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 University  
ruzica.piskac@yale.edu

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

Jakub Szefer  
Yale University  
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 tools 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 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...0 or 0xFF...F, then this instruction may lead to potential power side-channel vulnerability since power consumption of computation related to 0x00...0 is much different from for 0xFF...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 automate 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) :
\[ f(\text{sum}[8s], x[8s]) : \]
\[ r_0 := x_0 - 64 \]
\[ r_1 := r_0 \gg 7 \]
\[ r_2 := r_1 \]
\[ r_3 := r_2 \land x_0 \]
\[ r_4 := \text{sum}_0 + r_3 \]

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

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, \[ if (x \geq 64) then \text{sum} := \text{sum} + x \], where \( \text{sum} \) 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 \( \text{mask} \), 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 \( \star \)) extracts the sign bit of the 8-bit intermediate value, \( m_0[8s] \) (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 \( \text{mask} \) 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 \texttt{arm-none-eabi-gcc} compiler toolchain targeting 32-bit Arm architecture with the different compiler flags generates two different vulnerable locations. \texttt{asr} instruction performs arithmetic shift right operation over operands; and \texttt{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 utilizing 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 \( \text{sum} = 0 \) 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 \( \omega \)-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.,

\[ d(v, v') = \omega(x \oplus y), \quad \Delta d(v, v') \] and \( \Delta d(v, v') \) represent maximum and minimum Hamming distance between \( v \) and \( v' \), 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 \( P_{\mathbb{F}}(F(x)) \) 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 \( R_d : \overline{R_d} \gg x \) including an \( \text{asr} \) 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 monotonicity 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., \( \Delta_\omega(v, v') \triangleq |\omega(v) - \omega(v')| \). \( \Delta_\omega(v, v') \) 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, \( d(11110000, 00001111) = 8 \) whereas their \( \Delta_\omega \) 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 \( \Delta_\omega \) 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. \( |\mathbb{P}(F(k_1)) - \mathbb{P}(F(k_2))| > 0 \). 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 \texttt{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 \( \forall x, k_1, k_2 \in \mathbb{K}, |\omega(r_1(k_1, x)) - \omega(r_1(k_2, x))| \leq \nu \) where \( \omega \) is a function that returns Hamming weight of the register \( r \) and \( \nu \) is a threshold to distinguish Hamming weight difference. \( \nu \) 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 distinguishable 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 \( \omega \) of an 8-bit-width intermediate value \( v \), then this reduces the uncertainty about \( v \) as only \( \binom{8}{\omega} \) out of 256 values satisfy the observed Hamming weight. For example, observing an \( \omega \) of 4, it gives an attacker 70 possible secret keys, whereas observing an \( \omega \) 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: \( \omega \) classes and probabilities for \( \mathbb{F}_8 \)

<table>
<thead>
<tr>
<th>( \omega )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>\omega</td>
<td>)</td>
<td>1</td>
<td>.004</td>
<td>.031</td>
<td>.109</td>
<td>.219</td>
<td>.273</td>
<td>.219</td>
</tr>
</tbody>
</table>

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

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

\[
\bar{\eta}(r) = -\sum_{\omega_i \in \mathbb{F}_n} P_{\omega_i}(r) \cdot \log_2 P_{\omega_i}(r), \text{ where } r \in \mathbb{F}_n
\]

For instance, a register having only all-ones or all-zeros has \( \bar{\eta} \) 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 1 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 3 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 1) is lifted into the corresponding set of VEX statements in an irsb block by PyVEX [98] (stage 3). 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 2) 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 1. 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 4). 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 5) 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 6 and 7), we calculate the approximate entropy using \( \omega \)-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 \( r0 \), and store it in register \( r2 \) by sign extending to 32 bits. Consequently, \( r2 \) can only take two values, 0 or -1 (= 0xFFF ... F). Therefore, the
power side-channel analysis at address x103a8 results in
\(\Delta_{\omega} = 32\) and \(\Delta_{\omega} = 32\) with \(\bar{\eta} = 1.00\), and critical intermediate values for \(r_0\) are found as \(-32768\text{[16]}\) 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 Interest-of-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 \(\sigma_n\) that maintains a mapping of all registers, temporaries, and memory locations to symbolic expressions.

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 \(\bar{\beta}[8]\) denotes the secret (high) parameter \(x_0\), and symbolic bit-vector \(\bar{\lambda}[8]\) denotes the untainted (low) parameter \(sum[8]\). Taint propagation is performed dynamically over symbolic bit-vector expressions by relocating tainted 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 \(\bar{\beta}\) and \(\bar{\lambda}\) 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: \(\sigma_2\{r_1\} \rightarrow (\bar{\beta}[8] - 64) \gg 7\). At this point the bit-vector expression can be discharged to an SMT solver to get a satisfying model for symbolic value \(r_1: \exists x_8. bv_{lt}(x_8, 64)\), 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

![Figure 5: Pascal’s simplified tooling workflow under differential Hamming weight and \(\omega\)-class sampling models.](image-url)
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:

\[ \lambda = \lambda' \land \begin{cases} x_0 = \beta \land \text{sum}_0 = \lambda \land r_0 = x_0 - 64 \land r_1 = r_0 \gg 7 \land \\ x'_0 = \beta' \land \text{sum}'_0 = \lambda' \land r'_1 = x'_0 - 64 \land r'_1 = r'_0 \gg 7 \end{cases} \]

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 \( \varphi \) in state \( S_1 \) for the destination register \( r_1 \) can be then simplified to the following:

\[
\varphi_{SC_1} \triangleq r_1 \neq r'_1 \land r_1 = (\beta - 64) \gg 7 \land r'_1 = (\beta' - 64) \gg 7
\]

Once the symbolic execution reaches at state \( S_2 \), 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 \( \omega \) 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., \( |\omega(r_1) - \omega(r'_1)| = 7\) 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 Hamming weight class sampling (see Definition 8). If the maximum \( \Delta_w \) is equal to minimum \( \Delta_w \) or their difference (\( \uparrow \Delta_w \downarrow \downarrow \Delta_w \)) 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 \( \omega \) classes of those intermediate values (e.g. Table 1 for \( \mathbb{F}_2 \)).

To use in-bitvector expressions, we introduce a Hamming weight function \( \omega \) in the theory of bit-vectors in the SMT solver as \( \omega : \mathbb{F}_n \rightarrow \mathbb{F}_n \) 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 \( \omega \) function. The following shows the objective functions for the optimization problems constructed by Pascal at state \( S_2 \) (other states are similar).

<table>
<thead>
<tr>
<th>state ( S_k )</th>
<th>( \Delta_w ) objective function</th>
<th>optimization query</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_2 )</td>
<td>maximize ( \Delta_w(r_1, r'<em>1) ) ( \varphi</em>{SC_2} \land \max(\omega(r_1) - \omega(r'_1)) )</td>
<td>minimize ( \Delta_w(r_1, r'<em>1) ) ( \varphi</em>{SC_2} \land \min(\omega(r_1) - \omega(r'_1)) )</td>
</tr>
</tbody>
</table>

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 maximized, and at state \( S_1 \) 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.

Also, when 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 Pol 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

<table>
<thead>
<tr>
<th>machine code</th>
<th>( \bar{\eta}(\text{entropy}) )</th>
<th>( \Delta_w(\text{weight}) )</th>
<th>( d(\text{distance}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 )</td>
<td>( r_0 := x_0 - 64 )</td>
<td>( \bar{\eta}(r_0) = 2.54 )</td>
<td>( \Delta_1</td>
</tr>
<tr>
<td>( S_2 )</td>
<td>( r_1 := r_0 &gt; 7 ) ( \bar{\eta}(r_1) = 1.00 )</td>
<td>( \Delta_1</td>
<td>\mathcal{A}_0</td>
</tr>
<tr>
<td>( S_3 )</td>
<td>( r_2 := r_1 ) ( \bar{\eta}(r_2) = 1.00 )</td>
<td>( \Delta_1</td>
<td>\mathcal{A}_0</td>
</tr>
<tr>
<td>( S_4 )</td>
<td>( r_3 := r_2 \land x_0 )</td>
<td>( \bar{\eta}(r_3) = 2.54 )</td>
<td>( \Delta_1</td>
</tr>
<tr>
<td>( S_5 )</td>
<td>( r_4 := \text{sum}_0 + r_3 )</td>
<td>( \bar{\eta}(r_4) = 2.54 )</td>
<td>( \Delta_1</td>
</tr>
</tbody>
</table>

Figure 7: Three different leakage analysis results —Point-of-Interests are marked with stars.
Algorithm 1: Simplified version of differential symbolic register analysis at stage 4 in Figure 5

Ensure: for each secret-tainted VEX instruction
Data: r, r’: bit-vector expressions ∈ \mathbb{P}_u; solver: solver context
Result: (\bigvee \Delta_{\text{r'}}, \bigvee \Delta_{\text{r}}); (\omega\text{t}: witness, \omega\text{s}: witness)
1. (\bigvee \Delta_{\text{r'}}, \bigvee \Delta_{\text{r}}) ← (none, none).
2. objective ← | \omega\text{t}r(\text{r'})| − | \omega\text{s}r(\text{r'})|;
3. solver.push(solver.minimize(objective));
4. if solve(solver) = SAT then \bigvee \Delta_{\text{r}} ← solver.evaluating(objective);
5. solver.pop();
6. if \Delta_{\text{r}} = n then /* when \bigvee \Delta_{\text{r}} equals to width of r */
7. \bigvee \Delta_{\text{r'}} ← \bigvee \Delta_{\text{r}}; flag(vulnerable);
8. else
9. (\omega\text{t}, \omega\text{s}) ← (none, none);
10. solver.push(solver.maximize(objective));
11. if solve(solver) = SAT then
12. \bigvee \Delta_{\text{r'}} ← solver.evaluating(objective);
13. solver.pop();
14. (\omega\text{t}, \omega\text{s}) ← solver.evaluating(r, r');
15. solver.pop();
16. if \bigvee \Delta_{\text{r'}} = i ∧ \bigvee \Delta_{\text{r}} = i, where i is an \omega-class then
17. solver.push(solver.add(\omega(\text{n}) @ r'[\text{n}]));
18. if solve(solver) = UNSAT then flag(vulnerable);
19. solver.pop();
20. if \omega\text{t} ≠ none ∧ \omega\text{s} ≠ none ∧ \omega(\text{w}) = \omega(\text{w'});
21. solver.push(solver.add(\text{x}_{\text{w'}} ∧ \text{r'} ≠ \text{w}));
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 evaluate evaluates the given expression based on the model found. If the \bigvee \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 \bigvee \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 \bigvee \Delta_{\omega} equals to \bigvee \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 \omega 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. \text{w}_1 = 0x5A3C and \text{w}_2 = 0x5A5C) 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 shown 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

instance, for r_1 in Figure 6 it is \sigma_2[r_1] \implies [\beta] = 64 → 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. \( \mathbb{F}(2^8) \), 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 \( \Delta \omega \) poly, since there is no 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 \( \text{cortex-m4} \) options, it is experimentally shown to be exploitable in the literature. \[ \text{Listing 2: Full disassembly of Listing 1} \]

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.

A mask, containing 0 or -1 (\( = 0 \times FFFF \ldots \)), replaces the if-condition. The mask calculation is shown in Listing 3 at line 13 can be read from an oscilloscope display directly with the \(-O0\) shown in Listing 4), the shared secret \( \tilde{\omega} \) poly, frommsg(poly *r, unsigned char *msg){
  unsigned int i, j, mask;
  for (i = 0; i < 32; i++) {
    mask = -((msg[i] >> j) & 1);
    \( r \rightarrow \tilde{\omega}_{\text{coeffs}}[8+i]*j = \text{mask} \) (\( \text{NEWHOPE}_Q/2 \));
    \( r \rightarrow \tilde{\omega}_{\text{coeffs}}[8+i]*j+256 = \text{mask} \) (\( \text{NEWHOPE}_Q/2 \));
  }
}

\[ \text{Listing 3: Attack to NewHope’s message encoding} \]

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.

A mask, containing 0 or -1 (\( = 0 \times FFFF \ldots \)), replaces the if-condition. The mask calculation is shown in Listing 3 at line 13 can be read from an oscilloscope display directly with the \(-O0\) shown in Listing 4), the shared secret \( \tilde{\omega} \) poly, frommsg(poly *r, unsigned char *msg){
  unsigned int i, j, mask;
  for (i = 0; i < 32; i++) {
    mask = -((msg[i] >> j) & 1);
    \( r \rightarrow \tilde{\omega}_{\text{coeffs}}[8+i]*j = \text{mask} \) (\( \text{NEWHOPE}_Q/2 \));
    \( r \rightarrow \tilde{\omega}_{\text{coeffs}}[8+i]*j+256 = \text{mask} \) (\( \text{NEWHOPE}_Q/2 \));
  }
}

\[ \text{Listing 3: Attack to NewHope’s message encoding} \]

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.
In Listing 5 at line 2, Arm’s signed bit field extract (sbfx) is generated by the compiler for extraction of zero bit (1 bit) of \( r_2 \) and sign-extend it to 32 bits (if \( bit0(r_2) == 0 \), then \( r_2 = 0x0000\ldots \), else \( r_2 = 0xF000\ldots \)). This instruction’s semantics is also similar to right shifting, and actually the VEX IR lifter uses an ASr32 instruction in the intermediate representation.

```
1drb r2, [r3, #0] ; r2 = memory[r3]
sbfx r2, r2 , #0, #1 ; \( \Delta \omega_0 = 32 \) \( \Delta \omega_y = 32 \) \( \delta = 1.00 \)
strh r2, r2, #6144 ; r2 = r2 & 6144
and.w r2, r2 , [r0, #0]; \( \Delta \omega_0 = 32 \) \( \Delta \omega_y = 0 \) \( \delta = 3.53 \)
strh.w r2, [r0, #512] ; memory[r512] = r2
strh.w r2, [r0, #1024] ; memory[r1024] = 0
strh.w r2, [r0, #1536] ; memory[r1536] = r2
```

Listing 5: Partial disassembly at -03 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 P15, P17, P21, and P22. They are all confirmed with specific t-tests (TVLA) by attacking those PoIs reported by Pascal and using test vectors that are all confirmed with specific t-tests (TVLA) by attacking those PoIs reported by Pascal.

As an example, Listing 6 and Listing 7 shows the detected PoI in P21 of mbedTLS [112]. The value of \( \text{ret} \) at line 7 in C code is defined as a sensitive bit-dependent determinant. Its state corresponds to the register \( r_0 \) 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.

```
unsigned mbedtls_ct_mpi_uniq_lt(const uint64_t x, const uint64_t y) {
    \( \text{uint64\_t ret, \text{cond}}; \)
    cond = (x \& y); \( \text{ret} = (x \& y) \& \text{cond}; \)
    \( \text{ret} \| y \& \text{cond}; \)
    return (unsigned)ret;
}
```

Listing 6: P21; Constant-flow LT comparison.

### 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 2: Known and New Vulnerabilities

All known attacks in literature leverage Hamming weight leakage model.
### TABLE 3: Pascal’s Performance Analysis on known and some new vulnerabilities.

| # of Inst. | Hamming weight (\(|d|\Delta_\omega\))^\(\dagger\) | Hamming distance (\(|d|\))^\(\dagger\) | Approximate Entropy (\(\hat{q}\))^\(\dagger\) | Pascal |
|------------|-------------------------------|-----------------|-----------------|-------|
| P1         | 77                            | 2.13             | 4.49            | 5.13  |
| P2         | 73                            | 2.03             | 3.89            | 4.34  |
| P3         | 52                            | 1.22             | 1.96            | 3.01  |
| P4         | 46                            | 2.00             | 2.10            | 5.35  |
| P5         | 24                            | 2.17             | 2.21            | 2.22  |
| P6         | 64                            | 0.46             | 1.03            | 1.31  |
| P7         | 60                            | 0.31             | 0.91            | 1.43  |
| P8         | 75                            | 14.07            | 15.93           | 16.26 |
| P9         | 104                           | 23.88            | 27.45           | 27.09 |
| P10        | 100                           | 68.61            | 81.49           | 90.57 |
| P11        | 184                           | 129.033          | 146.66          | 159.51|
| P12        | 79                            | 2.51             | 2.97            | 7.55  |
| P13        | 139                           | 5.40             | 10.81           | 21.81 |
| P14        | 48                            | 5.35             | 4.82            | 48.13 |
| P15        | 63                            | 3.83             | 7.04            | 6.38  |
| P16        | 149                           | 369.77           | 139.70          | 600   |
| P17        | 223                           | 5.16             | 11.03           | 21.48 |
| P18        | 67                            | 24.71            | 48.76           | 58.32 |
| P19        | 110                           | 161.23           | 158.77          | 170.17|
| P20        | 98                            | 34.10            | 50.36           | 67.59 |
| P21        | 66                            | 3.83             | 6.04            | 1.02  |
| P22        | 31                            | 0.47             | 0.48            | 0.87  |
| P23        | 31                            | 1.64             | 2.26            | 4.08  |
| P24        | 80                            | 12.19            | 11.93           | 29.23 |

All times are in seconds. ^1 basic [20] and symba [61] algorithms are part of Z3 whereas obvbs is provided by OptiMathSAT [94]. ^\(\dagger\): parallel bitvector solving with a pool of 8 cores in Z3; bp: encoding \(\omega\)-classes as Pseudo Boolean equalities in Z3; inc: incremental bitvector solving via CV5. ^O: 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 Z3, 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 P16, the analysis takes 382.49 seconds with Hamming weight model using symba while the instruction count is 149, whereas, in P17, 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 P14 under Hamming distance model. We couldn’t identify what makes this specific query hard for solvers.

In \(\omega\)-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 \(\omega\)-class constraints as Pseudo-Boolean equalities in Z3 (pb.) and incremental encoding using CV5 [12] (inc.). We also set full timeout for algorithms, which is 600 seconds per benchmark and method. par. and pb. timed out for benchmark P16 and thus inc. method on CV5 solver outperforms par. and pb. methods, however, if we remove this benchmark, method pb. outperforms others.

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

![Figure 8: Distributions of computational times show that although median of those methods are close, in average (▲) \(\Delta_\omega\) analysis with obvbs performs slightly better.](image-url)
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].

```c
void poly_frommsg(poly *r,
const uint8_t msg[KYBER_INDCPA_MSKBYTES]) {
    unsigned int i, j;
    int16_t_t mask;
    for (i = 0; i < KYBER_N / 8; i++) {
        for (j = 0; j < 8; j++) {
            mask = (~int16_t_t(msg[i] >> j) & 1);
            /**\( \Delta_{\omega} = 16 \downarrow \Delta_{\omega} = 16 \| \Delta_{\omega} = 16 \) if \( \eta = 1.000^*/
            r->coeffs[8*i + j] = mask & ((KYBER_Q + 1)/2);
        }
    }
}
```

Listing 8: P6: CRYSTALS-Kyber's message encoding

<table>
<thead>
<tr>
<th>CRYSTALS-Kyber Message Encoding</th>
<th>t-test ( t_{max} )</th>
<th>Attack Success</th>
<th>Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td>P6 (Unprotected NIST Submission) [84], [109]</td>
<td>437</td>
<td>100%</td>
<td>TP</td>
</tr>
<tr>
<td>P7 (Message encoding with multiplication [4])</td>
<td>177.0</td>
<td>100%</td>
<td>TP</td>
</tr>
<tr>
<td>P8 (Data independent polynomial gen. [109])</td>
<td>24.8</td>
<td>68.6%</td>
<td>TP</td>
</tr>
<tr>
<td>P9 (Balanced data independent poly. gen. [109])</td>
<td>19.6</td>
<td>67.9%</td>
<td>TP</td>
</tr>
<tr>
<td>P10 (Polynomial randomization [109])</td>
<td>13.8</td>
<td>64.0%</td>
<td>TP</td>
</tr>
<tr>
<td>P11 (Byte and bit level random ordering [109])</td>
<td>5.2</td>
<td>50.1%</td>
<td>FP</td>
</tr>
</tbody>
</table>

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 P7, P8, P9 and P10. A combination of masking and shuffling increases the trace requirement for the attack, and benchmark P11 shuffles message encoding using byte and bit level random ordering. If we compare it with the reference implementation, P3, 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. P11 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.

```c
void masked_poly_frommsg(poly *r,
const uint8_t msg[KYBER_INDCPA_MSKBYTES]) {
    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++) {
        p_r[i] = r;
        p_r[i + 1] = &r_d;
        for (j = 0; j < 255; j++) {
            *(xorMasks + i) = *(xorMasks + i + b_inv);
            for (i_r = i ^ i_m;
                 j_r = j  ^ j_m;
                 r_d.coeffs[i] = (KYBER_Q + 1)/2; /\( \|↑\)Δ
            {
                r->coeffs[8*i + j] = mask & ((KYBER_Q + 1)/2);
            }
        }
    }
}
```

Listing 9: P11: Protected version of Listing 8.

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.

We expect no False Negatives since 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 P3) unless the precondition on input variable is given i.e. the input coefficients must be one of \([0, 2]\). In benchmarks P5, P15, and P23, neither of the models detect any vulnerabilities and in fact we weren’t able to find a significant t-test value above 10. Benchmark P23, ARX box construction of SPARX [40] (see Listing 2), is vulnerable while benchmark P24, ARX box of NIST’s LWC finalist Sparkle [15] is secure; Pascal did not detect any Pol, and we did not detect any presence of power leakage via TVLA.

**Target Architectures.** We have confirmed the presence of those vulnerable Pols 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 their 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.

There are leakage emulators in the literature such as Elmo [66], Rosita [96], and Rosita++ [97] for ARM Cortex-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.

<table>
<thead>
<tr>
<th>Work</th>
<th>Year</th>
<th>Target Device(s)</th>
<th>SC Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal</td>
<td>2022</td>
<td>Any Target</td>
<td>Yes</td>
</tr>
<tr>
<td>Rosita++ [97]</td>
<td>2021</td>
<td>ARM Cortex M0/M4</td>
<td>Yes</td>
</tr>
<tr>
<td>Emsim</td>
<td>2020</td>
<td>Rise-V</td>
<td>No</td>
</tr>
<tr>
<td>Rosita</td>
<td>2019</td>
<td>ARM Cortex M0</td>
<td>Yes</td>
</tr>
<tr>
<td>Elmo</td>
<td>2017</td>
<td>ARM Cortex M0</td>
<td>Yes</td>
</tr>
<tr>
<td>Ascold</td>
<td>2017</td>
<td>ATmega163</td>
<td>Yes</td>
</tr>
<tr>
<td>Savasca</td>
<td>2017</td>
<td>ATmega163</td>
<td>Yes</td>
</tr>
<tr>
<td>[87]</td>
<td>2016</td>
<td>not relevant</td>
<td>No</td>
</tr>
<tr>
<td>Sleak</td>
<td>2014</td>
<td>ARM Cortex A8</td>
<td>No</td>
</tr>
<tr>
<td>Silk</td>
<td>2014</td>
<td>ATmega328P</td>
<td>Yes</td>
</tr>
<tr>
<td>[46]</td>
<td>2013</td>
<td>Rise-V</td>
<td>Yes</td>
</tr>
<tr>
<td>[37]</td>
<td>2012</td>
<td>not specified</td>
<td>Yes</td>
</tr>
<tr>
<td>[111]</td>
<td>2009</td>
<td>AT90XX, ATmegaXX</td>
<td>Yes</td>
</tr>
<tr>
<td>Pinpas</td>
<td>2003</td>
<td>smartcards</td>
<td>Yes</td>
</tr>
</tbody>
</table>

† 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 grey-box model (microarchitectural information needed); and a white-box model (formal instruction semantics needed). ||: publicly available or not.

TABLE 5: Existing Power/EM Side Channel Analysis tools and methods, adapted from the SoK paper of [29].
References


[92] Edward J Schwartz, Thanassis Averinos, 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.


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, \( k = (k_{n-1}, k_{n-2}, \ldots, k_0) \) 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 ≤ i ≤ 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., \( P(0i) \neq P(0i) \)).

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

```c
int BN_is_bit_set(const BIGNUM *a, int n) {
    int i, j;
    bn_check_top(a);
    if (n < 0) return (0);
    i = n / BN_BITS2;
    j = n % BN_BITS2;
    if ((a->top <= i) return (0);
    return ((a->d[i] >> j) & ((BN_ULONG)1));
}
```

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.
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

<table>
<thead>
<tr>
<th>Work</th>
<th>Year</th>
<th>Type</th>
<th>Crypto Algorithm</th>
<th>Operation</th>
<th>Method</th>
<th>Classifier</th>
<th>Side-Channel Attack Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>[56]</td>
<td>2021</td>
<td>Power</td>
<td>NTRU, Dilithium</td>
<td>Polynomial Sampling</td>
<td>SOSD</td>
<td>NPDF</td>
<td>Single-Trace Template-based Attack</td>
</tr>
<tr>
<td>[47]</td>
<td>2021</td>
<td>Power</td>
<td>SIKE</td>
<td>Three Point Ladder</td>
<td>N/A</td>
<td>PCC</td>
<td>Single-Trace Correlation Power Analysis</td>
</tr>
<tr>
<td>[85]</td>
<td>2020</td>
<td>EM</td>
<td>LWE/LWR based</td>
<td>Message Decoding</td>
<td>TVLA</td>
<td>t-test</td>
<td>Single-Trace Template-based Attack</td>
</tr>
<tr>
<td>[102]</td>
<td>2019</td>
<td>Power</td>
<td>HQC</td>
<td>Error Correction</td>
<td>N/A</td>
<td>k-means</td>
<td>Single-Trace Template-based Attack</td>
</tr>
<tr>
<td>[86]</td>
<td>2019</td>
<td>EM</td>
<td>Lattice-based</td>
<td>FO Transform</td>
<td>TVLA</td>
<td>t-test</td>
<td>Chosen-Ciphertext Clustering Attack</td>
</tr>
<tr>
<td>[123]</td>
<td>2019</td>
<td>Power</td>
<td>Sparx</td>
<td>ARX-Box Assembly</td>
<td>N/A</td>
<td>PCC</td>
<td>Correlation Power Analysis Attack</td>
</tr>
<tr>
<td>[103]</td>
<td>2018</td>
<td>EM</td>
<td>Scalar Multiplication</td>
<td>Key bit check phase</td>
<td>SOST</td>
<td>k-means</td>
<td>Single-Trace Template-based Attack</td>
</tr>
<tr>
<td>[101]</td>
<td>2017</td>
<td>Power</td>
<td>Scalar Multiplication</td>
<td>Key bit check phase</td>
<td>SOST</td>
<td>k-means</td>
<td>Single-Trace Template-based Attack</td>
</tr>
<tr>
<td>[69]</td>
<td>2016</td>
<td>Power</td>
<td>Curve25519</td>
<td>Montgomery Ladder</td>
<td>N/A</td>
<td>NPDF</td>
<td>Single-Trace Template-based Attack</td>
</tr>
</tbody>
</table>


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 ... F) with $\omega = 32$ and 1 (0x000 ... 1) with $\omega = 1$; hence, two significantly different power measurements due to the high Hamming weight difference.

```c
#define int32_MINMAX(a, b)
    	int32_t ab = (b)\*(a);
    	int32_t c = (int32_t)(int64_t)(ab) - (int64_t)(a);
    	c = ab & (c \*= b);
    	c = c & 0x0000003f
```
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 $\omega = 32$, while the output 0 (0x00000000) behavior matches with $\omega = 0$.

Listing 12: NTRU Comparison

```c
while (0)
{
```

Figure 10: Observable difference in power traces. Execution of $R_d := R_d \gg x$ with all $\omega$ classes of $\mathbb{Z}_8$. 1000 samples per $\omega$ 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 $\omega$ classes of $\mathbb{Z}_8$. 1000 samples per $\omega$ 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 $\omega$ classes of $\mathbb{Z}_8$. 1000 samples per $\omega$ class collected from a XMEGA 8-bit RISC target.

```c

```