# Quantifying Software Leakage via Transmitters with Leakage Functions

## I. Introduction

In a hardware side-channel attack, a transmitter modulates a channel (a hardware resource) in a secret-dependent manner. Most often, and in this paper, a transmitter is an instruction in the victim program that causes variability in hardware resource usage (i.e. channel modulation) as a function of its operands [15]. A receiver (attacker) observes the channel modulation via its impact on certain non-deterministic aspects of program execution, like timing, resource contention, power dissipation, and more, and thereby infers the operand's value [17]. Hence, when a secret program input is passed to an unsafe transmitter operand, an attacker can gain information about the secret.

Our work uses *microarchitectural leakage functions* to quantify leakage through a program's transmitters and thereby assess the ability of a program to protect the secrets it processes.[1] Microarchitectural leakage functions characterize channel modulations by mapping transmitter operands to *microarchitectural execution paths* ($\mu$paths), where a $\mu$path is defined as a partial order on the state updates made by an instruction during its execution. In this paper, we consider transmitters who create operand-dependent variability in their own $\mu$paths. In general, however, a transmitter can create receiver-observable $\mu$path variability for itself and/or other instructions. From a leakage function, we can compute the probability distributions of observable $\mu$paths conditioned on transmitter operands. Using these probability distributions, we can quantify leakage with metrics such as *mutual information* [19] and *maximal leakage* [14].

In this paper, we profile three leakage functions; each corresponds to one of three hardware optimizations. Further, we look at two of the leakage functions in the context of the Poly1305 cryptographic hashing algorithm [2]. In doing so, we consider how individual transmitters contribute to overall leakage of a programs' secret inputs.

## II. Background & Motivation

State of the art hardware side-channel defenses adopt an "all or nothing" strategy, where instruction operands are either entirely safe or unsafe. Hence, unsafe operands can never process secrets. Such is the case in the well-known, constant-time programming discipline [7]. Due to the increasing complexity of post-Moore's Law optimizations, we will either have no safe instructions [17] or overly conservative mitigations that disable optimizations altogether, such that computational efficiency is soon prohibitively degraded [7]. Therefore, programmers need to be able to reason about the risk of a program leaking its secrets in the presence of leaky instructions.

[1] Our concurrent work proposes this abstraction for precisely characterizing the leakage of transmitters from SystemVerilog processor designs.

```
path MUL(instr i0):
  return ite ((i0.arg0 == 0) ∨ (i0.arg1 == 0),
    fast_path, slow_path)
```

Figure 1: Zero-skip multiplier leakage function

Like prior work [10], we define random variables in our communication channel model of a side-channel. The secret space, $S$, includes all possible secret values. The victim modulation space, $X_V$, captures the channel modulations that result from transmitters in the victim program. In our model, channel modulations correspond to the variability in instructions' $\mu$paths. Lastly, the observation space, $Y$, represents the attacker's observations of the channel. The observation space could be same as the victim modulation space, or it could be a projection partitioning of the victim modulation space based on an observer model, such as timing or contention. We assume a powerful attacker that can observe the $\mu$paths of in flight instructions, i.e. the former scenario. Note that prior work also models victim mitigation strategies and attacker strategies in side-channel models. While we do not include these here, they could easily be incorporated in future work.

## III. Methodology

Our work is based on the observation that transmitters leak more information about their operands in certain circumstances (e.g. for particular operand values), while in other circumstances, minimal information may be leaked. For example, consider a multiplier that implements the canonical zero-skip optimization, where a multiply (`MUL`) instruction takes a "fast" path if one or more of its operands are zero; else, it takes a "slow" $\mu$path. Clearly, a `MUL` creates $\mu$path variability for itself as a function of its operand values. If an attacker observes a "fast" path, they can infer that at least one operand is zero, which potentially reveals the entire secret. If a "slow" path is observed, an attacker only learns that the operands are non-zero, exposing relatively little information about the secret value. Therefore, we use leakage functions to model how $\mu$paths vary with respect to transmitter operands.

Our methodology is as follows: first, we acquire the leakage functions corresponding to all the transmitters in a program. Second, we deploy the leakage functions to compute the probabilities of observable $\mu$paths conditioned on transmitter operands. Using the conditional probabilities, we quantify side-channel leakage through each transmitter. Lastly, we use symbolic execution to determine if secret program inputs are passed to transmitter operands, and if so, determine if multiple transmitters can create additional leakage.

### A. Step 1: Microarchitectural Leakage Functions

We consider three leakage functions based on hardware optimizations proposed in the literature: 32-bit zero-skip mul-

| Optimization | Description | No. of Paths |
|---|---|---|
| Zero-skip mult | `MUL` insts take fast path if one or more operands are 0, else slow path | 2 |
| Digit-serial mult | `MUL` insts take 1 of 4 execution paths based on if bytes of 2nd operand are 0 | 4 |
| CVA6 div | `DIV` insts take 1 of 66 paths based on both operands' no. of leading 0s | 66 |

Table I: Leakage functions for 3 HW optimizations

tiplier [17], 32-bit digit-serial multiplier [11], and 64-bit serial division as implemented in CVA6 [21]. Each of these leakage functions outputs $\mu$paths as a function of one or more of its operands. See Table I for details.

### B. Step 2: Leakage Quantification

To quantify the leakage of a program, we compare two metrics, maximal leakage and mutual information. Maximal leakage measures the maximum probability of each observation given any secret value. Mutual information is the amount of information that an attacker learns about the secret space from the observation space. Maximal leakage and mutual information can be computed, respectively:

$$L_{ML} = \log_2 \sum_{y \in Y} \max_{s \in S} P(y|s)$$

$$L_{MI} = \sum_{s \in S, y \in Y} P(s)P(y|s) \log_2 \frac{P(y|s)}{\sum_{s \in S} P(s)P(y|s)}$$

It has been shown that mutual information is upper bounded by maximal leakage [20]. Thus, prior work uses maximal leakage as the appropriate metric to ensure that leakage quantification is conservative when exact leakage functions are unknown [10]. However, maximal leakage may be overly pessimistic, because it overlooks the probability of the secret value itself. Microarchitectural leakage functions allow us to more precisely define the probability of an attacker observation conditioned on the secret operands.

### C. Step 3: Program Analysis

We encountered two main challenges when trying to quantify leakage from transmitters in cryptographic programs. First, the values of secret program inputs may be transformed before they are passed to transmitter operands. Second, there can be many transmitters in a single program which may layer on top of one another and increase overall leakage. To overcome these challenges, we used the symbolic execution tool, KLEE [6], to determine how secrets are transformed from program inputs to transmitter operands and whether multiple transmitters leak additional information about the secret. To do so, we instrumented the program and created a software model of the leakage functions as a wrapper around the program's transmitters. Then by running the program on symbolic inputs, we characterize the observable $\mu$paths resulting from secret inputs passed to transmitter operands.

We consider the Poly1305 hashing algorithm for secure message authentication [18]. This program includes many multiplication instructions where one of the two operands is secret. Additionally, Poly1305 includes multiplications where one operand is a transformed value of a secret input. Therefore, this program is a good candidate to evaluate zero-skip multiplication and digit-serial multiplication leakage functions.

## IV. Preliminary Results

For each leakage function in Table I, we compute the mutual information and maximal leakage, assuming that the secret values are uniformly distributed. This assumption is valid for our experiments because cryptographic keys are randomly distributed. For zero-skip multiplication, the mutual information was nine orders of magnitude smaller than maximal leakage, because the probability that the secret value is zero is very small, $\frac{1}{2^{32}}$. Since mutual information takes the probability of the secret into account and maximal leakage does not, this explains the significant difference in the two metrics. For the digit-serial multiplier, the mutual information is less than two orders of magnitude smaller than maximal leakage. For CVA6 division, the mutual information is only about one third of the maximal leakage. While we expect maximal leakage to be conservative, these results show that when certain observations only result from secret values with very low probabilities, maximal leakage might be overly pessimistic.

Using KLEE to analyze the multiplication instructions in Poly1305, we found that multiplying by a transformed secret value had different leakage depending on the leakage function. For zero-skip multiplication, the transformed secret does not change the attacker observation, and therefore does not leak additional information about the secret. For digit-serial multiplication, however, the transformed secret can result in a different attacker observation, resulting in additional leakage.

**Conclusion:** Precise leakage functions allow us to be more realistic about the risk of side-channel leakage. Thus, programs can be designed to protect secrets despite using unsafe instructions.

## V. Related Work

**Leakage quantification:** Several works have evaluated different quantitative metrics for side-channel leakage [1, 3, 9, 14, 19, 20, 22]. A few works look at metrics for specific types of side-channels [13] or specific mitigation strategies [10]. Our work evaluates how more precise metrics can leveraged by using microarchitectural leakage functions.

**Leakage contracts:** Hardware-software leakage contracts have been proposed in prior work [12, 16, 17]. These contracts often broadly classify operands of transmitters as safe or unsafe. Microarchitectural leakage functions, on the other hand, return exact $\mu$paths, and therefore allow us to consider the conditions under which said operands are safe or unsafe.

**Symbolic execution for leakage analysis:** Prior work uses symbolic execution in the realm of side-channel discovery and mitigation [4, 5, 8]. However, these approaches often target a single type of side-channel, such as cache-based side-channel attacks, or they are purely focused on using symbolic execution to analyze software. We use symbolic execution to model attacker observations (i.e. $\mu$paths) that result from secrets passed to transmitter operands.

## References


[1] Màrio S. Alvim et al. "Measuring Information Leakage Using Generalized Gain Functions". In: *2012 IEEE 25th Computer Security Foundations Symposium*. 2012, pp. 265–279. DOI: 10.1109/CSF.2012.26.

[2] Daniel J. Bernstein. "The Poly1305-AES Message-Authentication Code". In: *Fast Software Encryption*. Ed. by Henri Gilbert and Helena Handschuh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 32–49. ISBN: 978-3-540-31669-5.

[3] Christelle Braun, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. "Quantitative Notions of Leakage for One-try Attacks". In: *Electronic Notes in Theoretical Computer Science* 249 (2009). Proceedings of the 25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009), pp. 75–91. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2009.07.085. URL: https://www.sciencedirect.com/science/article/pii/S1571066109003077.

[4] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. "Symbolic path cost analysis for side-channel detection". In: May 2018, pp. 424–425. DOI: 10.1145/3183440.3195039.

[5] Robert Brotzman et al. "CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 505–521. DOI: 10.1109/SP.2019.00022.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[7] Sunjay Cauligi et al. "SoK: Practical Foundations for Spectre Defenses". In: *CoRR* abs/2105.05801 (2021). arXiv: 2105.05801. URL: https://arxiv.org/abs/2105.05801.

[8] Sudipta Chattopadhyay et al. "Quantifying the Information Leak in Cache Attacks through Symbolic Execution". In: *CoRR* abs/1611.04426 (2016). arXiv: 1611.04426. URL: http://arxiv.org/abs/1611.04426.

[9] John Demme et al. "Side-Channel Vulnerability Factor: A Metric for Measuring Information Leakage". In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA '12. Portland, Oregon: IEEE Computer Society, 2012, pp. 106–117. ISBN: 9781450316422.

[10] Peter W. Deutsch et al. "Metior: A Comprehensive Model to Evaluate Obfuscating Side-Channel Defense Schemes". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA '23. Orlando, FL, USA: Association for Computing Machinery, 2023. ISBN: 9798400700958. DOI: 10.1145/3579371.3589073. URL: https://doi.org/10.1145/3579371.3589073.

[11] Johann Großschädl et al. *Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications*. Cryptology ePrint Archive, Paper 2009/538. https://eprint.iacr.org/2009/538. 2009. URL: https://eprint.iacr.org/2009/538.

[12] Marco Guarnieri et al. *Hardware-Software Contracts for Secure Speculation*. 2020. arXiv: 2006.03841 [cs.CR].

[13] Casen Hunger et al. "Understanding contention-based channels and using them for defense". In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture, HPCA 2015* (Mar. 2015), pp. 639–650. DOI: 10.1109/HPCA.2015.7056069.

[14] Ibrahim Issa, Aaron B. Wagner, and Sudeep Kamath. "An Operational Approach to Information Leakage". In: *CoRR* abs/1807.07878 (2018). arXiv: 1807.07878. URL: http://arxiv.org/abs/1807.07878.

[15] Vladimir Kiriansky et al. "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018, pp. 974–987. DOI: 10.1109/MICRO.2018.00083.

[16] Nicholas Mosier et al. "Axiomatic Hardware-Software Contracts for Security". In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22. New York, New York: Association for Computing Machinery, 2022, pp. 72–86. ISBN: 9781450386104. DOI: 10.1145/3470496.3527412. URL: https://doi.org/10.1145/3470496.3527412.

[17] Jose Rodrigo Sanchez Vicarte et al. "Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 347–360. DOI: 10.1109/ISCA52012.2021.00035.

[18] The OpenSSL Project. "OpenSSL: The Open Source toolkit for SSL/TLS". www.openssl.org. Apr. 2003.

[19] Isabel Wagner and David Eckhoff. "Technical Privacy Metrics: a Systematic Survey". In: *CoRR* abs/1512.00327 (2015). arXiv: 1512.00327. URL: http://arxiv.org/abs/1512.00327.

[20] Benjamin Wu, Aaron B. Wagner, and G. Edward Suh. "A Case for Maximal Leakage as a Side Channel Leakage Metric". In: *CoRR* abs/2004.08035 (2020). arXiv: 2004.08035. URL: https://arxiv.org/abs/2004.08035.

[21] Florian Zaruba and Luca Benini. *CVA6 RISC-V CPU*. https://github.com/openhwgroup/cva6. 2019.

[22] Tianwei Zhang et al. "Side Channel Vulnerability Metrics: The Promise and the Pitfalls". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '13. Tel-Aviv, Israel: Association for Computing Machinery, 2013. ISBN: 9781450321181. DOI: 10.1145/2487726.

2487728. URL: https://doi.org/10.1145/2487726.2487728.