

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

microSCALE: Static Analysis for Microarchitectural Side-channel Leakage Evaluation

Akshay Kumar E, Pranav Krishna N, Annapurna Valiveti, Pallavi Borkar, Aditi Roy, and Chester Rebeiro, *IIT Madras*

https://www.usenix.org/conference/usenixsecurity26/presentation/kumar-e

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

microSCALE: Static Analysis for Microarchitectural Side-channel Leakage Evaluation

Akshay Kumar E*
IIT Madras, India

Pranav Krishna N*
IIT Madras, India

Annapurna Valiveti
IIT Madras, India

Pallavi Borkar
IIT Madras, India

Aditi Roy
IIT Madras, India

Chester Rebeiro
IIT Madras, India

Abstract

Developing cryptographic implementations that are resistant to side-channel attacks is challenging in modern CPUs. Data-dependent switching activity induces subtle transitions deep in the microarchitecture that can cause sensitive information to leak through the CPU's power consumption. Existing leakage detection tools largely rely on simulation-based testing, which is not only time-consuming but is also limited to the explored execution scenarios.

We present `microSCALE`, a fully automated framework for detecting power side-channel leakages that occur due to transitions in the CPU's microarchitecture. By statically analyzing the CPU's RTL, `microSCALE` enables systematic exploration of microarchitectural leakage sources within seconds. It precisely localizes the origin of leakage in the hardware and traces it back to the corresponding software instructions. We evaluate `microSCALE` on several protected crypto-implementations, including GIFT, ASCON, PRESENT, and AES, running on an open-source RISC-V processor, and identify several leakage sources. For instance, we identify that a protected implementation of the GIFT cipher has about 80% of its instructions that leak on the Rocket Core. We show how `microSCALE` can help in hardening this implementation, reducing the side-channel leakage around 90.3%.

1 Introduction

Since Paul Kocher's seminal work in the late 1990s demonstrated that cryptographic keys could be leaked from the device's power consumption [21], side-channel analysis (SCA) has emerged as a critical challenge in the design of secure embedded systems. Unlike traditional cryptanalytic techniques that target weaknesses in algorithms, side-channel analysis exploits information leaked inadvertently from the implementation. In particular, the instantaneous power drawn by a device exhibits data-dependent variations that correlate with the operations being executed. By capturing and analyzing these signals, an adversary can infer sensitive information about the program. The non-invasive nature of these measurements, combined with their high fidelity in extracting sensitive information from the device, makes power and electromagnetic (EM) side-channel attacks a potent threat to the security of modern cryptographic implementations.

Microarchitectural effects in modern processors, arising from unintended interactions of sensitive data within registers, caches, and the pipeline stages, have emerged as a major source of side-channel leakage and are known to invalidate established countermeasures. One prominent example is masking, a widely studied technique that decomposes sensitive variables into multiple statistically independent shares in order to decorrelate side-channel observations from secret values. However, microarchitectural effects can reintroduce dependencies between these shares, rendering practical implementations vulnerable to side-channel attacks [2, 10].

Leakage due to microarchitectural interactions arises from the gap between the CPU's instruction set architecture (ISA) and the behavior of the underlying hardware during execution [23,26]. While the ISA defines the functional outcomes of instructions, it abstracts the low-level implementation details that govern how these outcomes are achieved. In practice, executing an instruction entails a sequence of micro-operations, such as data transfers across pipeline stages, forwarding, and cache activity in the memory hierarchy. Similarly, load and store operations may trigger complex interactions within the memory subsystem, including alignment adjustments and prefetching mechanisms.

Together, these behaviors collectively determine how intermediate values flow through shared microarchitectural structures during execution. The instantaneous power consumed by the processor is influenced by the switching activity in each of these structures. The switching activity, in turn, is a function of the change in the data stored in the structure, giving rise to *transitional leakage* [25]. This can potentially lead to recombination of shares of a masked variable to leak sensitive information. Accurately identifying and analyzing these

*Both authors contributed equally. Corresponding authors contact: akshay@cse.iitm.ac.in, npranav7619@gmail.com

transitions is therefore critical to assessing the side-channel security of cryptographic implementations.

Contemporary approaches that estimate side-channel leakage from cryptographic implementations rely on empirical techniques that statistically evaluate leakage [16, 34] or develop device-specific models [22,24,29–31]. Such approaches may miss leakage sources due to limited statistical coverage and restricted exploration of the processor's behavior during analysis. In contrast, alternate approaches aim to systematically identify potential leakage sources by modeling the microarchitectural side-channel behavior of the CPU in a machine-readable form [6, 17, 33]. However, this methodology involves considerable manual effort, requiring experts with knowledge of both formal verification and microarchitectural design to develop and adapt the specifications. These challenges are further exacerbated as processor designs become more complex, resulting in a state-explosion problem that complicates formal modeling and verification. Moreover, much of the prior literature focuses on detecting leakage or establishing its presence, but provides limited insight into its origin. On the hardware side, existing methods rarely attribute leakage to specific microarchitectural signals or the elements whose transitions generate observable power variations. On the software side, they similarly fail to identify the instruction-level interactions responsible for transitional leakage. As a result, detection provides limited insight into the underlying causes of leakage.

Challenges: These limitations highlight the challenges in comprehensively characterizing leakage behavior. Nevertheless, reasoning about transitional leakage requires understanding not only the program logic, but also the operand-dependent microarchitectural execution behavior underlying it. The following points outline the key factors that complicate a comprehensive analysis of side-channel leakage on modern processors. **(C1)** *Modeling microarchitectural interactions.* Systematic analysis of a processor's Register-Transfer Level (RTL) design is particularly challenging due to its complex structure and tightly interconnected modules. Characterizing leakage requires modeling the feasible paths through which data propagates within the microarchitecture and capturing data propagation across microarchitectural components. As these components may interact under varying conditions, precise and complete modeling is non-trivial. **(C2)** *Detecting leakage.* Data may interact within the microarchitecture in several different ways. Not all of these interactions can result in leakage. Identifying interactions that can cause leakage requires analyzing how variables propagate and transform within the design, and identifying the specific interactions responsible for leakage. **(C3)** *Localizing the source of leakage*: After detecting leakage, the next step is to identify the microarchitectural element responsible and trace it back to the program-level operations that induced it. Pinpointing the exact source of leakage is essential for understanding its cause and designing effective mitigations.

Our Contribution: In this paper, we propose `microSCALE`, a fully automated framework that statically analyzes the processor's structural design to detect and localize hidden leakage sources. To address challenge C1, `microSCALE` models the processor as a graph, where nodes represent signals (registers and wires), and edges capture the data and control flow between them. This representation enables the framework to track all reachable data paths across the microarchitecture. Our framework uses a custom data structure to track the propagation of program variables across microarchitectural components. To address C2, the framework analyzes all possible interactions between program variables as they propagate through the graph. It then applies a set of rules to determine which of these interactions may lead to side-channel leakage. These rules identify the interactions that result in the recombination of secret shares, thereby violating the masking assumptions. Further, `microSCALE` aims to precisely localize the source of the identified leakage within the target microarchitecture. To this end, the framework pinpoints the microarchitectural elements involved in the identified interactions and traces them back to the corresponding program instructions operating on the associated data, thus addressing C3.

In summary, the key contributions of this work are as follows.

- We develop a novel framework named `microSCALE` that detects transitional leakage when a secure cryptographic program is implemented on a target processor RTL. `microSCALE` localizes the root cause of transitional leakage in both hardware and software. In the hardware design, it identifies the microarchitectural signals whose transitions produce observable leakage. In the software implementation, it pinpoints the instruction pairs that trigger these transitions when executed on the underlying hardware.

- Our framework extracts a graph-based representation of the processor's RTL, capturing how program variables propagate through the microarchitecture. On this representation, `microSCALE` exhaustively identifies all possible data paths and applies a set of rules to model data propagation and detect microarchitectural interactions that result in leakage in secure cryptographic implementations. By performing this analysis statically, `microSCALE` avoids hardware execution and remains independent of runtime inputs.

- We empirically validate our framework on the open-source RISC-V *RocketChip* processor by verifying secure implementations of cryptographic ciphers, including GIFT, AES, Present, and Ascon, at various security orders.

- We also validate a complete round of the GIFT cipher on our framework and evaluate the same using an FPGA setup. We further evaluate a hardened implementation of GIFT and demonstrate a reduction of 90.3% in leakage.

The source code of `microSCALE`, along with the corresponding cipher implementations, is available at: https://zenodo.org/records/20409592.

2 Background: Side Channel Attacks and Masking

In CMOS circuits, a signal switching from one logical state to another requires charging or discharging of internal capacitors. These transitions induce data-dependent variations in the circuit's power consumption. An adversary who can observe the power consumption can exploit this dependency [8] to infer data used in the operation. In a crypto-device, for example, these side-channel observations have been shown to compromise the secret key by leaking information about sensitive intermediate variables present in the implementation.

Masking [9, 20] is a widely used countermeasure for side-channel attacks that breaks correlation between intermediate variables and the secret. In a d-order masking scheme, each sensitive variable is split into $d+1$ shares. For example, in first-order masking, a secret variable x is encoded using two shares x_0 and x_1. The share x_0 is chosen from a uniform random distribution, while x_1 is chosen such that: $x = x_0 \oplus x_1$. Thus, each individual share x_0 and x_1 is statistically independent from the secret variable x.

Assuming that shares are always processed in isolation, such masking schemes are shown to be provably secure [5, 20, 28], leaking no information about the secret. Formally, the security proofs guarantee that no information about the secret variable can be obtained from any single share or from any combination of shares below the masking order d. In practice, however, when masked software executes on a processor, shares can unintentionally recombine producing side-channel leakage that depends jointly on multiple shares, undermining the guarantees of masking [10]. Such recombinations can occur in registers, memory, or any other element deep within the processor's microarchitecture. Identifying where and how recombinations occur in a processor's microarchitecture is critical to realize side-channel secure crypto software. In the next section, we discuss with examples how such recombinations can occur in the processor's microarchitecture, subverting the benefits of masking.

3 Side-Channel Leakage due to Microarchitectural Recombinations

To understand how recombinations occur, consider the instruction trace shown in Listing 1, where x_0 and x_1 are shares of the secret variable x and y_0 and y_1 are shares of the secret variable y (*i.e.* $x = x_0 \oplus x_1$ and $y = y_0 \oplus y_1$). A random variable r is used to ensure the statistical independence of intermediate computations involving two or more shares. Notice that none of the instructions operate directly on the sensitive variables x and y. Thus, if the shares are chosen to be independent of x and y, and the random value r uniformly distributed, there should be no side-channel leakage of these sensitive variables.

In practice, however, this is not true. Side-channel leakage of x and y does occur when these instructions execute on a microprocessor. To illustrate this, consider Figure 1, which shows part of a processor's execution unit that performs `AND`, `XOR`, and `OR` operations. Based on the instructions, multiplexers and demultiplexers channel data into arithmetic gates, causing transitions at their output registers. Microarchitectural elements D_1 to D_7 in the circuit store intermediate data.

Consider instructions I_1 and I_2, which moves the shares x_0 and x_1 to temporary registers t_1 and t_2 respectively. During the execution of these instructions, data corresponding to x_0 and x_1 propagate through the pipeline in every clock cycle. A partial propagation of the data corresponding to the execution of these instructions is shown in Figure 1: x_0 is first stored in the microarchitectural element D_1 when I_1 executes. It then transitions to x_1 subsequently, when I_2 executes. The switching activity induced by this transition in D_1 jointly depends on both x_0 and x_1, causing a *recombination* of the shares of x. This joint dependence manifests in the power consumption, which can correlate with the secret variable x, thus resulting in side-channel leakage. Such leakage due to recombination of shares is called *transitional leakage*.

Adding a random value to mask intermediate computations does not help prevent leakage. For example, instructions I_5 and I_6, performs the `XOR` of t_3 (i.e. x_0y_0) and t_5 (i.e. x_1y_1) with the random value r, can cause a transition at the output of the `XOR` gate at D_5. The transition between $x_0y_0 \oplus r$ and $x_1y_1 \oplus r$ cancels the effect of the randomness, making the associated switching activity dependent on both the shares of x and y. This results in the leakage of both secret variables.

Recombinations resulting in side-channel leakage can also occur between different instruction opcodes. For example, the instruction I_7 computing `AND` of x_1 and y_1 stores the result in D_6. The subsequent instruction I_8 performs `OR` of x_1 and y_0, inducing a transition in D_6 that depends on the values of y_0 and y_1, causing a leakage of the secret variable y.

Side-channel leakage may also occur due to the control flow. For example, consider instructions I_8 and I_{10}, which are distinct and non-consecutive, and which access the same microarchitectural element D_7. Assuming that no intermediate instruction overwrites this, the data written by I_8 remains in D_7 until I_{10} executes and overwrites it. Thus, the execution of I_{10} induces a transition in D_7: from the result of the `OR` operation to the result of the `XOR` operation. Since this transition depends on both operands x and y, it can lead to side-channel leakage that jointly reveals information about these values.

```
I1: mv   t1, x0
I2: mv   t2, x1
I3: and  t3, x0, y0
I4: and  t5, x1, y1
I5: xor  t4, t3, r
I6: xor  t6, t5, r
I7: and  t7, x1, y1
I8: or   t8, x1, y0
...
I10: xor t8, x0, y1
I11: and t9, t4, t7
```

Listing 1: Example instruction sequence which executes on sample circuit shown in Figure 1. We use this listing as a running example in the paper.

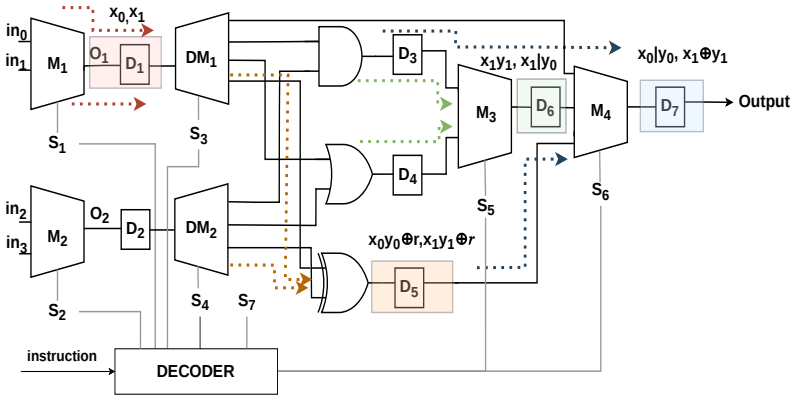


Figure 1: An example of a circuit that is part of a processor's execution unit. It performs MV, AND, XOR, and OR operations. We use this circuit as a running example to present microSCALE.

Importantly, the leakage is specific to the control flow and the order in which instructions traverse through the microarchitecture, highlighting that microarchitectural effects are not purely data-dependent but also execution-dependent.

These examples illustrate that even when shares are processed independently or masked with sufficient randomness, the instruction execution sequence within microarchitectural elements can induce transition-dependent recombinations. Such recombinations remain invisible to programmers as microarchitectural behavior is abstracted away from the software. Consequently, algorithmic-level masking alone cannot guarantee side-channel security without accounting for the detailed behavior of the underlying hardware. This highlights the need for a systematic evaluation of microarchitectural leakage, enabling a rigorous assessment of the side-channel resistance of cryptographic software and ensuring that its security guarantees hold on the target hardware.

4 The microSCALE Framework

We introduce microSCALE, a static analysis framework that identifies side-channel leakage caused by transitions in a processor's microarchitecture. Given a Cryptographic program Under Test (CUT), microSCALE identifies instructions that leak on a target processor and pinpoints the specific microarchitectural registers responsible for the leakage. Figure 2 presents a high-level overview of the microSCALE framework, which consists of five stages.

In the first stage, microSCALE leverages the processor's RTL to construct a Microarchitectural Dependency Graph (MDG). Each node in the MDG represents a microarchitectural element, such as a register, wire, or port, while edges encode the control dependencies that govern information flow within the processor. This graph-based representation enables systematic reasoning about data movement across the microarchitecture, thereby addressing challenge C1 (Section 1). Details of the graph construction are provided in Section 4.1.

The second stage of microSCALE tracks how each instruction's opcode and operands propagate through the microarchitecture. By traversing the MDG, this stage identifies the RTL expressions associated with each activated node along the instruction-specific path and outputs a table of these control flow deciding expressions. The construction of the processor-wide MDG and the extraction of instruction-specific control flow expressions are processor-specific and need to be performed only once per target processor. We present the details of this stage in Section 4.2.

Given the disassembly of a Crypto-program Under Test (CUT), the third stage evaluates the RTL expressions identified in the previous stage by assigning concrete operand values for each instruction. This evaluation enables precise tracking of data propagation through the microarchitecture and identifies the specific microarchitectural components that store or transform operand values during instruction execution. The resulting set of nodes and edges constitutes the instruction-specific path in the MDG, providing a fine-grained characterization of the information flow associated with each instruction. The details of this process are described in Section 4.3.

The fourth stage uses the instruction-specific path for each instruction in the CUT to determine how sensitive variables propagate through the microarchitecture during program execution. This instruction-level analysis tracks sensitive variables as they move, transform, and interact across shared microarchitectural resources, and captures this information

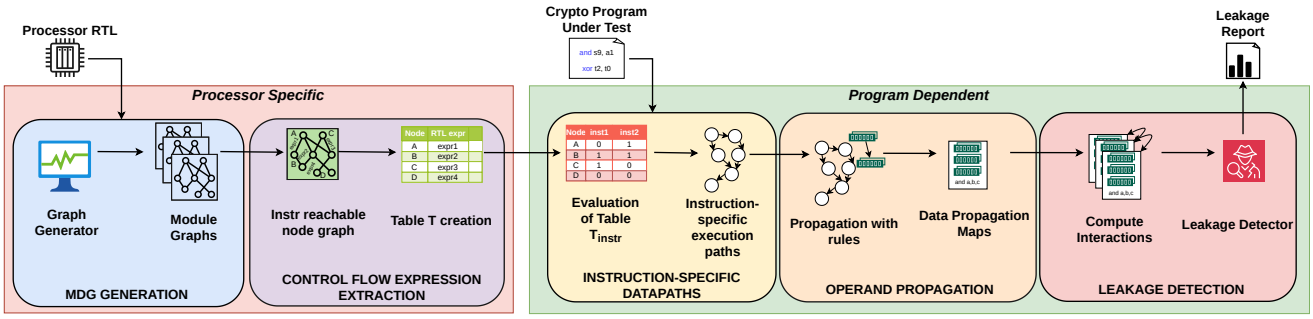


Figure 2: microSCALE comprises five stages: (Stage 1) generation of graphs from the processor's RTL; (Stage 2) extraction of instruction-reachable nodes and construction of a table of expressions for nodes reachable by each instruction; (Stage 3) evaluation of table using the Crypto-program Under Test (CUT) to derive instruction-specific execution paths; (Stage 4) propagation of data elements through the graphs to build per-instruction data-propagation maps; and (Stage 5) interaction analysis of these maps to identify leakage-causing instruction pairs and the responsible hardware elements.

in *Data Propagation Maps*. This data-propagation step is detailed in Section 4.4.

These data propagation maps are used in the fifth stage to identify interactions between sensitive variables and filter the interactions that cause side-channel leakage. For example, interactions where shares recombine and leak sensitive variables through side-channels. The third and fourth stages together address challenge C2 (Section 1). Challenge C3 is addressed by localizing the sources of leakage-causing interactions at the hardware as well as the software. In the software, instructions in the CUT that leak sensitive variables are identified, while in the hardware, corresponding microarchitectural elements that are responsible for the leakage is also identified. Details are provided in Section 4.5.

4.1 Microarchitecture Dependency Graph Generation

Taking the processor RTL as input, microSCALE extracts all inputs, outputs, wires, and registers of each module in the processor. The extracted information is then organized as a directed graph called the *Microarchitectural Dependency Graph* (MDG), where nodes correspond to microarchitectural elements and edges capture the dependencies between them.

Each node in the graph is labeled with a type (for example, wire, register, input, or output port), capturing the nature of the underlying hardware element within the microarchitecture. Graph generation is performed module by module, with input and output ports serving as the interface between modules. In addition, each node is annotated with an attribute that captures the corresponding RTL expression defining how the value of that node is computed from its parent nodes.

These expressions encoding the RTL logic are required when constructing the set of reachable nodes from the opcode (Section 4.2). Each edge in the graph is also annotated with a label representing the condition under which data can propagate along that edge.

Figure 3(a) shows the MDG corresponding to the circuit in Figure 1 with registers (D_1–D_7) and wires (W_0–W_6). Nodes are annotated with RTL expressions that they define. For example, in Figure 3(a), the expressions corresponding to nodes D_3 and D_5 are shown as representative instances, where $D_3 = W_1 \cdot W_4$ and $D_5 = W_3 \oplus W_6$. We omit the annotated RTL expressions for the other nodes in the figure for clarity and provide the complete details in the Appendix A. Edges capture the flow of values from input ports (in_0–in_3) to the output port (D_7). Information flows along an edge only when the annotated condition holds. For example, information flows from node D_3 to D_6 only when the condition on the corresponding edge (i.e., $s_5 = 0$) is True. Figure 3(b) presents the MDG corresponding to the decoder module shown in Figure 1. The graph captures the dependencies among instruction fields and the internal control signals generated by the decoder.

4.2 Extraction of Control Flow Expressions

In the second stage, microSCALE identifies RTL expressions associated with each node in the MDG that govern the information flow in the microarchitecture. To extract this information, microSCALE starts with the node in the MDG graph that fetches the instruction into the processor. With this node as source, it traverses the MDG using a Breadth First Search (BFS) algorithm to identify all reachable nodes that are influenced by the instruction. The set of reachable nodes capture the control signals in the processor that influence data flow and the operations that are performed on the data.

For example, in Figure 3(b), microSCALE starts with the u_{inst} node, which holds the instruction. BFS identifies seven reachable nodes $S_1, \ldots, S_7$, for which it records the corresponding RTL expression in a table, **T** (Table 1). For most of these nodes, the RTL expression depends exclusively on the instruction bits, that comprises of the opcode

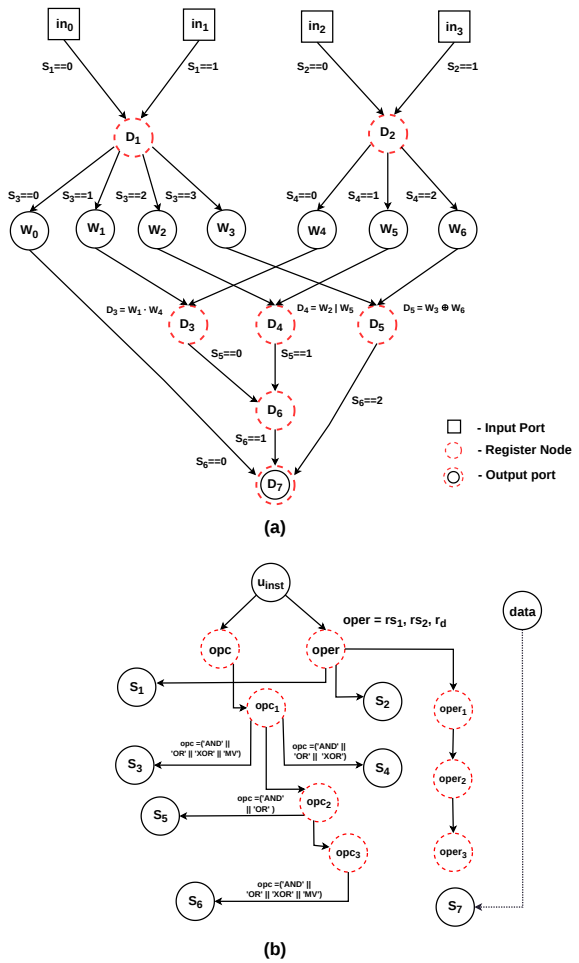


Figure 3: Microarchitectural Dependency Graph (a) for circuit in Figure 1 and (b) decoder module in Figure 1.

opc, source registers rs_1 and rs_2, a destination register rd, and values derived from the pipeline registers (for example, opc_1, opc_2, opc_3) in a typical Instruction Set Architecture (ISA), as shown in Figure 3(b). In few cases, nodes also depend on program specific data, for example, if a carry is produced due to an arithmetic operation, or a branch is taken due to a condition. For such nodes, execution flow cannot be determined statically, hence, microSCALE records a don't care, $\times$, in the table. Thus, table **T** records abstract symbolic expressions that can be derived from the node in the graph that holds the instruction.

4.3 Instruction Specific Microarchitectural Data Paths

Given the disassembly of a Crypto-program Under Test (CUT), microSCALE now assigns concrete values to each entry in the table **T**, for each instruction in the CUT. This

Table 1: Example instantiation of table **T** for the decoder in Figure 3 generated by the second stage of microSCALE.

Node	RTL Expression
S_1	$(rs_1 == 0)?0 : 1$
S_2	$(rs_2 == 0)?0 : 1$
S_3	$(opc_1 == \texttt{MV})?0 : ((opc_1 == \texttt{AND})?1 : ((opc_1 == \texttt{OR})?2 : ((opc_1 == \texttt{XOR})?3 : 0)))$
S_4	$(opc_1 == \texttt{AND})?0 : ((opc_1 == \texttt{OR})?1 : ((opc_1 == \texttt{XOR})?2 : 0))$
S_5	$(opc_2 == \texttt{OR})?1 : 0$
S_6	$(opc_3 == \texttt{MV})?0 : (((opc_3 == \texttt{AND}) \| (opc_3 == \texttt{OR}))?1 : ((opc_3 == \texttt{XOR})?2 : 0))$
S_7	$\times$

enables reasoning about data propagation paths through the microarchitecture, precisely identifying the microarchitectural components that store or transform operand data during the execution of an instruction.

Evaluating the RTL expressions in table **T** uses the instruction's encoding, relying on the instruction bits (*i.e.* opc, rs_1, rs_2, and rd). In contrast, expressions that depend on operand data values or contain don't-care entries cannot be evaluated statically, since the corresponding nodes take values determined by runtime data.

The evaluation is performed using a function *Eval*, which computes the values of entries in table **T** for a given instruction instr. All edges whose conditions involve don't-care signals are included in the data-flow path to capture the possible runtime values of these nodes. Formally, for each node v in **T**, with associated RTL expression expr, the instruction-specific evaluation result is stored in the table $\mathbf{T}_{\text{instr}}$:

$$\mathbf{T}_{\text{instr}}[v] = Eval(v, \text{expr}, \text{instr}, \mathbf{T}) \qquad \forall v \in \mathbf{T} \ .$$

Eval uses the instruction encoding and any previously computed values from $\mathbf{T}_{\text{instr}}$ required to assign concrete values to the node's expression. These signals determine the nodes and edges that are active in the MDG for a given instruction.

For example, consider the instruction xor a_5, a_1, a_2 . Let us assume that its ISA encoding has: $opc = \texttt{XOR}$, the source register indices are $rs_1 = 1, rs_2 = 0$, and the destination register index is $rd = 5$. The corresponding entries in Table 1 can be directly evaluated using values derived from the instruction encoding, resulting in $S_1 = 1$, $S_2 = 0$, $S_3 = 3$, $S_4 = 2$ and $S_6 = 2$.

The xor instruction we consider in the example, is similar in semantics to instruction I_5 in Listing 1. The evaluated signals for I_5 and other instructions in Listing 1 is shown in Table 2. The nodes marked as X in the table indicates that its value cannot be determined from the instruction encoding. Figure 4 illustrates the instruction-specific control-flow paths within the MDG shown in Figure 3(a), from which we observe that each instruction has a unique path corresponding to each input node in the MDG. These paths, taken together for an instruction is denoted $\text{MDGPath}_{\text{instr}}$. These paths serves as the input to the next stage that analyzes operand data propagation for each instruction in the CUT.

Table 2: Evaluation of table **T** for various instructions in Listing 1.

Signal	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_{10}	I_{11}
S_1	1	1	0	0	1	1	0	0	1	0
S_2	X	X	0	0	0	0	0	1	0	0
S_3	0	0	1	1	3	3	1	2	3	1
S_4	X	X	0	0	2	2	0	1	2	0
S_5	X	X	0	0	X	X	0	1	X	0
S_6	0	0	1	1	2	2	1	1	2	1

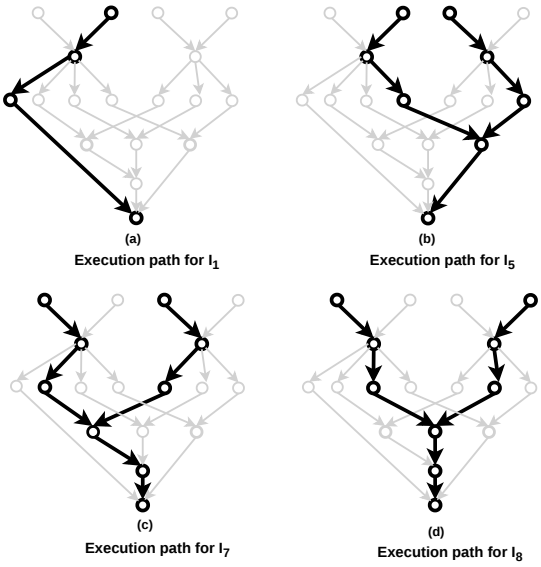


Figure 4: Instruction specific execution paths in Figure 3 for various instructions in Listing 1.

4.4 Operand Propagation through Instruction-Specific Path

A typical masked implementation of a cryptographic algorithm takes the secret shares and random values as input. As the program executes, information about the shares and randoms propagates from one instruction to another. Accordingly, the operands of the instructions hold inputs or values derived from them. The objective of this stage is to track operand propagation for each instruction in the CUT using the $\text{MDGPath}_{\text{instr}}$.

`microSCALE` employs a custom data structure referred to as a *data propagation map* that maps each node in the MDG to the set of inputs (shares and randoms) that may influence the value computed by that node's expression. The set of inputs is encoded as a bit vector, where each bit corresponds to a specific input. A bit is set to 1 if the value computed by the node is statistically dependent on the corresponding input, and 0 otherwise.

We motivate the operand propagation using the code snippet in Listing 1, which takes as inputs the secret shares of x and y, namely x_0, x_1, y_0, y_1 and a random value r. Accordingly, the data propagation map assigns to each node a bit vector with one bit for each input: (x_0, x_1, y_0, y_1, r), in the specified order. Formally, we define the map

$$\mathbf{M} : v \rightarrow \vec{B} \ ,$$

which associates each node v in the MDG with a bit vector comprising of input secret shares and randoms. We define two functions on the bit vector, namely, $\mathcal{S}(\mathbf{M}(v))$ and $\mathcal{R}(\mathbf{M}(v))$, that respectively outputs the corresponding secret shares and randoms corresponding to the bit vector of node v.

For a given instruction, the data propagation map is initialized at the input nodes of the $\text{MDGPath}_{\text{instr}}$, which holds the operand values. For all other nodes, the corresponding bit vector is set to 0. If the operand corresponds to an input (share or random), the bit corresponding to that input is set to 1, while all other bits are set to 0. On the other hand, if the operand corresponds to a value computed by a preceding instruction (e.g., $t_1 = x_0 \cdot y_0$), the operand inherits the bit vector encoding of the input set computed for that intermediate value.

We illustrate the initialization at the input nodes using the instruction I_4 from Listing 1: `and` t_5, x_1, y_1 . The first operand corresponds to the share x_1, while the second operand corresponds to the share y_1. The $\text{MDGPath}_{\text{instr}}$for `and` has two input nodes (Figure 4(c)). The first input node in_0 receives the share x_1 and its bit vector is initialized as $\mathbf{M}(\text{in}_1) = (0,1,0,0,0)$. The secret shares and randoms for this bit vector is $\mathcal{S}(\mathbf{M}(v)) = (0,1,0,0)$ and $\mathcal{R}(\mathbf{M}(v)) = (0)$. The second input node in_2 receives the share y_1 and its bit vector is initialized as $\mathbf{M}(\text{in}_2) = (0,0,0,1,0)$.

As operands propagate along the instruction-specific path $\text{MDGPath}_{\text{instr}}$, they are not always forwarded unchanged. Instead, they pass through nodes whose annotated expressions transform these values. Consider node D_3 in Figure 3(a). This node is annotated with the expression $W_1 \cdot W_4$; accordingly, the values produced at D_3 depend on the masking semantics of `and` operation while processing the expression $W_1 \cdot W_4$ [4]. For example, at node D_3, `and` operation combines the inputs x_1 and y_1. Since the output of D_3 depends on both x_1 and y_1, the data propagation map associated with D_3 has the entries corresponding to x_1 and y_1 marked as 1.

While this case is straightforward, deeper in the program, the operands of an instruction may themselves be complex expressions of the inputs. For example, consider the propagation for instruction I_{11} in Listing 1, where the first operand is $(x_0 \cdot y_0 \oplus r)$ and the second operand is $(x_1 \cdot y_1)$. At node D_3, the `and` operation combines these two operands. A naïve information flow tracking system would set all bits in the bit vector of D_3 to 1. However, we observe that the random value r masks the product of $x_0 \cdot y_0$ in the first operand. Consequently, only the shares x_1 and y_1 influence the output at D_3, and only their corresponding entries are marked as 1 in the data propagation map of D_3.

To correctly track the evolution of program inputs during propagation, `microSCALE` defines a set of transformation rules for evaluating node expressions along the $\text{MDGPath}_{\text{instr}}$. Each rule specifies how the bit vector for a node is derived from the corresponding bit vectors of its input nodes, keeping in context the input shares and randoms. The *transformation rules* are as follows:

Direct forwarding nodes: Nodes that forward data without modifying its value propagate the associated data propagation information unchanged from input to output. This category includes register writes, pipeline registers, buffers, and wires. For a source node v_{src} and an destination node v_{dest}:

$$\mathbf{M}(v_{\text{dest}}) = \mathbf{M}(v_{\text{src}}) \ .$$

For example, consider instruction I_3 in Listing 1(`and` t_3, x_0, y_0). The data from in_0, which holds x_0, is propagated to D_1, resulting in $\mathbf{M}(D_1) = (1,0,0,0,0)$. Similarly, the data from in_2, which holds y_0, is propagated to D_2, resulting in $\mathbf{M}(D_2) = (0,0,1,0,0)$.

Negation (NOT): The `not` operation does not alter the statistical dependence of the processed value on the output. As a result, the data propagation information is propagated unchanged from the source to the destination, similar to direct forwarding nodes.

$$\mathbf{M}(v_{\text{dest}}) = \mathbf{M}(v_{\text{src}}) \ .$$

XOR and XNOR: Under Boolean masking, combining identical shares or random values cancels their contribution. Accordingly, `xor` and `xnor` operations remove any share or random value that appears on both source nodes, while propagating those that appear on exactly one source. Let v_{src_1} and v_{src_2} denote the two source nodes, with v_{dest} denoting the destination node.

$$\mathbf{M}(v_{\text{dest}}) = \mathbf{M}(v_{\text{src}_1}) \oplus \mathbf{M}(v_{\text{src}_2}) \ .$$

Consider instruction I_{10} in Listing 1: (`xor` t_8, x_0, y_1). The bit vector from in_1 gets forwarded to D_1 and then to W_3. Similarly, the bit vector from in_2 is copied into D_2 and W_6. The data paths of these source nodes converge at node D_5, whose expression computes the `xor` of signals W_3 and W_6. Applying the XOR propagation rule to D_5 yields:

$$\begin{aligned}\mathbf{M}(D_5) =& \mathbf{M}(W_3) \oplus \mathbf{M}(W_6) \\ =& (1,0,0,0,0) \oplus (0,0,0,1,0) \\ =& (1,0,0,1,0) \ .\end{aligned}$$

All other operations: In these cases, the data propagation information associated with the destination node is obtained as the union of the information associated with the source nodes, with the contribution of each input determined by the masking effect of any randoms present. The rules for propagation pertaining to randoms are given as follows:

1. If neither source node is associated with random values, the bit vector associated with the destination node is the union of the bit vectors associated with the source nodes.
2. If exactly one source node is associated with a random value, the data propagated from that source is masked by the random and therefore does not propagate to the destination node; the bit vector associated with the other source node is copied to the destination node.
3. If both source nodes are associated with the same randoms, the random no longer masks their combined effect. Consequently, the bit vectors associated with both source nodes are propagated to the destination node.
4. If the two source nodes are associated with distinct randoms, no information propagates to the destination node.

Overall, the set of secret shares associated with the destination node is given by $\mathcal{S}(\mathbf{M}(v_{\text{dest}})) =$

$$\begin{cases} \mathcal{S}(\mathbf{M}(v_{\text{src}_1})) \cup \mathcal{S}(\mathbf{M}(v_{\text{src}_2})), & \text{if } \mathcal{R}(\mathbf{M}(v_{\text{src}_1})) = \varnothing \wedge \mathcal{R}(\mathbf{M}(v_{\text{src}_2})) = \varnothing, \\ \mathcal{S}(\mathbf{M}(v_{\text{src}_2})), & \text{if } \mathcal{R}(\mathbf{M}(v_{\text{src}_1})) \neq \varnothing \wedge \mathcal{R}(\mathbf{M}(v_{\text{src}_2})) = \varnothing, \\ \mathcal{S}(\mathbf{M}(v_{\text{src}_1})), & \text{if } \mathcal{R}(\mathbf{M}(v_{\text{src}_1})) = \varnothing \wedge \mathcal{R}(\mathbf{M}(v_{\text{src}_2})) \neq \varnothing, \\ \mathcal{S}(\mathbf{M}(v_{\text{src}_1})) \cup \mathcal{S}(\mathbf{M}(v_{\text{src}_2})), & \text{if } \mathcal{R}(\mathbf{M}(v_{\text{src}_1})) = \mathcal{R}(M(v_{\text{src}_2})), \\ 0, & \text{otherwise.} \end{cases}$$

For all of these other operations, no randoms propagate to the destination node. Thus, $\mathcal{R}(\mathbf{M}, n_{\text{dest}}) = 0$ for these operations. We illustrate operand data propagation corresponding to using the instruction I_{11} from Listing 1, `and` t_9, t_4, t_7 . The source node in_0 contain $t_4 = x_0 \cdot y_0 \oplus r$ and the other source node in_2 has $t_7 = x_1 \cdot y_1$. The source nodes are initialized with:

$$\mathbf{M}(\text{in}_0) = (1,0,1,0,1), \qquad \mathbf{M}(\text{in}_2) = (0,1,0,1,0).$$

Figure 5 highlights the propagation paths of the data originating from the two input nodes. The value from in_0 propagates through the direct-forwarding node D_1 and is copied to node W_1. Similarly, the value from in_2 propagates through node D_2 and reaches W_4. Hence, the entries corresponding to D_1, W_1, D_2 and W_4 are as follows:

$$\begin{aligned}\mathbf{M}(D_1) &= (1,0,1,0,1), \qquad \mathbf{M}(W_1) = (1,0,1,0,1), \\ \mathbf{M}(D_2) &= (0,1,0,1,0), \qquad \mathbf{M}(W_4) = (0,1,0,1,0).\end{aligned}$$

The two paths converge at node D_3, whose expression computes the `and` of signals W_1 and W_4. As `and` belongs to "All other operations" category, the randoms in each of the inputs decides the propagation to the output. As per Rule 2,

$$\begin{aligned}&\mathcal{S}(\mathbf{M}(D_3)) = \mathcal{S}(\mathbf{M}(W_4)), \\ &\quad \because \mathcal{R}(\mathbf{M}(W_1)) = (1) \wedge \mathcal{R}(\mathbf{M}(W_4)) = \varnothing \ ,\end{aligned}$$

and $\mathcal{R}(\mathbf{M}(D_3)) = 0$, resulting in $\mathbf{M}(D_3) = (0,1,0,1,0)$, indicating that only the secret shares x_1 and y_1 propagate to

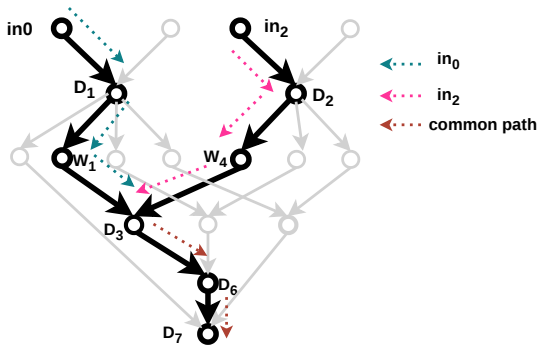


Figure 5: Propagation path for each input nodes corresponding to instruction I_{11} from Listing 1.

Table 3: Data propagation maps instructions in Listing 1 for each nodes in the MDG present in Figure 3(a). Colored cells denote leakage due to recombinations – blue leaks x, green leaks y, and red leaks both x and y. Commas in the bit vectors are omitted for brevity. Grayed text indicates that a node is unaffected by an instruction's execution.

Instr	D_1	D_2	D_3	D_4	D_5	D_6	D_7
I_1	(10000)	(00000)	(00000)	(00000)	(00000)	(00000)	(10000)
I_2	(01000)	(00000)	(00000)	(00000)	(00000)	(00000)	(01000)
I_3	(10000)	(00100)	(10100)	(00000)	(00000)	(10100)	(10100)
I_4	(01000)	(00010)	(01010)	(00000)	(00000)	(01010)	(01010)
I_5	(10100)	(00001)	(01010)	(00000)	(10101)	(01010)	(10101)
I_6	(01010)	(00001)	(01010)	(00000)	(01011)	(01010)	(01011)
I_7	(01000)	(00010)	(01010)	(00000)	(01011)	(01010)	(01010)
I_8	(01000)	(00100)	(01010)	(01100)	(01011)	(01100)	(01100)
I_{10}	(10000)	(00010)	(01010)	(01100)	(10010)	(01100)	(10010)
I_{11}	(10101)	(01010)	(01010)	(01100)	(10010)	(01010)	(01010)

the destination of D_3. Further, the values are propagated to D_6 and D_7, leading to the entries in $\mathbf{M}$ as $\mathbf{M}(D_6) = (0,1,0,1,0)$,$\mathbf{M}(D_7) = (0,1,0,1,0)$.

The same procedure is applied to all instructions in Listing 1 to derive their corresponding operand data propagation maps. Each row in Table 3 shows the data propagation map for each instruction in the listing, while the columns represent the data propagation activity across the registers D_1 to D_7. These maps are provided as input to the next phase for pinpointing instructions and regions in the microarchitecture that result in transitional leakage of the sensitive data.

4.5 Leakage Detection

Transitions occur in microarchitectural elements when the instructions in the CUT execute. Since these microarchitectural elements are represented by nodes in the MDG, transitions can be determined by tracking activity across the data propagation maps of the instructions. For example, each column in Table 3 represents the *trace* of activities in a register across the instruction execution. Moving from one instruction to another tracks the transitions in the register. The leakage detection logic in `microSCALE` checks whether a transition results in a recombination of shares that are associated with a secret input.

Detecting transitional leakage proceeds in two steps. First, for each node v in the MDG, `microSCALE` constructs the trace of v across the N instructions of the masked cryptographic program and computes transitions between successive entries, i.e., between entries i and $i+1$ for $i \in \{1,\ldots,N-1\}$, totaling $N-1$ computations. Second, `microSCALE` analyzes these transitions to identify those that produce recombinations of secret input shares, resulting in transitional leakage at node v. Each such transition is attributed to a pair of instructions: the instruction $i+1$ where the recombination occurs and the prior instruction that last updated v before $i+1$.

We illustrate transitional leakage using the data propagation maps summarized in Table 3. Consider the bit vectors associated with D_1 between instructions I_1 and I_2, given by (10000) and (01000), respectively. The transition between these vectors is $(10000) \oplus (01000) = (11000)$, corresponding to a recombination of the shares of x, namely x_0 and x_1, as discussed in Section 3 and highlighted in Figure 1. Similarly, the bit vectors of D_5 between I_5 and I_6 result in a leakage of shares of both x and y, despite the presence of an entry for random in their respective bitmaps. Transitional leakage can arise between non-consecutive and different-type instructions; e.g., activity in D_6 between I_8 and I_{11} leaks input y.

4.6 Security Coverage and Time Complexity

Security Coverage. We characterize the security coverage of `microSCALE` by analyzing each stage of the framework. In Stage 1, MDG is generated directly from the RTL, enabling the framework to capture the structural behavior of the hardware implementation. Stage 2 precisely identifies the instruction-dependent control behavior within the microarchitecture by analyzing the control signals and execution paths associated with each instruction. Stage 3 derives operand propagation paths over the identified microarchitectural components to capture how operands and intermediate values propagate during execution. In Stage 4, the framework applies masking-aware propagation rules to track how secret shares and random values flow through the derived paths. Finally, Stage 5 analyzes transitions across these propagation paths to identify potential sources of transitional leakage.

Since `microSCALE` employs static analysis, the framework reasons about feasible propagation and control-flow behavior without relying on concrete runtime operand values. Accordingly, multiple runtime executions are represented through a unified propagation abstraction during analysis. This helps scale to large and complex processors; however, the abstraction may lead to some cases where `microSCALE` may miss

detecting leakages, leading to false negatives. We expect such corner cases to occur rarely, and we did not observe any such cases in our evaluation.

Time Complexity. Let the MDG contain V nodes and let the input program contain N instructions. Stage 1 constructs the MDG from the RTL in $O(V)$, and Stage 2 performs a BFS over the MDG starting from the instruction node to identify all reachable control signals. Since BFS visits each node and explores its connections, it runs in $O(V^2)$. Stage 3 performs per-instruction control-flow identification over the graph, requiring $O(N \cdot V)$. Stage 4 then propagates input shares along the derived paths, which also requires $O(N \cdot V)$. Finally, Stage 5 evaluates transitions at each node along the propagated paths across successive instructions, requiring $O(N \cdot V)$. Since Stages 1 and 2 are processor-dependent, `microSCALE` incurs a per-processor cost of $O(V^2)$, whereas analyzing a CUT on the target processor requires $O(N \cdot V)$.

5 Implementation Considerations

While the preceding analysis illustrates the methodology of `microSCALE`, an automated implementation must account for key microarchitectural features. Operand forwarding, branching, loops, and related microarchitectural optimizations influence how values propagate and where transitions occur during execution, while masking introduces both linear and non-linear leakage behaviors that must be modeled carefully. The remainder of this section outlines the key implementation considerations for accurate analysis.

Operator forwarding. Operator forwarding allows an instruction to read a value directly from an intermediate pipeline stage instead of from the architectural register file. Although architecturally invisible, this changes the physical source from which an operand is obtained. For example for the set of instructions `add` x_3, x_1, x_2, `add` x_4, x_3, x_5 , the value of x_3 used by the second instruction may come either from the register file or from a pipeline register holding the result of the first instruction, depending on the microarchitecture pipeline design.

Modern in-order pipelines commonly implement forwarding paths from multiple stages (e.g., execute, memory, and write-back). As a result, the exact datapath traversed by an operand cannot be determined from the instruction stream alone. `microSCALE` therefore models pipeline timing and hazard resolution to identify the concrete producer that supplies each operand: when forwarding is taken, it initiates data-flow tracking from the corresponding pipeline register or bypass-network source; otherwise, it initiates tracking from the architectural register file port.

Handling of loops and branching. Typical cryptographic programs contain loops with finite number of iterations. `microSCALE` handles these loops by tracking iteration indices and reporting the corresponding instruction instances in the leakage report, thereby allowing leakage to be associated with the specific loop iteration in which it occurs.

Operand-dependent branching, however, introduces additional complexity in leakage analysis, as different execution paths may exhibit distinct microarchitectural propagation behaviors. If there is one such branch operation in the program, we analyze each branch path independently and combine the resulting leakage reports to capture leakage arising across different execution paths. While such simple branch structures can be handled in this manner, branch-intensive execution patterns can substantially increase the number of feasible execution paths that must be analyzed, thereby significantly increasing the complexity of the analysis. Addressing such cases is left for future work.

Representation of share information. The choice of representation for annotating data elements directly affects both the efficiency of dataflow analysis and the accuracy of leakage detection. Masked implementations involve secret shares, random values, and intermediate results from arithmetic and logical operations. The representation must preserve enough semantic information to distinguish how shares combine and transform. For instance, $x_1 \oplus y_1$ differs from $x_1 \cdot y_1$ because they interact differently with other shares: a linear expression cancels when xored with the same share, $x_1 \oplus y_1 \oplus x_1 = y_1$, whereas a non-linear expression does not, $x_1 \cdot y_1 \oplus x_1 \neq y_1$.

Failing to preserve this distinction can lead to incorrect leakage conclusions. To address this, we classify operations as *linear* or *non-linear*, and the *share propagation map* maintains separate representations for these classes to track share transformations accurately during analysis. Consider the instruction sequence from Listing 1. The instruction pair I_7, I_{10} interacts at the register D_7. Inputs to the data propagation map are x_0, x_1, y_0, y_1, r. The *linear* map contains one entry for each input, while the *non-linear* map captures non-linear combinations of the input shares. The maps for the instructions I_7 $(x_1 \cdot y_1)$ and I_{10} $(x_0 \oplus y_1)$ are:

$$x_1 \cdot y_1: \quad \text{linear } [0,0,0,0,0], \quad \text{non-linear } [0,1,0,1].$$
$$x_0 \oplus y_1: \quad \text{linear } [1,0,0,1,0], \quad \text{non-linear } [0,0,0,0].$$

If further in the circuit, if there exists a register where these values are xored with x_0, the maps transform as follows:

$$(x_1 \cdot y_1) \oplus x_0: \quad \text{linear } [1,0,0,0,0], \quad \text{non-linear } [0,1,0,1].$$
$$(x_0 \oplus y_1) \oplus x_0: \quad \text{linear } [0,0,0,1,0], \quad \text{non-linear } [0,0,0,0].$$

6 Evaluation

In this section, we evaluate `microSCALE` on a full round cipher and representative S-box/masking gadgets, reporting detected leakage points and attributing them to the responsible

hardware structures and program locations. Our experiments use the open-source Rocket Chip processor [1], an in-order 5-stage RISC-V core: Instruction Fetch (**IF**), Instruction Decode (**ID**), Execute (**EX**), Memory Access (**MEM**), and WriteBack (**WB**). We focus on first-order leakage, and include a second-order example to illustrate extensibility.

Evaluation Setup. `microSCALE` first converts RocketChip's RTL into its graphical representation as described in Section 4.1. The generated graphs are used to obtain instruction-specific data paths (refer Section 4.2). The initial node for this step corresponds to the Instruction Buffer signal `ibuf_io_inst_0_bits_inst_bits`. For each opcode, this node is set to the corresponding RISC-V encoding [27], and `microSCALE` constructs an opcode-specific subgraph for each opcode in the ISA. In RocketChip, the subgraph is constructed using only the opcode fields of the instruction word; operand fields such as `rs1`/`rs2` are ignored. This captures the control decisions that steer the main datapaths, and allows opcode-specific subgraphs to be computed once per processor and reused across programs. For Rocket, MDG construction and opcode-specific subgraph generation take about 2.5 hours, and it is a one-time cost for the processor.

After the opcode-specific subgraphs are available, `microSCALE` performs operand propagation as described in Section 4.4. For each instruction, propagation begins at the register-file read ports and follows the selected datapaths through the microarchitecture. For example, for ALU operations, the two source operands are read from `rf_id_rs_MPORT_data` and `rf_id_rs_MPORT_1_data` and propagates further in to the microarchitecture. After the completion of operand propagation, `microSCALE` takes the share-propagation map at the ALU result wire `alu_io_out` and uses it to store it back to the destination register, so that subsequent instructions that uses the resultant register value can reuse the propagated state. However, with operand forwarding, Rocket changes the starting point of operand propagation: a dependent instruction may source its operand from the forwarding paths `mem_reg_wdata` or `wb_reg_wdata`, rather than from the register-file read ports.

Results on masked crypto implementations. We evaluate `microSCALE` on the masked primitives and crypto implementations summarized in Table 4. The benchmarks cover small Boolean gadgets, standalone S-boxes, and a full cipher round. For gadgets, we include masked AND/XOR building blocks: ISW-style masking (`ISW-AND`, `ISW-XOR`) [20] and Domain-Oriented Masking for AND (`DOM-AND`) [19], along with two re-randomization steps, `Refresh` and `Locality Refresh`, that re-mask intermediate shares [11, 12]. For S-boxes, we evaluate masked implementations of the GIFT S-box [3], ASCON S-box [15], PRESENT S-box [7], and an AES S-box in the Rivain-Prouff style (`AES S-box (RP)`) [28]. To capture end-to-end behavior, we also evaluate one full round of GIFT (`GIFT One Round`). Most results focus on first-order leakage, and we additionally include a second-order instance of GIFT S-box (`GIFT S-box 2nd order`) to show that the same workflow extends to higher orders.

Analysis of ASCON, PRESENT, AES S-box results. Table 4 highlights that leakage grows sharply as implementations move from small gadgets to larger pieces of crypto code. Across the first-order gadget benchmarks, `microSCALE` reports only 23-58 transition leakages, with an average of 33.4. For the S-box benchmarks, the number of transition leakages increases to the hundreds or thousands, with an average of 1170.3 across the four S-boxes. At the full-round level, leakage increases again: `GIFT One Round` reports 6709 leakages, which is about 7.8× higher than the standalone `GIFT S-box` (858). This shows that leakage does not only come from the core nonlinear operation, but also accumulates from the surrounding data movement and intermediate state that appear when the S-box is integrated into a full round.

For each cipher, we compute the percentage of transitional leakages that arise from the various pipeline stages. These percentages are consistent across most benchmarks, with leakage dominated by **WB**. This stage attribution is obtained by mapping each leakage point to the pipeline stage of the microarchitectural register where it is observed (Table 6 in the Appendix). Averaged over the first-order rows, **WB** accounts for about 61% of leak points, and this skew is even stronger for the full-round experiment (`GIFT One Round`: 80.3% in **WB**). In contrast, **MEM** remains a smaller share (typically around 10-15%) and **EX** accounts for most of the remainder. The **Other** category is usually near zero, but becomes visible for `AES S-box (RP)` (8.8%) and, to a smaller extent, for `GIFT One Round` (0.5%), indicating that some leakage arises outside the main execute/memory/write-back datapath for these cases.

We hypothesize that the predominance of leakage in the WB stage arises from its role as a common convergence point, where results from many different instruction types are funneled through a small set of shared registers and multiplexers. The repeated reuse of these physical paths increases switching activity and overlap between sensitive values, making the WB stage particularly susceptible to observable leakage. Interestingly, the additional leakage observed in the Other category can be traced to signals within the CSR module, indicating activity outside the main EX/MEM/WB datapath.

The offset column provides a view of locality. Across the first-order benchmarks, the mean offset is 1.55 (median 1.58), and the range is narrow (1.15-2.18), meaning that leakage is typically observed within one to two instructions after the source. This short distance helps localize root causes in practice, since the leaking instruction and the producing instruction usually lie close together in the trace.

Analysis of GIFT S-box results. Figure 7 takes a closer look at leakage in the 4-bit masked GIFT S-box. For one S-box execution, it shows (i) which bits contribute to leakage, (ii)

which pipeline stages the leakage is linked to, and (iii) how many instructions later the leakage is observed. Together, these plots help pinpoint the instruction ranges that cause leakage, identify where it appears in the pipeline, and show whether it persists beyond the producing instruction. A clear example appears around instructions 72-75, which are a sequence of AND operations (`and` s_{11}, a_0, a_3, `and` s_{10}, a_0, a_7, `and` s_9, a_4, a_3, `and` s_8, a_4, a_7) .

This window forms a strong leakage hotspot: the total leakage rises, the stacked bars are dominated by contributions from Bit 1 and Bit 3 (visible as prominent dark-blue and dark-green segments in Figure 7), and most of the leakage is attributed to the **EX** and **WB** pipeline stages. A few instructions later, the `EXOR` sequence at instructions 78-81 (`xor` s_{11}, s_{11}, s_7, `xor` s_{10}, s_{10}, s_7, `xor` s_{10}, s_{10}, s_9, `xor` s_{10}, s_{10}, s_8) shows a much smaller footprint in both the bit-level and pipeline-stage views.

The offset plot strengthens this comparison. Near the AND instructions, several leakage points have non-zero offsets, meaning values produced by the ANDs can still lead to leakage one or two instructions later. Near the XOR instructions, offsets are fewer and shorter, showing that the leakage is not only smaller but also less persistent. Overall, Figure 7 suggests that leakage in this S-box is driven by short instruction windows, and that it is mostly linked to **EX/WB** activity and offsets of one to two instructions.

FPGA setup and results for GIFT cipher. To validate `microSCALE`'s results, Test Vector Leakage Assesement (TVLA) [18] is performed on a one-round masked GIFT implementation using 1 million *EM traces*. The design is deployed on a Nexys Video FPGA board [14] (Xilinx Artix-7 XC7A200T), and EM signals are captured with a Teledyne LeCroy HDO4104 oscilloscope and a Langer RF-U 5-02 passive near-field probe, amplified by a Langer PA 303 preamplifier. Sampling is performed at 500 kS/s (every 2 μs), yielding roughly 10 samples per processor cycle at 50 MHz. Each trace contains 50,000 samples, and acquisition is triggered using two board I/O pins that mark the start and end of the trace window.

TVLA is first applied to the original (unhardened) masked code to establish a baseline. `microSCALE` is then run on the same program to obtain the leakage report containing the leakage points and the corresponding instruction interactions responsible for leakage. Note that as explained in Sections 4.5, each reported leakage corresponds to a recombination of secret shares that violates the masking assumptions. Hence, the identified leakages could be exploited using standard side-channel attacks, such as Differential Power Analysis (DPA) [21]. Based on the leakage report, a hardened version of the one-round GIFT implementation is constructed, and TVLA is re-run on the hardened code. We target low-overhead hardening to demonstrate that the leakage identified by our tool is meaningful and actionable. The original GIFT

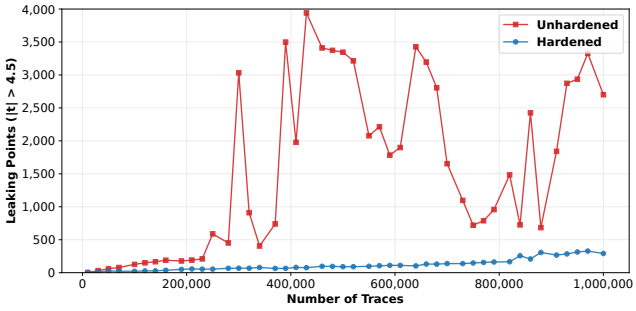


Figure 6: Leakage growth of GIFT One round Unhardened vs. Hardened.

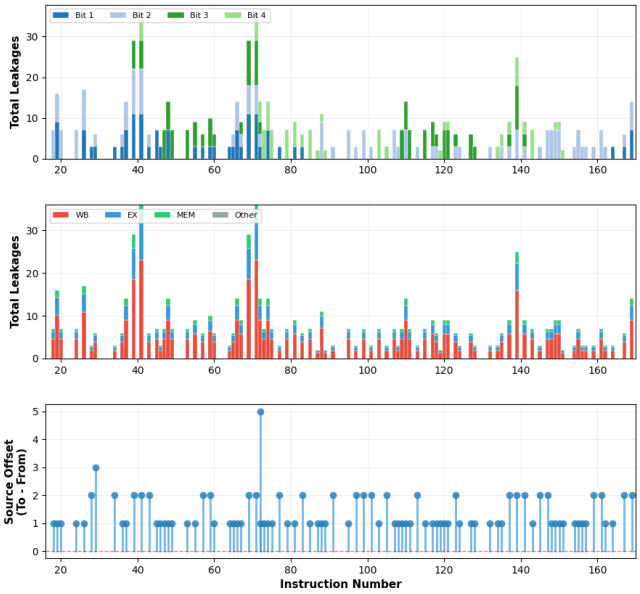


Figure 7: Leakage analysis for the GIFT S-box: (top) leakage contributions from individual bit positions of the secret shares, (middle) leakage across pipeline stages, and (bottom) source–destination separation in leaking instruction pairs.

implementation has 760 instructions. We hardened the code with minor hardware and software-level hardening techniques, including instruction rearrangement and dummy instruction insertion. We added 87 dummy instructions (≈11% overhead), causing the number of leakage points to drop from 6,709 to 986 (85% reduction), as shown in Table 4.

Figure 6 plots the number of leaking time samples ($|t| > 4.5$) with respect to the trace count. The unhardened implementation exhibits a peak of approximately 3×10^3 leaking points for 10^6 traces, whereas the hardened implementation remains below 300 leaking points over the same number of traces. This corresponds to at least an 90.3% reduction in TVLA leakage points, showing that `microSCALE`'s reported leakage points translate to observable EM leakage and that addressing them measurably reduces leakage.

Table 4: For each cipher, this table reports the total number of instructions; the percentage of leakage-causing instruction pairs; the average offset between leaking pairs; the total transitional leakage; stage-wise percentage breakdown of observed transitional leakages across processor stages; and the runtime of `microSCALE` on the cipher.

Masked Cipher	# Total Instructions	% Leakage Pairs	# Transition Leakages	MEM	WB	EX	Other	Avg. offset distance	Time (s.)
ISW-AND [20]	22	36%	31	12%	56%	32%	0%	1.48	8.03
DOM-AND [19]	22	36%	58	12%	72%	16%	0%	1.68	11.64
Refresh [12]	12	33%	24	12.5%	62.5%	25%	0%	1.71	4.37
Locality Refresh [11]	13	23%	23	11.8%	64.7%	23.5%	0%	2.18	4.48
ISW-XOR [20]	14	28%	31	16.21%	51.6%	32.19%	0%	1.68	4.34
GIFT S-box [3]	169	52%	858	11.2%	64.1%	24.7%	0%	1.42	115.40
ASCON S-box [15]	217	60%	2428	15.2%	55%	29.7%	0%	1.15	132.74
PRESENT S-box [7]	107	64%	1324	13.9%	57%	29.1%	0%	1.25	63.29
AES S-box (RP) [28]	43	23%	71	14%	49.1%	28.1%	8.8%	1.72	23.88
GIFT One Round [3]	760	80%	6709	5.4%	80.3%	13.8%	0.5%	1.21	860.73
GIFT One Round hardened	847	12.6%	986	13.8%	55.2%	31%	0%	1.50	874.73
GIFT S-box 2nd order	1096	22.4%	253	28.6%	14.3%	57.1%	0%	1.34	234.02

7 Related Work

While many existing tools target leakage arising from the microarchitectural execution of cryptographic software, they differ in how this leakage is modeled and analyzed. A wide range of techniques have been proposed to detect or mitigate such leakage, spanning simulation-based, formal, and hybrid approaches.

Simulation-based instruction-level analysis: ELMO [24] is one of the earliest tools to analyze side-channel leakage in cryptographic software running on microcontrollers. It uses linear regression models trained from real power and EM measurements on ARM Cortex-M devices to predict leakage from instruction sequences. ELMO identifies leakage caused by interactions between consecutive instructions, but it does not capture effects caused by non-adjacent instructions or by deeper microarchitectural reuse.

ROSITA [30] extends ELMO through ELMO*, which models non-consecutive instruction interactions and combines trace simulation with Test Vector Leakage Assessment (TVLA) to verify whether detected leakage is mitigated. ROSITA also supports automatic rewriting of leaking instruction sequences. However, both ELMO and ROSITA detect only leakage associated with single intermediate variable and rely on statistical testing of simulated traces. ROSITA++ [29] further extends this approach to analyzing combinations of multiple intermediate values. Despite this improvement, all ELMO-based tools depend on learned or statistical leakage models and have only been demonstrated on simple microcontrollers such as the ARM Cortex-M0. Their reliance on trace simulation makes scaling to modern pipelined processors difficult.

Microarchitectural simulation and RTL-based analysis: PARAM [16] analyzes side-channel leakage at the RTL level using behavioral simulation, which runs cryptographic software with varying inputs. For each microarchitectural module in the RTL, it computes the Side-channel Vulnerability Factor (SVF) [13], which uses Hamming-distance correlations between sensitive data and internal signal activity. PARAM can localize leakage to specific hardware blocks but relies on Hamming-based leakage models and does not reason about masking or higher-order leakage.

Telescope [22] extends this line of work by performing hierarchical leakage assessment across three abstraction levels, namely architecture, microarchitecture, and gate, using power estimation based on toggle on a gate-level netlist. It employs TVLA-based statistical testing to automatically identify leakage across different levels of abstraction using timing analysis on the netlist. It identifies both the cycle responsible for leakage and the specific microarchitectural components that causes it. However, like PARAM, Telescope is simulation-based, and its leakage detection capability is bounded by the execution scenarios exercised during testing. Consequently, leakage sources not covered by the selected test vectors may remain undetected. In contrast, unlike Telescope's approach of using clock cycles to pinpoint vulnerable instructions, `microSCALE` statically analyzes the RTL to precisely identify the pairs of instructions responsible for leakage, thereby facilitating targeted and effective software-level hardening.

Instruction-set simulators such as SAVRASCA [32] generate power traces from binaries using Hamming-weight and Hamming-distance models. These tools capture transition-based leakage but rely on simplified models and are limited to small microcontrollers.

Formal verification of masked software: Formal tools such as MaskVerif and its successors [4] use SMT-based reasoning to prove that no set of up to d intermediate values depends on secrets under abstract leakage models that may include transitions and glitches. These tools provide strong guarantees for masked code but do not model how values flow through real processor pipelines.

PROLEAD_SW [34] bridges this gap by extending hardware-level formal verification to masked software on ARM Cortex-M processors. It symbolically simulates a generic microarchitectural leakage model to identify violations of probing security. While effective at finding vulnerabilities missed by software-only tools, its generic model leads to conservative results, it is limited to specific ARM cores, and its exponential complexity restricts practical analysis to low-order masking.

Hardware-software co-verification and contracts: COCO [17] performs co-verification of masked software on CPU gate-level netlists, identifying microarchitectural structures that violate masking assumptions and proposing hardware modifications to restore formal security. Power

Table 5: Comparison of Leakage Detection Approaches

Tool	Approach	Design source	Automatic	Scalable	Precise pinpointing of leakage	
					Software	Microarch
ELMO [24]	Simulation	Measured Hamming Weight model	✓	✗	✓	✗
ROSITA [30]	Simulation	Measured Hamming Weight model	✓	✗	✓	✗
ROSITA++ [29]	Simulation	Measured Hamming Weight model	✓	✗	✓	✗
PROLEAD_SW [34]	Formal	Generic CPU model	✓	✗	✓	✗
PARAM [16]	RTL simulation	RTL	✓	✓	✗	✓
TELESCOPE [22]	Gate level simulation	Gate-level RTL	✓	✓	✗*	✓
COCO [17]	Formal	Gate-level RTL	✗	✓	✗	✓
Power Contracts [6]	Formal	RTL	✗	✓	✓	✗
microSCALE	Static analysis	RTL	✓	✓	✓	✓

* The identified instruction may not be the true source, as leakage introduced by an earlier instruction can appear in a later cycle.

Contracts [6] define formal leakage contracts for processors, specifying what information may be exposed through side channels and enabling compliance checking of software against these contracts. These approaches provide strong formal guarantees but are tightly coupled to specific processor implementations.

All the prior works we discussed detect the presence of leakage; however, they either attribute leakage to instruction-level behavior or pinpoint the hardware components responsible for share recombination, but do not correlate both perspectives, thereby leaving a critical gap for further research. Table 5 gives a comparison between the existing works.

8 Conclusion and Future Work

Existing techniques for the detection of power side channels typically consist of a trade-off between strong guarantees and efficiency. They do not pinpoint the root cause of the leakage across both the hardware and software. To address this gap, we develop `microSCALE`, an automated framework for detecting power-side channel attacks in masked cryptographic implementations on target processors, that pinpoints the root cause of the leakage in both the hardware and software. The framework further associates the observed leakage with the corresponding secret input(s). `microSCALE` provides completeness guarantees while also executing with a short runtime.

For a given processor, the first two stages of `microSCALE` execute once; their outputs are reused to evaluate any new cipher implementation on that processor and therefore does not need to be repeated. This assists in building strong cryptographic implementations catered towards a specified processor. In the future, our aim is to develop compilers for target processors to help designers mitigate the leakage sources identified by the framework. We also plan to improve `microSCALE`'s support for handling branch instructions.

Acknowledgments

This work is funded by the Ministry of Electronics and Information Technology, India (MeitY) and the Industrial Development Bank of India (IDBI). The authors also acknowledge the use of generative AI tools for minor editorial assistance, including grammar, clarity, and stylistic refinement of the manuscript.

Ethical Considerations

Decision to Conduct and Publish. Masked cryptographic software can still leak on real processors due to microarchitectural effects, and existing tools do not give practitioners a systematic way to find and fix such leakages. We built `microSCALE` to address this. The tool is defensive in nature and does not introduce any new attack capability. We believe the security benefits of publishing this work outweigh the risks.

Stakeholders. The main stakeholders are cryptographic software developers, hardware designers working with in-order processors, and end users who rely on masked cryptographic software for security.

Impacts. Even when the processor is correct and the masking scheme is algorithmically secure, shares can recombine inside the microarchitecture during execution, breaking the masking guarantee. `microSCALE` helps developers find exactly where this happens. It is a detection tool, not an attack tool, and we do not publish any exploits which make attacks easier.

Mitigation. We notified the RocketChip maintainers via their security channel (`security@lists.chipsalliance.org`) before submission. Post-silicon, software-level fixes such as instruction reordering and dummy instructions can reduce leakage significantly, as confirmed by TVLA. Pre-silicon, the stage-level attribution provided by `microSCALE` allows hardware designers to identify and address structurally vulnerable designs before tapeout.

Experimental Setup. All experiments were done on dedicated hardware in a controlled setting. No commercial hardware, live systems, or third-party platforms were involved, and no end user data was used at any point.

Open Science

The artifact repository (`https://zenodo.org/records/20409592`) provides the main source code, selected test programs, a Dockerfile, and a detailed README describing how to build and use the framework.:

References

[1] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moretó, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip Generator. *EECS Department, University of California, Berkeley*, 2016. Technical Report No. UCB/EECS-2016-17. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[2] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the Cost of Lazy Engineering for Masked Software Implementations. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, Lecture Notes in Computer Science, pages 64–81. Springer, 2014. doi:10.1007/978-3-319-16763-3_5.

[3] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A Small Present - Towards Reaching the Limit of Lightweight Encryption. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 321–345. Springer, 2017. doi:10.1007/978-3-319-66787-4_16.

[4] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019. doi:10.1007/978-3-030-29959-0_15.

[5] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong Non-Interference and Type-Directed Higher-Order Masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129. ACM, 2016. doi:10.1145/2976749.2978427.

[6] Roderick Bloem, Barbara Gigerl, Marc Gourjon, Vedad Hadzic, Stefan Mangard, and Robert Primas. Power Contracts: Provably Complete Power Leakage Models for Processors. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 381–395. ACM, 2022. doi:10.1145/3548606.3560600.

[7] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007. doi:10.1007/978-3-540-74735-2_31.

[8] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004. doi:10.1007/978-3-540-28632-5_2.

[9] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In *CRYPTO 1999, Proc.*, pages 398–412, 1999. URL: http://dx.doi.org/10.1007/3-540-48405-1_26, doi:10.1007/3-540-48405-1_26.

[10] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. Conversion of Security Proofs from One Leakage Model to Another: A New Issue. In Werner Schindler and Sorin A. Huss, editors, *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*, Lecture Notes in Computer Science, pages 69–81. Springer, 2012. doi:10.1007/978-3-642-29912-4_6.

[11] Jean-Sébastien Coron, Aurélien Greuet, and Rina Zeitoun. Side-channel Masking with Pseudo-Random Generator. *IACR Cryptol. ePrint Arch.*, page 1106, 2019. URL: https://eprint.iacr.org/2019/1106.

[12] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-Order Side Channel Security and Mask Refreshing. *IACR Cryptol. ePrint Arch.*, page 359, 2015. URL: http://eprint.iacr.org/2015/359.

[13] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*, pages 106–117. IEEE Computer Society, 2012. doi:10.1109/ISCA.2012.6237010.

[14] Digilent. Nexys Video FPGA Board. https://www.digikey.in/en/product-highlight/d/digilent/nexys-video-fpga-board.

[15] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2: Lightweight Authenticated Encryption and Hashing. *J. Cryptol.*, 34(3):33, 2021. URL: https://doi.org/10.1007/s00145-021-09398-9, doi:10.1007/S00145-021-09398-9.

[16] Muhammad Arsath K. F, Vinod Ganesan, Rahul Bodduna, and Chester Rebeiro. PARAM: A Microprocessor Hardened for Power Side-Channel Attack Resistance. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020*, pages 23–34. IEEE, 2020. doi:10.1109/HOST45689.2020.9300263.

[17] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1469–1468. USENIX Association, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/gigerl.

[18] 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.

[19] Hannes Gross, Stefan Mangard, and Thomas Korak. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. *IACR Cryptol. ePrint Arch.*, page 486, 2016. URL: http://eprint.iacr.org/2016/486.

[20] Yuval Ishai, Amit Sahai, and David A. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003. doi:10.1007/978-3-540-45146-4_27.

[21] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *CRYPTO 1999*, pages 388–397, 1999.

[22] Zhenyuan Liu, Andrew Malnicof, Arna Roy, and Patrick Schaumont. Telescope: Top-Down Hierarchical Pre-silicon Side-channel Leakage Assessment in System-on-Chip Design. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2025, Hanoi, Vietnam, August 25-29, 2025*, pages 1280–1293. ACM, 2025. doi:10.1145/3708821.3736216.

[23] Ben Marshall, Dan Page, and James Webb. MIRACLE: MIcRo-ArChitectural Leakage Evaluation A study of micro-architectural power leakage across many devices. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):175–220, 2022. URL: https://doi.org/10.46586/tches.v2022.i1.175-220, doi:10.46586/TCHES.V2022.I1.175-220.

[24] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 199–216. USENIX Association, 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/mccann.

[25] Nicolai Müller, David Knichel, Pascal Sasdrich, and Amir Moradi. Transitional Leakage in Theory and Practice Unveiling Security Flaws in Masked Circuits. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(2):266–288, 2022. URL: https://doi.org/10.46586/tches.v2022.i2.266-288, doi:10.46586/TCHES.V2022.I2.266-288.

[26] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the Gap: Towards Secure 1st-Order Masking in Software. In Sylvain Guilley, editor, *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, volume 10348 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2017. doi:10.1007/978-3-319-64647-3_17.

[27] RISC-V International. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. RISC-V International,

20191213 edition, 2019. https://riscv.org/technical/specifications/.

[28] Matthieu Rivain and Emmanuel Prouff. Provably Secure Higher-Order Masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010. doi:10.1007/978-3-642-15031-9_28.

[29] Madura A. Shelton, Lukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. Rosita++: Automatic Higher-Order Leakage Elimination from Cryptographic Code. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 685–699. ACM, 2021. doi:10.1145/3460120.3485380.

[30] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. URL: https://www.ndss-symposium.org/ndss-paper/rosita-towards-automatic-elimination-of-power-analysis-leakage-in-ciphers/.

[31] Ischa Stork, Vipul Arora, Lukasz Chmielewski, and Ileana Buhan. Towards Explainable Side-Channel Leakage: Unveiling the Secrets of Microarchitecture. *IACR Cryptol. ePrint Arch.*, page 1792, 2024. URL: https://eprint.iacr.org/2024/1792.

[32] Nikita Veshchikov and Sylvain Guilley. Use of Simulators for Side-Channel Analysis. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 51–59. IEEE, 2017. URL: https://doi.org/10.1109/EuroSP.2017.31, doi:10.1109/EUROSP.2017.31.

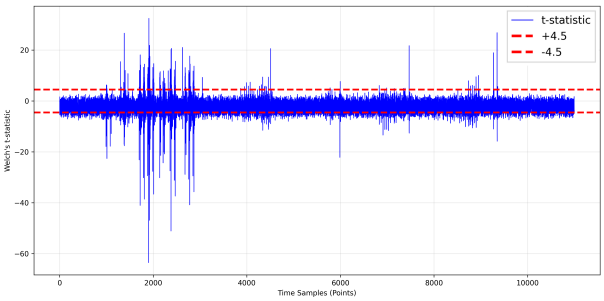
[33] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 2128–2142. ACM, 2023. doi:10.1145/3576915.3623192.

[34] Jannik Zeitschner, Nicolai Müller, and Amir Moradi. PROLEAD_SW Probing-Based Software Leakage Detection for ARM Binaries. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):391–421, 2023. URL: https://doi.org/10.46586/tches.v2023.i3.391-421, doi:10.46586/TCHES.V2023.I3.391-421.

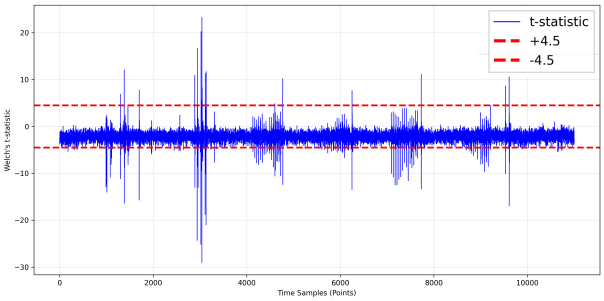
A Appendix

Table 6: Leaking components in each processor stage.

Register Types	Register Names
MEM	mem_reg_rs2, mem_reg_wdata
WB	wb_reg_xcpt, wb_reg_wdata, wb_reg_flush_pipe
EX	ex_reg_rs_lsb_1, ex_reg_rs_lsb_0
Other	reg_smtval, reg_stval



(a) TVLA t-statistics for GIFT unhardened.



(b) TVLA t-statistics for GIFT hardened.

Figure 8: TVLA t-statistics: unhardened vs. hardened GIFT.

A.1 Verilog Codes corresponding to Figure 1

```
reg  D1, D2, D3, D4, D5, D6, D7;
wire W0, W1, W2, W3, W4, W5, W6;

assign W0 = (S3 == 2'd0) ? D1 : 1'b0;
```

```
assign W1 = (S3 == 2'd1) ? D1 : 1'b0;
assign W2 = (S3 == 2'd2) ? D1 : 1'b0;
assign W3 = (S3 == 2'd3) ? D1 : 1'b0;
assign W4 = (S4 == 2'd0) ? D2 : 1'b0;
assign W5 = (S4 == 2'd1) ? D2 : 1'b0;
assign W6 = (S4 == 2'd2) ? D2 : 1'b0;

always @(posedge clk) begin
    D1 <= (S1 == 1'b1) ? in1 : in0;
    D2 <= (S2 == 1'b1) ? in3 : in2;
    D3 <= W1 & W4;
    D4 <= W2 | W5;
    D5 <= W3 ^ W6;
    D6 <= (S5 == 1'b1) ? D4 : D3;
    D7 <= (S6 == 2'd0) ? W0 :
          (S6 == 2'd1) ? D6 :
          (S6 == 2'd2) ? D5 : 1'b0;
end
```

Listing 2: RTL code corresponding to Circuit in Figure 1.

```
input inst;
reg  opc, oper;
reg  opc_1, opc_2, opc_3;
reg  oper_1, oper_2, oper_3;
wire       s1, s2, s5, s7;
wire [1:0] s3, s4, s6;
wire [4:0] rs1 = oper[4:0];
wire [4:0] rs2 = oper[9:5];
wire [4:0] rd  = oper[14:10];
assign s1 = (rs1 == 0) ? 1'b0 : 1'b1;
assign s2 = (rs2 == 0) ? 1'b0 : 1'b1;
assign s3 = (opc_1 == `MV ) ? 2'd0 :
            (opc_1 == `AND) ? 2'd1 :
            (opc_1 == `OR ) ? 2'd2 :
            (opc_1 == `XOR) ? 2'd3 : 2'd0;
assign s4 = (opc_1 == `AND) ? 2'd0 :
            (opc_1 == `OR ) ? 2'd1 :
            (opc_1 == `XOR) ? 2'd2 : 2'd0;
assign s5 = (opc_2 == `OR) ? 1'b1 : 1'b0;
assign s6 = (opc_3 == `MV ) ? 2'd0 : ((opc_3 == `AND)
             || (opc_3 == `OR)) ? 2'd1 :(opc_3 == `XOR)
             ? 2'd2 : 2'd0;

assign s7 = (data == 0) ? 1'b0 : 1'b1;

always @(posedge clk) begin
    opc_1  <= opc;
    oper_1 <= oper;
    opc_2  <= opc_1;
    oper_2 <= oper_1;
    opc_3  <= opc_2;
    oper_3 <= oper_2;
end
```

Listing 3: RTL code corresponding to Decoder in Figure 1.

A.2 Leakage Analysis plots for ciphers in Table 4

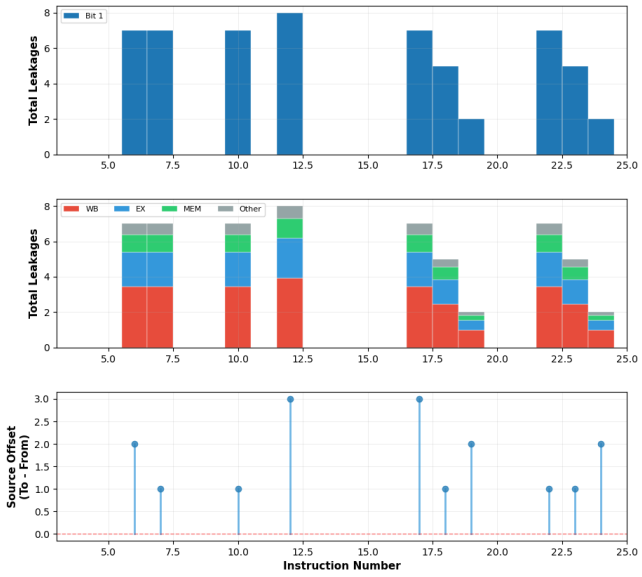


Figure 9: Leakage analysis plot for AES s-box.

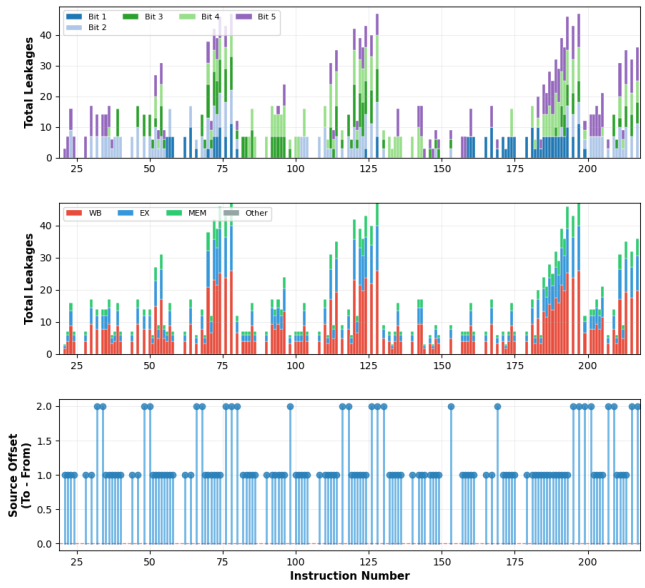


Figure 10: Leakage analysis plot for ASCON s-box.



Figure 11: Leakage analysis plot for PRESENT s-box.

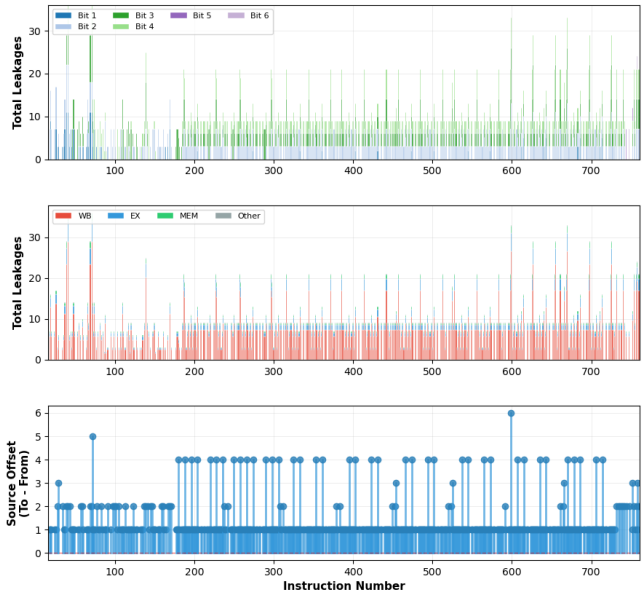


Figure 12: Leakage analysis plot for GIFT one round.