Observable difference in power traces. Execution of $R_d := R_d \gg x$ with all ω classes of $\mathbb{F}_8$. 1000 samples per ω class collected from a 32-bit ARM Cortex-M0 (STM32F0) target.

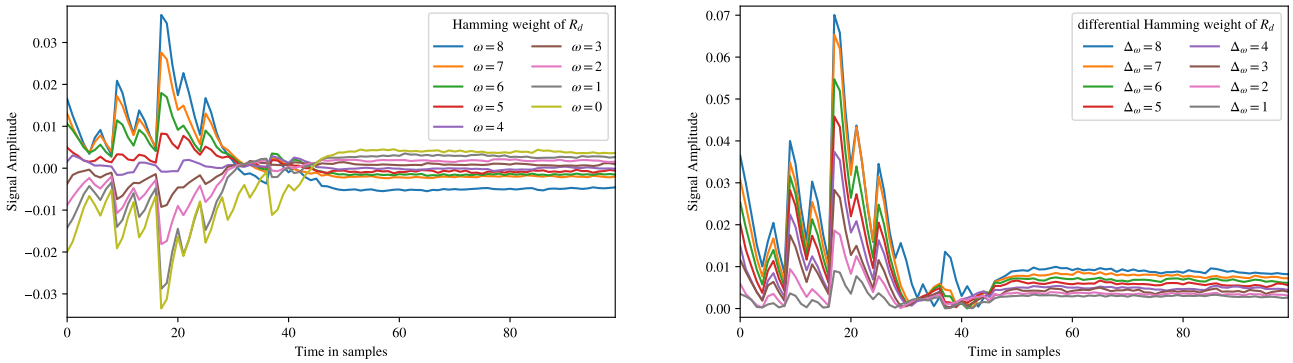

Figure 12: Observable difference in power traces. Execution of $R_d := R_d \gg x$ with all ω classes of $\mathbb{F}_8$. 1000 samples per ω class collected from a XMEGA 8-bit RISC target.

```
10 } while (0)
```

Listing 12: NTRU Comparison

NTRU is a public-key encryption and key-encapsulation mechanism, which allows to safely transfer a session key between two (or more) parties over an insecure medium. NTRU is one of the four remaining finalists of NIST post-quantum project. A specific constant-time sorting sub-routine used in NTRU [17] and NTRU Prime [17], is vulnerable to a power-based side-channel attack [56]. A significant power consumption difference between the two possible outputs 0 and -1 during the execution of the targeted shift operation at line 6 in Listing 12. This vulnerability occurs due to the significant difference in the Hamming weight representations. The -1 ($0xFFFFFFFF$) shows a power consumption behavior for ω = 32, while the output 0 ($0x00000000$) behavior matches with ω = 0.