Quantifying Leakage Using Side Channel Leakage Functions

Samantha Archer
Electrical Engineering Department
Stanford University
Stanford, CA
samanthaarcher@stanford.edu

Abstract—**Hardware side channels have been a long standing problem for the security research community. As data-dependent hardware optimizations become more and more common in the post-Moore's Law era of computing, novel hardware implementations will likely be vulnerable to side channel attacks. Unfortunately, there have been two shortcomings in current side channel research. First, most security-centric descriptions of hardware implementations are binary: either a hardware design, instruction, or program is safe, or it is unsafe. Second, there is no established way to model and communicate microarchitectural side channel vulnerabilities to programmers, such that they can design secure software with hardware in mind.**

To overcome these limitations of existing side channel analyses, we propose using existing *microarchitectural leakage functions* to model side channels in specific hardware implementations. Leakage functions, generated from hardware execution paths, can be used to quantify the theoretical leakage of a side channel and to determine the leakage of instructions in the context of a specific program. In this project, we evaluate three different leakage functions which arise from proposed data-dependent hardware optimizations. From the leakage functions, we reason about the conditional probability of observable instruction execution paths and use the probabilities to compute the leakage. Additionally, we consider two of these leakage functions in the context of the Poly1305 cryptographic hashing algorithm. Using this program as a test case, we hope to reason about the combined leakage of a program with an arbitrary number of leaky instructions.

I. Introduction

Hardware side channels are a class of security vulnerabilities that arise when specific instructions, called *transmitter instructions*, cause data-dependent hardware resource usage. For example, transmitter instructions might cause a program to vary in timing, power, memory access pattern, or contention in a data-dependent manner. When secret data is passed to transmitter instructions, attackers have the opportunity to gain information about the secret by monitoring the data-dependent hardware resources.

Both hardware and software mitigations have been proposed to prevent leakage through side channels. However, in many of the proposed mitigations, there have been two fatal flaws: either the mitigations are too heavy-handed such that the performance degradation is prohibitive, or they are insufficient such that they do not provide complete protection against novel side channel attacks. Many hardware mitigations fall into the first category, as they disable optimizations that induce side channels. These coarse-grained mitigations often result in such significant performance overhead that users are disinclined to enable them. On the other hand, many software mitigations are often incomplete, because side channel vulnerabilities cannot be prevented in software without a complete and accurate model of the hardware implementation.

Therefore there has been a push for hardware-software co-design in side channel mitigation. In order for programmers to be able to design secure software to run on a particular hardware, they need to be able to accurately reason about the vulnerabilities of the hardware. Prior work has shown that it is feasible to extract the possible microarchitectural execution traces of an instruction. Using these instruction execution traces, we can specify *microarchitectural leakage functions* that define the side channel behavior of the hardware.

This project uses microarchitectural leakage functions to analyze programs and quantify the leakage through transmitter instructions. Based on the leakage functions, we compute the probability distributions of observable execution paths conditioned on secret input data. Using the probability distributions, we can quantitatively reason about the leakage of transmitter instructions, using metrics such as mutual information and maximal leakage. To evaluate this methodology, we consider a variety of leakage functions for proposed hardware optimizations and compute the theoretical leakage of each function.

In an end-to-end example, we attempt to reason about the leakage of transmitter instructions in the Poly1305 cryptographic hashing algorithm. Looking at a program in its entirety provides needed context to compute the leakage of individual transmitters. However, it also introduces challenges, including reasoning about the transformation of secrets before they are passed to transmitter instructions and reasoning about the combined leakage of multiple transmitters in a single program.

Lastly, I want to mention that this project is ongoing. Some experiments are still in progress, and some results may not be conclusive yet.

II. Motivation & Background

Hardware side channels pose a serious threat to computer systems, and therefore, they have been the focus of much security research. To differentiate our work from existing work, our motivation is to model hardware side channels precisely, and based on these models of side channels, use program analysis techniques to quantify the leakage of a program. This will allow hardware and software designers to come up with more robust and efficient mitigation strategies to prevent side channel leakage without taking a steep performance hit.

A. Modeling side channels

There is existing work looking at modeling the hardware side channel leakage using microarchitectural leakage functions. Using model checking tools, prior work identifies the execution paths of an instruction in a microarchitecture. From the execution paths, we can define the leakage function with respect to the operands of that instructions and other related instructions. Therefore software developers can specify how secret inputs are passed to transmitters based on the leakage functions, enabling the design of verifiably secure software with minimal overhead. However, since the leakage functions were only recently introduced, there are still open questions in regards to how these leakage functions are defined for novel hardware designs and how the functions can be used by programmers.

Historically leakage has been quantified in a binary manner: either an instruction, program, or microarchitectural design is safe or unsafe. There has been prior work showing that it is possible to quantify the amount of leakage using metrics such as mutual information and maximal leakage. Often however, research has defaulted to the most pessimistic metric, maximal leakage, because it was not possible to reason more precisely about the leakage with respect to the space of inputs. Using leakage functions, we can accurately quantify leakage based on the probability of side channel observations.

Lastly, in prior work, side channels have been classified with respect to their characteristics. Specifically, side channels can be either *active* or *passive*, depending on whether the attacker actively interferes to exaggerate the effects of the side channel. Another categorization of side channels is *persistent* or *ephemeral*. A persistent side channel allows an attacker to make an observation after the transmitter instruction has committed, where as ephemeral side channels are only observable for a short amount of time. Lastly, we are adding a new categorization of side channels: *independent* or *dependent*. An independent side channel is one where the transmitter instruction does not depend on any other instruction either in flight or previously executed. A dependent side channel requires both a transmitter instruction and at least one other instruction in order for the side channel to be observable.

B. Mitigating side channels

To prevent leakage through hardware side channels, effective software-based mitigations have been introduced, including programming principles such as constant time programming. However the software mitigations have been proven to be insecure for speculative side channels, where transmitters are executed speculatively, such as Spectre and Meltdown.

On the other hand, proposed hardware mitigations generally have high overhead. For example, to mitigate Spectre and Meltdown, hardware companies have introduced modes to disable speculative execution altogether, which results in severe performance degradation. As Moore's Law has come to an end, computer architects will almost certainly find creative ways to use data-dependent hardware optimizations to continue improving performance. Such optimizations have already been proposed, and while they have potential to improve hardware performance, they will likely lead to new security vulnerabilities through hardware side channels [1].

Because it is impossible to ensure the execution of every instruction is free of side channels, it is imperative that hardware security vulnerabilities are well-defined. This will allow software designers to take security vulnerabilities into account when designing provably secure software and compilers.

III. Methodology

A. Communication Channel Model of Security

It is common today to model a hardware side channel as a communication channel, where a secret input is communicated with some modulation [2]. For hardware side channels, the modulated communication channel is the hardware resource. To complete the communication definition of side channels, we define random variables to represent the spaces of secrets, victim modulations, attacker modulations, and observations. Specifically, the secret space, S includes all the possible values of a secret $s \in S$. The victim modulation space, X_V, is defined as the channel modulation resulting from the secret value. If the victim modulation varies in a value-dependent way, the channel is considered a hardware side channel. This random variable may also include any mitigations to prevent side channel leakage. For this project, however, we will not consider any mitigations. We hope mitigation schemes can be included in future work. The attacker modulation space X_A is the modulation that an attacker might use to induce the maximal amount of leakage. An example of attacker modulation might be priming a cache in a cache-based side channel attack. Again, we will not consider attacker modulation strategies until later stages of this project (beyond the scope of this class). Lastly, the observation space, Y, is the space of potential observations that the attacker sees, such as the varying hardware resource usage. We will assume a worst case attacker that can exactly view the execution path of the hardware. By this assumption and because we are not considering any attacker modulation, the observation space Y is exactly equal to the victim modulation space X_V.

B. Microarchitectural Leakage Functions

In order to accurately model the leakage of each transmitter instruction, we will use the previously proposed microarchitectural leakage functions. However, for novel hardware optimizations, the microarchitecture leakage functions are not yet formally defined. Therefore to consider these hardware optimizations, we will need to come up with the appropriate leakage functions by analyzing how arguments to transmit instructions or other instructions affect microarchitecture execution paths. Using these leakage functions, we can determine the data-dependent timing behavior induced in the side channel, and as a result, the amount of leakage.

Additionally, we want to model the various side categorizations in the leakage functions. For example, the leakage functions should take into account whether the side channel is passive or active, ephemeral or persistent, and independent or dependent. In this project, we only had time to consider leakage functions of a subset of these side channel categorizations.

C. Leakage Quantification

To quantify the leakage of a program, we plan to use information theory metrics, such as maximal leakage and mutual information. Given that s is a secret in secret space S and y is a attacker observation in observation space Y, maximal leakage can be computed as:

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

and mutual information can be computed as:

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

It has been shown in [3] that mutual information is upper bounded by maximal leakage. Therefore in prior work [2], maximal leakage has been used as the appropriate metric to ensure that leakage quantification is strictly pessimistic. However, microarchitectural leakage functions allow us to more precisely define how secrets modulate the hardware resources. Concretely, we can use the leakage functions to determine the probability of an attacker observation conditioned on the secret operands. Mutual information takes into account the probability of a specific secret value that can lead to an observation. For example if one observation can leak a lot of information about the secret, but that observation is very unlikely based on a uniformly distributed secret value, mutual information will take in account the low probability of this secret value in the information leakage. Therefore we are able to reason when it is be appropriate to use mutual information as a more accurate and less pessimistic estimate of leakage.

While we continue to consider the benefits of the different leakage metrics, we also need to consider the fact that a single program can have many transmitter instructions. Therefore we will need to reason about how leakage quantities are affected when there are multiple transmit instructions in the program.

D. Program Analysis

We analyze cryptographic programs to gain insights into how secrets are passed to transmitters and which transmitters are reachable in program execution paths. To do so, we use the symbolic execution tool, KLEE. Additionally, we used KLEE to determine whether a program's transformations of a secret can leak additional information about the secret.

For future work, we are also considering using approximate model counting to partition the input space with respect to the reachable transmit instructions for those inputs. However, for large input spaces, such as cryptographic keys, approximate model counting may be intractable. Other directions of program analysis include using taint tracking to determine which inputs to transmit instructions are secret. We can annotate data with a "taint", such as public, secret, or attacker-controlled. Therefore we can track secret data to determine when they reach the inputs of transmit instructions. Lastly, we could use probabilistic program analysis tools in order to verify the probability distribution that a secret data value arrives at a specific transmitter's operand. However, these program analysis techniques were out of scope for this class.

IV. Evaluation Methodology

A. Leakage functions

To evaluate our methodology, we consider 3 leakage functions based on hardware optimizations proposed in the literature: zero-skip multiplier [1], digit serial multiplier [4], and CVA6 division [5]. For the multipliers we consider 32 bit operands, and for the division we assume 64 bit operands. Each of these leakage functions assumes a timing side channel, where the transmitter instruction (multiplication or division) takes a different number of clock cycles depending on one or more of the instruction operands.

The zero skip multiplier optimizes a multiplication instruction by taking a fast execution path (shorter run time) if either of the two operands of the multiplication are 0. Therefore by monitoring the run time of a multiplication instruction, you can learn whether one or both of the operands is 0.

The digit serial multiplier similarly optimizes multiplication by taking varied length execution paths depending on whether each byte of the multiplier operand (`b` operand in multiplication `a*b`) is 0. Therefore if each of the 4 bytes of `b` are non-zero, then the multiplication will take 4 clock cycles. If the top 8 bits of `b` are zero, but the bottom 24 bits are non-zero, the multiplication instruction will take 3 cycles. If the top 16 bits of `b` are zero, and the bottom 16 bits are non-zero, the multiplication instruction will take 2 cycles. Lastly, if the top 24 bits of `b` are zero, the multiplication will take 1 clock cycle. Note that the multiplication will always take at least one clock cycle, even if the 8 least significant bits are zero.

The third leakage function we look at is CVA6 division. In this division implementation, a division instruction will take 1 of 66 different paths, each with unique numbers of cycles. Say you have a dividend `a` and divisor `b`, such that the division is `a//b` (note `//` signifies integer division). If `a<b`, then the instruction will take 1 cycle. If `a>b` but `a` and `b` have the same number of leading zeros, then the instruction will take 2 cycles. Lastly, if `a>b` and `a` has `d` more leading zeros than `b`, then the division instruction will take `d+2` cycles.

For each of the above leakage functions, I generated the conditional probability of each of the observable clock cycle counts. From the probability distribution, I computed the theoretical leakage using mutual information and maximal leakage. For each of these leakage function, we originally assume that both of the inputs are randomly distributed. However, if an operand can be attacker-controlled, that may change the conditional probability, and likely increase the leakage computation. Therefore in one case where attackers can specify the inputs in order to maximize the leakage functions, we relax this assumption and allow non-secret inputs to be attacker controlled.

B. Cryptographic program

In order to validate our methodology end-to-end, we consider one cryptographic program, the Poly1305 hashing algorithm for secure message authentication from OpenSSL [6]. This program was chosen as it includes many multiplication instructions where one of the two 32 bit operands comes from a secret key. Therefore this program was a good candidate to evaluate the two multiplication leakage functions, zero-skip multiplication and digit serial multiplication.

Poly1305 computes a 16 byte hash from a 16 byte key, `r`, and an *L* byte message, `m`. Each loop of the hashing algorithm processes 16 bytes of the message `m`, in which the four 32 bit blocks of the key, `r`, (or a modified version of the key) are multiplied by the hash value, `h`, of the previous iteration. Each multiplication's second operand is a 32 bit integer derived from the secret key: either it is a 32 bit block of key, or a modified value `s`, where `s` is defined as `s = r' + (r'>>2)` and `r'` is a 32 bit block of the key. There are at least 16 unique multiplications in each loop of the hashing algorithm, each with 1 secret operand. Note in evaluating the program, we assume that the attacker is maximally powerful, in that they can detect the exact number of cycles of each of the multiplication instruction. Also note that the 16 byte key requires that each 32 bit block must have the top 4 bites be 0. Additionally the top 3 32 bit blocks must have the bottom 2 bits be 0. This restricts the secret key to 2^{106} distinct values.

In future work, we hope to also evaluate RSA, as it includes a division instruction in order to evaluate the CVA6 division leakage function, as well as other cryptographic programs for other leakage functions.

V. Results

A. Individual instruction leakage computation

1) Zero-skip multiplier: For the 32 bit zero skip multiplier with both operands randomly distributed and one of the two operands is secret, `s`, we found the following conditional probabilities:

	$s = 0$	$s \neq 0$
p(Y=fast|s)	1	$\frac{1}{2^{32}}$
p(Y=slow|s)	0	$\frac{2^{32}-1}{2^{32}}$

Using these conditional probabilities, we can compute the maximal leakage, L_{ML} and mutual information, L_{MI}:

$$L_{ML} = 0.9999999998320482$$

$$L_{MI} = 7.320822924519409e - 09$$

Next we modify the 32-bit zero-skip multiplier leakage model to be an active side channel. The secret operand, `s`, is still randomly distributed, but the non-secret operand is attacker controlled. For an ideal attack strategy, the attacker would choose the second input value to be non-zero, in order to increase the information leakage of the secret input. Therefore we have the following conditional probabilities and leakage quantities:

	$s = 0$	$s \neq 0$
p(Y=fast|s)	1	0
p(Y=slow|s)	0	1

Using these conditional probabilities, we can compute the maximal leakage and mutual information:

$$L_{ML} = 1$$

$$L_{MI} = 7.786484211772939e - 09$$

From these results, we can see that the maximal leakage is significantly larger than the mutual information, as the mutual information is very close to 0. Further we can see that an attacker controlled second input only increases the leakage very slightly in both metrics.

2) Digit serial multiplier: Now consider a 32-bit digit serial multiplier where the secret operand, multiplier `s`, is randomly distributed, and where values for Y are the number of cycles. We found the following conditional probabilities, where the columns are the probabilities of each observation conditioned on the number of upper most bits of secret multiplier being 0:

	Top 24 bits 0	Top 16 bits 0	Top 8 bits 0	else
p(Y=1|s)	1	0	0	0
p(Y=2|s)	0	1	0	0
p(Y=3|s)	0	0	1	0
p(Y=4|s)	0	0	0	1

Using these conditional probabilities, we can compute the maximal leakage and mutual information:

$$L_{ML} = 2$$

$$L_{MI} = 0.037142090494774085$$

Since the digit serial multiple only depends on the second operand of the multiplication instruction and no other instruction operands, it is by default a passive side channel. Therefore there is no ideal strategy that an attacker can use to increase the leakage.

The leakage values for the digit serial multiplier are significantly larger than the zero-skip multiplier: the maximal leakage is twice as large, and the mutual information is over 6 orders of magnitude larger.

3) CVA6 division: Since there are 66 different execution paths and each execution path is a unique attacker observation, there are far too many possibilities for a complete conditional probability table. Instead, I will define conditional probability as a piece-wise function. Assume you have some 64 bit secret, s, which is the dividend in the division, and s has z leading zeros. Then the conditional probabilities of observations where Y is the number of cycles for the division instruction:

$$P(Y=1|s) = \frac{2^{64}-1-s}{2^{64}}$$

$$P(Y=2|s) = \frac{(s-2^{64-z-1}+1)}{2^{64}}$$

$$P(Y=d+2, d \geq 1|s) = \begin{cases} \frac{2^{64-z-d-1}}{2^{64}}, & z+d<64 \\ \frac{1}{2^{64}}, & z+d=64 \\ 0, & otherwise \end{cases}$$

Using the conditional probability distribution, I computed the following maximal leakage and mutual information:

$$L_{ML} = 0.9971794809376213$$

$$L_{MI} = 0.3126548229683998$$

Note that we assumed that the dividend is the secret, but this can be easily extended such that the divisor is the secret.

We can see that division has significantly higher mutual information than the two multiplication functions but similar or less maximal leakage. This allows us to conclude that it is at least worthwhile to revisit these metrics in order to have the most accurate measurement of leakage for a particular instruction.

B. Program leakage analysis

Using symbolic execution tool KLEE, I analyzed the multiplication instructions in Poly1305, under the two multiplication leakage models: zero-skip multiplication and digit serial multiplication. There are two primary considerations when looking at the leakage functions in a program context. First, we hope to examine how a secret value, the key `r` in this program, is transformed before it is passed to the multiply instructions. As described in the IV.B, we are considering the transformation of a 4 byte block of secret key `r'` to be `s = r' + (r'>>2)`. Second, we hope to consider how multiple multiplication instructions contribute to the overall leakage of the program. Note that these program analysis results are a work in progress, so I am providing intermediate results here.

1) Zero-skip multiplier model: Under the zero-skip multiplier model, transforming the secret does not affect the observation based on the leakage function. This is a result of the restricted key values, described in IV.B. We proved with KLEE that for `r` equal to 0, the transformed `s` value must also be zero, and for `r` not equal to 0, `s` is not equal 0. Since the observation space is not affected by the transformation of the key, the leakage is identical when multiplying by `s` as when multiplying by `r'`.

We also wanted to look at how repeated multiplications of can affect leakage. Since zero-skip multiplication depends on both operands of the multiplication, it is possible to learn additional information with iterative multiplication instructions. Say for example on the first multiplication instruction with `r`, the attacker sees a fast path. This means that one or both operands are 0. We know that each loop of the Poly1305 algorithm is multiplying the intermediate hash `h` by `r`, and `h` can change every iteration of the loop while `r` does not. Therefore if the next iteration of the loop is not a fast path, we know that `r` must not have been the zero operand. Therefore iterative multiplications can increase the leakage of the secret key `r`. However, we are still working on a way to formalize this notion mathematically.

2) Digit serial multiplier model: Under the digit serial multiplication model, the transformation of 32 bit blocks of the key `r'` to `s` can change the observation by the leakage function. For example, assume that `r'` had its top 8 bits be 0, such that the leakage function determined the observation would be 3 cycles. Using KLEE, we showed that there are values of `r'` such that `s = r' + (r'<<2)` does not have its top 8 bits be 0. Therefore this transformation of the key value changes the output observation of the leakage function to be 4 cycles. Only a subset of the values of `r'` resulting in one observation will have a corresponding `s` which results in a different observation. Therefore more information about the secret is leaked when the attacker can observe both multiplications by both `r'` and `s`, which is the case in Poly1305.

However, the digit serial multiplication leakage function only depends on the second operand in the multiplication. In the case of Poly1305, that is the key `r'` (or modified `r'` as `s`). Therefore, there is no more information you can learn from repeating multiplication with different second operands, and thus repeated multiplication under the digit serial leakage model does not increase the leakage.

VI. Conclusion

In this project, we used symbolic execution, information theory, and program analysis in order to reason about the leakage of instructions under various leakage function models. Our methodology for quantifying leakage for a particular program under a leakage model will allow programmers to more accurately account for the leakage of transmitter instructions when designing software. This is useful as it allows hardware and software mitigation techniques to be more exact depending on the security properties of the side channels. As it is unrealistic to expect hardware to be free from side channels, it is imperative that side channels are modeled accurately so that they can be mitigated without prohibitive overhead. Therefore using leakage functions, software and compilers can more easily be tuned to prevent side channel leakage.

In future work, we intend to analyze more leakage functions from other hardware optimizations, including dependent side channels, where the leakage function depends on other executing instructions. We also want to extend the program analysis to programs with instructions under each leakage model. Further, we want to be able to extend the quantification metrics to provide leakage estimates for an entire program. Lastly, we would like to prove the leakage functions are accurate with simulation, in order to show that a piece of hardware which implements a leakage function leaks the expected information.

VII. Github

https://github.com/samanthaarcher0/Leakage-Quantification

References

[1] J. R. Sanchez Vicarte et al., ”Opening Pandora’s Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data,” 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 2021, pp. 347-360, doi: 10.1109/ISCA52012.2021.00035.

[2] P. W. Deutsch, W. T. Na, T. Bourgeat, J. Emer, and M. Yan, “Metior: A Comprehensive Model to Evaluate Obfuscating Side-Channel Defense Schemes,” 2023 ACM/IEEE 50th Annual International Symposium on Computer Architecture (ISCA), Jun. 2023, doi: 10.1145/3579371.3589073.

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

[4] Grobschadl, J., Oswald, E., Page, D., and Tunstall, M. (2009). Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications. Cryptology ePrint Archive, Paper 2009/538. https://eprint.iacr.org/2009/538

[5] Zaruba, F., and Benini, L. (2019). The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 27(11), 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114

[6] “OpenSSL: Cryptography and SSL/TLS toolkit,” 2021, https://www.openssl.org/.