Efficient and Generic Microarchitectural Hash-Function Recovery

Lukas Gerlach[1], Simon Schwarz[2], Nicolas Faroß[3], Michael Schwarz[1]
[1] CISPA Helmholtz Center for Information Security [2] Saarland University, Saarland Informatics Campus
[3] Saarland University, Department of Mathematics

***Abstract*—Modern CPUs use a variety of undocumented microarchitectural hash functions to efficiently distribute data within microarchitectural structures such as caches. A well-known function is the cache slice function that distributes cache lines to the slices of the last-level cache. Knowing these functions considerally improves microarchitectural attacks, such as Prime+Probe or Rowhammer. However, while several such linear functions have been reverse-engineered, there is no generic or automated approach for reverse-engineering non-linear functions, which are common with modern CPUs.**

In this paper, we introduce a novel generic approach for automatically reverse-engineering a wide range of microarchitectural hash functions. Our approach combines techniques initially used for logic-gate minimization and from computer algebra to infer the hash functions based on input-output pairs observed via side channels. With our framework, we infer 3 previously unknown non-linear hash functions on both AMD and Intel CPUs, including the new Alder Lake hybrid-CPU architecture. We verify our approach by reproducing known hash functions and evaluating side-channel attacks that rely on these functions, resulting in success rates above 97.65 %. We stress the need to design such functions with both performance and security in mind and discuss alternative designs that can be used in future CPUs.

1. Introduction

To ensure that new CPU generations outperform their predecessors, CPU vendors continually improve the microarchitecture. A common method is to add more caches and predictors to the microarchitecture. Both caches and predictors rely on data structures that must be indexed efficiently. Such indexing heavily relies on microarchitectural hash functions. These hash functions are *not* cryptographically secure but ensure that data is uniformly distributed within these data structures. Typically, these hash functions use virtual or physical addresses of data as input and they output an index into a microarchitectural data structure. These hash functions are designed to balance the load on the data structure and to reduce collisions when linearly accessing memory. A well-known example of such a hash function is the cache-slice function [35], which distributes data uniformly to the slices of the last-level cache.

These hash functions are not only relevant for the performance of the CPU but also have a significant impact on microarchitectural attacks. Microarchitectural attacks, such as Prime+Probe [40], DRAMA [41], TLBleed [21], Collide+Probe [33], or Spectre [29] require knowledge of the hash function to choose addresses mapping to a specific part of a microarchitectural element [17]. Without knowledge of the hash function, attacks are often less effective [41], [51]. In addition, the knowledge of these hash functions enables performance improvements by compiling cache-aware code [36] and allows for cache coloring [66], [53]. However, these functions are typically undocumented.

Previous work reverse-engineered various hash functions, including those used for cache slices [52], [35], [27], [34], [25], [26], [65], [36], [64], DRAM addressing [51], [41], [63], [24], [59], AMD's way predictor [33], and TLB sets [21], [30]. The approaches for reversing the functions have in common that they collect a large number of inputs and corresponding outputs and try to infer the hash function from these pairs. For the data recording, various side-channel measurements [41], [33], [22], [26], [59], [63] or CPU interfaces [35], [24], [36] have been used. However, as the general problem of inferring a compact function representation from inputs and outputs has shown to be a hard problem [11], [57], previous work mainly focused on linear hash functions, i.e., functions that only consist of a single layer of XORs. All previous non-linear microarchitectural hash functions have been reverse-engineered by manually discovering patterns or by assuming a specific structure of the function [65], [36], [26].

This paper presents the first generic reversing approach to infer microarchitectural hash functions from input-output pairs. We handle both linear and non-linear functions without relying on specific assumptions about the hash function. To this end, we employ a function-minimization pipeline that finds small representations for a large class of boolean functions. We find compact representations for most of our analyzed microarchitectural hash functions. This confirms the assumption that microarchitectural hash functions are usually of low depth and complexity, as deep functions would needlessly increase the latency of the function.

Our function-minimization pipeline consists of 3 main steps: First, linearity of input bits is checked. Using a heuristic check allows to reduce the sample size of input-output pairs significantly. Since many hash functions are linear in at least some bits, this insight makes collection feasible for larger address input sizes. Next, the sampled truth table is converted to a disjunctive normal form (DNF). On this formula, the classical ESPRESSO heuristic logic minimizer [8] is used to pre-minimize the DNF. Compared to the naïve representation as a truth table, the pre-minimized DNF representation is already significantly smaller in practice. Still, a representation in DNF has exponential blowup for functions that heavily use exclusive-or operations. Hence, in the last step, we use a novel function minimization technique based on Gröbner bases. Gröbner bases are an important tool in computer algebra and can be used to simplify or solve systems of polynomial equations. Thus, we encode our pre-minimized DNF into such a system of equations, which can be further minimized by computing a Gröbner basis. The resulting Gröbner basis can be interpreted again as a boolean function. This process can be continued recursively to yield even more compact function representations.

Our approach is also the first automated approach to infer non-linear hash functions, such as those used for cache slices, without relying on specific structures [65] or large look-up tables [36]. We demonstrate our approach on multiple non-linear cache-slice functions for which no hash function is known. Our functions also include the non-linear cache-slice function of AMD Zen3(+) CPUs, which we measure using a timing side channel. In total, we provide 10 hash functions inferred using our approach. To experimentally verify the correctness, we implement a Prime+Probe attack on OpenSSL T-tables on Intel CPUs using the reverse-engineered cache-slice function. Our attacks achieve success rates above $99.9\,\%$, demonstrating the correctness of the slice function.

We additionally demonstrate our approach by using it on a wide range of previously reverse-engineered hash functions. For all functions analyzed, we successfully infer an equivalent and compact hash function. Our work is fully automated without requiring any adaptions or assumptions for specific targets. Moreover, we show that 2 manually reverse-engineered functions can be represented in an even simpler form. In addition to being generic, our approach is also fast enough to be used in practice. All hash functions are inferred within a day, given the required input-output pairs. Hence, for most hash functions, collecting these input-output pairs is the limiting factor, not the recovery of the hash function. Given the resulting small hash functions, we assume that our result is close to the hardware implementation.

In addition to the solving approach, we present an open-source framework for measuring the input-output pairs of cache slices, AMD's way predictor, and DRAM addressing. With its modular design, it can be adapted easily for other hash functions. The output of this framework can be used directly by our solving approach to infer the underlying hash function automatically. The framework handles linear and non-linear functions without requiring knowledge about the structure or properties of the function. With this framework, we show the first reverse-engineering of the cache-slice function on the new hybrid CPU architecture introduced with Alder Lake. On this hybrid CPU architecture, one cache slice is used by either 1 or 4 cores, increasing the complexity of the hash function. Even CPUs with 16 cores, such as the Intel Core i9-12900K, have a non-linear cache-slice function that distributes addresses among 10 slices.

Based on the results, we discuss several use cases for the reverse-engineered functions, including more effective attacks and defenses. We propose secure and fast alternatives to the currently used microarchitectural hash functions. Our proposed hash functions keep the benefits of currently-used functions but are more difficult to reverse-engineer, which impairs future microarchitectural attacks.

To summarize, we make the following contributions:

1) We systematically analyze the currently used approaches for inferring microarchitectural hash functions, showing their limitations in measurement and solving approaches.
2) We design a framework to overcome these limitations with novel measurement and solving techniques, supporting linear and non-linear functions.[1]
3) We evaluate our framework by inferring new microarchitectural hash functions, among those the first ones on Intel Alder Lake hybrid cores and AMD processors, and finding new minimal forms for known functions.

Outline. Section 2 provides background. Section 3 discusses approaches and limitations of previous approaches. Section 4 introduces our automated approach for recovering hash functions. Section 5 introduces the measurement framework. Section 6 evaluates our approach. Section 7 discusses alternative hash functions. Section 8 discusses limitations and future work. Section 9 concludes.

2. Background

In this section, we provide the necessary background about hash functions, linear functions, logic minimization and Gröbner bases.

2.1. Hash Functions

Hash functions are mathematical functions that take an input and map it to a fixed-size output, generally referred to as the hash value. In this paper, we focus on hash functions with fixed input sizes. In contrast to cryptographic hash functions, microarchitectural hash functions do not provide any of the classical security guarantees. Instead, they are mainly designed to load-balance by distributing data efficiently, as in the LLC or DRAM. Formally, our considered hash functions map an input $\{0,1\}^b$ to a bucket $\{0,\dots,n-1\}$.

Notably, for load-balancing purposes, microarchitectural hash functions usually have a low complexity. As the hash functions are often evaluated (e.g., on every memory access), low latency of the function is of great importance as it has to keep up with the speed of modern CPUs. Hence, microarchitectural hash functions can usually be implemented by a shallow circuit with few gates. Still, to ensure that they balance the load, the function should have a uniform output in $\{0,\dots,k\}$ and, for small changes in the input,

1. Source code can be found at: https://github.com/CISPA/Microarchitectural-Hash-Function-Recovery

the output should change. Such a function is represented by a circuit in hardware. By analyzing every output bit of the circuit *independently*, we can simplify our analysis to functions $h : \{0,1\}^b \to \{0,1\}$.

2.2. Linear Functions

When analyzing hash functions $h : \{0,1\}^b \to \{0,1\}$, previous work [35], [27], [34], [25] focused mainly on *linear* functions. A function h is linear, if it is an exclusive-or combination of its inputs, i.e., for $a_1, \dots a_n \in \{0,1\}$, h has the shape $h(x_1, \dots, x_n) = (a_1 \wedge x_1) \oplus \cdots \oplus (a_n \wedge x_n)$. Such an equation can be trivially solved by solving a system of linear equations for $a_1, \dots, a_n$. In particular, n measurements are sufficient to completely reconstruct linear functions.

In contrast, non-linear boolean functions can contain additional, arbitrary operators. Compared to linear functions, the function must be evaluated on all 2^n possible inputs to determine it uniquely. Furthermore, finding a compact representation is significantly more difficult.

2.3. Logic Minimization

Multiple methods exist to minimize a boolean function specified by its output on all possible input combinations. From the specification, one can easily build a disjunctive normal form (DNF) of the corresponding formula by building a disjunction over all values where the function evaluates to true. However, in general, this representation is exponentially big in the number of input bits.

Exactly finding a minimum equivalent formula is NP-hard (it even is Σ_2^P-complete) [57]. Still, some approaches exist for boolean function minimization, most notably the work from Quine and McCluskey [45], [37], which optimally minimizes a function in DNF. This approach is refined by the heuristic-driven ESPRESSO [8] for DNF minimization. While ESPRESSO does not guarantee minimal functions, in practice, it provides near-to-optimal results. This heuristic trade-off provides a reasonable runtime for ESPRESSO. Therefore, ESPRESSO is widely used in logic synthesis tools or to perform logic minimization before synthesizing a design. However, ESPRESSO can only minimize formulas to DNF. In practice, this leads to an exponential blowup for some functions, in particular for functions that use the $\oplus$-operator. This motivates so-called multi-level logic optimization, where output formulas can have arbitrary shape. Prior work usually focused on minimizing cost while synthesizing circuits in hardware. To this end, techniques based on, e.g., if-then-else constructions [28] or decomposition and factorization [7] are employed. A more detailed overview is given by Brayton et al. [6]. Recent approaches [55] make use of SAT-solvers for optimal multi-level minimization. However, in practice, the runtime becomes infeasible for instances with more than 6–7 variables. This motivates our heuristic function minimization approach based on Gröbner bases.

2.4. Gröbner Bases

The computation of Gröbner bases is a powerful technique used in computer algebra and has found widespread application in commutative algebra and algebraic geometry. Mathematically, the theory of Gröbner bases generalizes polynomial division for systems of equations with multiple variables. A Gröbner base then yields a reduced representation of the system of equations with respect to polynomial division. For a more precise mathematical description of Gröbner bases and the underlying algorithms, we refer to related work [10], [3], [13].

In the context of logic minimization, Gröbner bases have not been used previsousely. However, there has been some progress in SAT-solving, where Gröbner bases were used to simplify sets of input clauses in a pre-processing step [12], [38]. These approaches use a similar encoding technique, but are not designed for logic minimization.

3. Microarchitectural Hash Functions

In this section, we provide a systematic overview of known microarchitectural elements using a hash function and the corresponding measurement and solving approaches. This overview, also summarized in Table 1, motivates the need for a generic automated approach and also defines the requirements for a measurement framework and a solving approach.

3.1. Cache Slices

Since the Sandy Bridge microarchitecture, Intel CPUs divide the last-level cache into cache slices. Until the Skylake microarchitecture, each *physical* CPU core had one cache slice. Since Skylake, each *logical* CPU core has one cache slice [9], i.e., , hyperthreads share one slice. However, with the introduction of hybrid E- and P-cores on Alder Lake, the slice count per core has changed again. While every P-core has one slice, 4 E-cores share one slice. AMD also uses cache slices since the Zen microarchitecture [1]. To map a physical address to a cache slice, a microarchitectural hash function is used. The primary purpose of this function is to balance the pressure on the cache slices.

In the following, we provide an overview of current approaches to collect measurement data and infer the cache slice function. For AMD, there is no information about the slice function, hence we do not include these CPUs in the overview. However, we show the first reverse-engineering of this function in Section 5.3.2. We note that so far, generic approaches only exist for linear functions on Intel CPUs. Intel CPUs split into E- and P-cores, and CPUs with a non-power-of-two slice count use non-linear functions. Hence, previous approaches for CPUs where the number of cores is a power of two no longer work on these new CPU designs.

Takeaway In modern hybrid CPU architectures (e.g., Intel Alder Lake), the number of slices is no longer a multiple of the number of CPU cores.

TABLE 1: Types and properties of microarchitectural hash functions.

Function	Linear	Mapping (input → output)	Vendor	Previous Work
Cache slice	some	Physical address → cache slice	Intel, AMD	[52], [35], [27], [34], [25] [26], [65], [36], [64]
DRAM mapping	all	Physical address → DRAM bank	Intel, AMD, ARM	[51], [41], [63], [24], [59], [23]
Way predictor	all	Virtual address → μtag	AMD	[33]
TLB Sets	all	Virtual address → TLB set	Intel	[21], [30]

Measurement Two different measurement methods have been established to measure the cache-slice function. In both cases, the output of the cache-slice function can only be observed indirectly. First, performance counters can be used to count the number of accesses for every cache slice [35], [36]. By accessing the address of interest many times, the slice with the highest number of accesses is likely the correct slice. As a cache flush needs to acess the cache slice to check whether a writeback needs to be performed, a flush instruction can be used for cache acess, eliminating the need for eviction-set generation. As the performance counters for cache slices are subject to change for different microarchitectures, we provide a list of relevant model-specific registers (MSRs), including configuration values for microarchitectures from Sandy Bridge to Alder Lake in Section A. Second, a timing side channel can be used [22] to detect whether an address maps to the slice of the current CPU. Another indirect timing side-channel is possible via eviction sets [25], [26], [27]. By verifying if an eviction set still works after replacing one address with a different address from the same cache set, an attacker can infer if the address maps to the same cache slice.

Solving Solving the function has been automated in the case of linear slice functions [35], [27]. To solve a linear function, an attacker performs an optimized brute-force search by using the commutative and associative properties of the `xor` operation. A more efficient approach used by Irazoqui et al. [27] is to interpret the function as a polynomial over $\mathbb{F}_2$ and solve it to retrieve the function. Finding a minimal form for a non-linear function, in contrast, is a hard problem. All current approaches rely on manual reverse-engineering after a sufficient number of measurements have been dumped. Inci et al. [26] work with assumptions on the depth and degree of the polynomial used for the non-linear function based on the distribution of addresses among the slices. They generate such functions until they find an equivalent function resulting in the same distribution. Yarom et al. [65] and McCalpin [36] look for patterns in the slice number over a contiguous memory region, and identify index permutations to transform an arbitrary start pattern into a target pattern. However, McCalpin's approach [36] only works due to the specific structure of the currently-used function and still requires a look-up table for the start pattern. In Section 4, we solve linear and non-linear functions with our approach, without relying on specific structures, distributions, or look-up tables. We can even further simplify the manually-inferred functions of McCalpin [36].

Takeaway Current measurement and solving approaches are limited to Intel CPUs and do not work if there is more than one slice per core.

3.2. DRAM Addressing

DRAM addressing functions have been a part of microarchitectural attacks in the past, both for side-channel attacks [41], [60] and Rowhammer attacks [51], [4], [47], [18], [20]. These functions map a physical memory address to a DRAM bank. Current approaches assume that the function is linear. However, we expect non-linear variants, as announced DDR5 modules support non-power-of-two memory sizes [54].

Measurement There are 3 approaches for obtaining measurement data for the DRAM addressing function. All 3 approaches have in common that they only observe indirect information, i.e., the output of the addressing function for a given input, not the function itself. Pessl et al. [41] first employed hardware measurements using an oscilloscope to obtain a ground truth for labeling the addressing functions. They additionally proposed a timing side channel to measure in software, which was also used by Xiao et al. [63] and Wang et al. [59]. Helm et al. [24] relied on DRAM performance counters available on Intel Xeon CPUs to determine the DRAM bank of an address.

Solving Solving the currently-used DRAM functions is comparatively simple, as they are all linear. Still, multiple approaches have been proposed. Pessl et al. [41] use a generic brute-force method to infer the DRAM functions. Additional graph-based methods [63] have been used too to infer the DRAM addressing function. While the graph-based methods provide a significant speed-up, they assume that every address bit is used in at most 2 output bits of the function. Wang et al. [59] rely on domain knowledge and the assumption that the function only consists of XORs to improve the performance of the brute-force search for functions.

Takeaway Current DRAM reversing approaches only work if the involved functions remain simple XORs of address bits.

3.3. Cache Way Predictor μtag

On AMD CPUs, the cache-way predictor reduces the cache power consumption by predicting in which cache way an address is located [33]. Each virtual address is tagged with a μtag before entering the cache. This tag can then be used to predict the cache way an address maps to before the address is translated and the physical address becomes available. Lipp et al. [33] reverse-engineer the microarchitectural hash function used to compute the μtag and leverage their results to perform different cache attacks.

Measurement Two virtual addresses mapping to the same μtag cause a delay in the way predictor, which can be observed via timing measurements. To reverse-engineer the μtag, a complete truth table for the bits relevant to the function is needed. For the analyzed CPU generations, i.e., Bulldozer to Zen 2, the μtag hash function produces an 8-bit output. The measurements for the truth table can be obtained by either randomly measuring until sufficient measurements are taken or by systematically constructing a minimal set of measurements covering all input combinations for the μtag hash function. We describe the approach to systematically measure any microarchitectural hash function in Figure 2.

Solving Lipp et al. [33] assume that the μtag hash function is a linear function consisting only of XORs of the address bits. As they can recover the hash function by solving a system of equations over $\mathbb{F}_2$, this assumption is verified. However, there is no necessity that the μtag hash function is linear. To reduce the number of hash collisions, the hash output can be increased to any size. If the output size is not a power of two, this hash function becomes a non-linear function.

Takeaway Current way-predictor reversing approaches only work because AMD currently uses a power of two for the size of the μtag.

3.4. TLB Set

The mapping between virtual addresses and the TLB is controlled via a microarchitectural hash function. Gras et al. [21] reverse-engineered this mapping between virtual addresses and TLB entries for Sandy Bridge and Coffee Lake processors for the first and second TLB levels. Koschel et al. [30] provided a function for the second-level TLB on Skylake processors.

Measurement Measuring the TLB mapping function can be done by means of timing measurement or performance counters tracking TLB misses [21], [30]. Similar to the previously described functions, performing timing measurements requires minimal assumptions but adds noise. Measurements using the performance counters are generally more stable but require an architecture that exposes the performance counters relevant for tracking the TLB state.

Solving In the case of a linearly-mapped TLB, inferring the mapping function is trivial. Addresses directly map to TLB entries and wrap around when the maximum capacity of the TLB is reached. For more complex but linear functions [21], [30], it is possible to brute force the function or solve a system of equations over $\mathbb{F}_2$.

4. Generic Hash-function Recovery

In this section, we describe our generic approach to infer an underlying boolean formula of a microarchitectural hash function. Figure 1 provides an overview of our approach. While the way input-output pairs are obtained differs by application, we describe a generic method applicable to all microarchitectural hash functions and illustrate it in Figure 2.

Formally, we are given a black-box function $h : \{0,1\}^b \to \{0,\dots,n\}$. A key idea for the analysis of these functions is to analyze the function with regard to the *binary representation* of the output. Hence, instead of analyzing h, we reconstruct the functions $h_i\colon \{0,1\}^b \to \{0,1\}$ independently, where h_i denotes the i-th bit in the binary representation of the output of h. For a given bit i, the specification $\left\{(x, h_i(x)) \mid x \in \{0,1\}^b\right\}$ is the *truth table* of the function h_i.

In the following, we restrict our view to simple boolean functions $h : \{0,1\}^b \to \{0,1\}$. After all input-output pairs are obtained, each input bit of the function is heuristically checked for linearity (cf. Section 4.1). Linear bits can be eliminated in the preprocessing, which significantly improves runtime in almost all cases. In the next step, we build and minimize a disjunctive normal form of the formula, detailed in Section 4.2. Afterward, our novel minimization technique with Gröbner bases further simplifies the formula (Section 4.3). This minimization pipeline is then called recursively on subterms until no smaller overall representation is found (Section 4.4).

4.1. Linearity Check

The first step in our pipeline aims to pre-simplify functions even before explicitly collecting all input-output pairs. To this end, we consider two cases: (a) the i-th bit of the input does not influence the output of h at all (it is *irrelevant* for the output), or (b) the i-th bit always flips the output.

Formally, this is captured by the following definition: We call a boolean function h *linear* in bit i if for all values of $x_1,\dots,x_{i-1}$ and $x_{i+1},\dots,x_b$, either

$$\begin{aligned} & h(\dots,x_{i-1},0,x_{i+1},\dots) = h(\dots,x_{i-1},1,x_{i+1},\dots) \\ \text{or}\quad & h(\dots,x_{i-1},0,x_{i+1},\dots) \neq h(\dots,x_{i-1},1,x_{i+1},\dots) \end{aligned}$$

Note that in such a case, the function h can always be expressed as $h(x) = h'(x) \oplus (a_i \wedge x_i)$, where the value of h' is independent of x_i. In particular, in case (a) it holds $a_i = 0$, whereas in case (b) $a_i = 1$. Hence, in such a case, it is sufficient to find a representation for $h'(x)$. Note that this effectively halves the space needed to represent the truth table. Thus, eliminating linear bits during the preprocessing can significantly impact the runtime of the method.

Explicitly *collecting* the values for a truth table for a specific function quickly becomes infeasible, as the required memory grows exponentially in the number of input bits. Hence, we incorporate a *heuristic check* for linearity already before the collection of input-output pairs. To

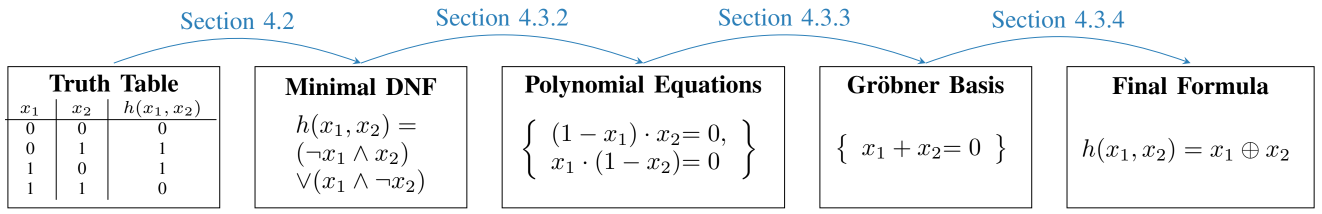

Section 4.2
Section 4.3.2
Section 4.3.3
Section 4.3.4
Truth Table
x_1 x_2 $h(x_1, x_2)$
0 0 0
0 1 1
1 0 1
1 1 0
Minimal DNF
$h(x_1, x_2) = (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)$
Polynomial Equations
$\{ (1 - x_1) \cdot x_2 = 0, x_1 \cdot (1 - x_2) = 0 \}$
Gröbner Basis
$\{ x_1 + x_2 = 0 \}$
Final Formula
$h(x_1, x_2) = x_1 \oplus x_2$

Figure 1: Overview of the function recovery pipeline. First, a truth table is transformed into a DNF. Afterward, the DNF is interpreted as a system of polynomial equations, for which a Gröbner basis is computed. Finally, a smaller boolean formula is recovered from the Gröbner basis.

check the i-th bit for linearity, we perform the following method: We sample $i_0 = h(a_1, \ldots, a_{i-1}, 0, x_{i+1}, \ldots)$ and $i_1 = h(a_1, \ldots, a_{i-1}, 1, x_{i+1}, \ldots)$ for 100 random vectors $(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)$. If $i_0 = i_1$ for all random vectors, or $i_0 \neq i_1$ for all random vectors, we conclude that bit i is linear. This check is implemented in our framework (cf. Section 5.2).

This approach is only a heuristic approximation for linearity. However, the load-balancing property of hash functions makes it highly unlikely that a bit is incorrectly labeled as linear. If bit i is relevant, we expect it by the load-balancing property to influence the output for a significant share of inputs.

If the i-th bit fulfills the heuristic linearity check, it can be ignored for measurement. Only in the last step, the final hash function $h(x) = h'(x) \oplus (a_i \wedge x_i)$ is constructed. All steps in between focus on reconstructing $h'(x)$. In particular, note that for a linear function that only consists of exclusive-or operations, this step already reconstructs the complete formula.

4.2. DNF Minimization

After preprocessing and collecting input-output pairs, a truth table $T = \{(x, h(x)) \mid x \in \{0,1\}^b\}$ is obtained. We transform this truth table to a *disjunctive normal form* (DNF) over the variables $x_1, \ldots x_n$ in the canonical way: First, we define a primitive $\mathrm{def}(v_1, \ldots, v_b)$,which is true if and only if the input is $(v_1, \ldots, v_b)$:

$$\mathrm{def}(v_1, \ldots, v_b) = \bigwedge_{i \in \{1, \ldots, b\}} \begin{cases} x_i \text{ if } v_i = 1 \\ \neg x_i \text{ if } v_i = 0 \end{cases}$$

Next, we define our overall formula as a disjunction over all inputs that are true in T:

$$h(x) = \bigvee_{((v_1, \ldots, v_b), 1) \in T} \mathrm{def}(v_1, \ldots, v_b)$$

Note that, by definition, the function $h(x)$ fulfills our truth table directly: For every output that is specified as true, exactly one disjunct evaluates to true, making the overall formula true. In general, the resulting DNF has a size exponential in b. In comparison, we expect the microarchitectural hash function to be implemented by a circuit of low complexity. Hence, the formula should have a compact form. In the following, we present additional steps to minimize the formula, bringing it closer to the actual hardware implementation. In the first step, the constructed DNF is minimized with the ESPRESSO [8] algorithm. Afterward, we make use of Gröbner bases to find a more compact representation. Minimizing the DNF with ESPRESSO is not strictly required for the second step. However, the ESPRESSO algorithm is quite efficient and can significantly reduce the size of the formula that is passed to the Gröbner basis computation. This improves the combined runtime of both steps.

Still, note that a disjunctive normal form can be far from the optimal representation of a formula. In particular, even linear formulas can only be represented with exponential overhead in DNF. For example, the most compact DNF representation of the formula $f(x_1, \ldots, x_n) = x_1 \oplus \cdots \oplus x_n$ in DNF is already exponentially larger in n than f. As our evaluated microarchitectural hash functions make heavy use of linear operations, further processing of the DNF is necessary.

4.3. Gröbner Minimization

After obtaining a minimal DNF, the second step in our minimization pipeline is based on Gröbner bases. Notably, Gröbner basis computations can be used to find a simpler representation of a system of polynomial equations. To take advantage of this property, we implement the following process: First, we encode our (minimized) DNF as a system of polynomial equations. Next, we simplify the resulting system of polynomial equations by computing a Gröbner basis. This process yields another system of polynomial equations, which is then again converted to a boolean formula. To further reduce the size of terms, we apply this process recursively to smaller subterms. During this process, we treat the implementation of the Gröbner algorithm mostly as a black box.

4.3.1. Encoding a Boolean Formula. First, the input DNF must be encoded as a system of polynomial equations. For this encoding, we first construct an underlying mathematical ring that can express boolean functions. We than show, a method for encoding a DNF as a system of polynomial equations in this ring. Consider the polynomial ring $R = \mathbb{F}_2[x_1, \ldots, x_n]$. Within this ring, all boolean functions

TABLE 2: Encoding of boolean operations in $\mathbb{F}_2$

Boolean operation	**Equivalent in $\mathbb{F}_2$**
$\neg x$	$(1-x)$
$x \wedge y$	$x \cdot y$
$x \oplus y$	$x + y$
$x \vee y$	$1-(1-x)\cdot(1-y)$

in the n variables $x_1, \dots, x_n$ can be expressed [42]. The conversion is an extension of the mapping described in Table 2, allowing a straightforward translation of formulas to polynomials in $\mathbb{F}_2[x_1, \dots, x_n]$.

Additional care must be taken to encode the idempotency law of boolean operations. In particular, for our purpose, the functions $h_n(x) = x_n$ and $h'_n(x) = x_n \cdot x_n$ are equal. However, this is not directly entailed by the definition of the polynomial ring. However, we can work over the ring $R' = R/(x_1^2 - x_1, \dots, x_n^2 - x_n)$ instead. By definition, in R', the terms h_i and h'_i are equal for all i. In particular, this allows to simply replace any term x_n^2 by the term x_n during calculations, which significantly simplifies the computations of the Gröbner basis.

In our implementation, we use the SageMath [16] computer algebra system to perform these computations.

4.3.2. Interpretation as System of Linear Equations. Next, we construct a system of polynomial equations H over $\mathbb{F}_2$, which satisfies the following two properties $\mathcal{P}1$ and $\mathcal{P}2$. If $h(x_1, \dots, x_n) = 0$, then $(x_1, \dots, x_n)$ fulfills *all* equations in H ($\mathcal{P}1$). If $h(x_1, \dots, x_n) = 1$, then *at least one* equation in H is not fulfilled by $(x_1, \dots, x_n)$ ($\mathcal{P}2$). This is achieved by translating each disjunct of the DNF to a polynomial $p_i \in \mathbb{F}_2[x_1, \dots, x_n]$, using the translation described in Section 4.3.1. Then, for all disjuncts, we add $p_i = 0$ to our set of equations.

Note that if $h(x_1, \dots, x_n) = 0$ for a formula in DNF, then *all* disjuncts must evaluate to 0, i.e., all equations are fulfilled. Contrary, if $h(x_1, \dots, x_n) = 1$, at least one polynomial equation is not fulfilled. For example, consider the following function h in DNF:

$$h(x) = (x_1 \wedge \neg x_2 \wedge x_3) \vee (x_2 \wedge \neg x_3) \vee (\neg x_1).$$

Its corresponding system of equations is given by

$$H = \left\{ \begin{array}{ll} x_1 \cdot (1-x_2) \cdot x_3 & = 0, \\ x_2 \cdot (1-x_3) & = 0, \\ 1 - x_1 & = 0 \end{array} \right\}.$$

4.3.3. Gröbner Bases. We use the property that a Gröbner basis usually is a more compact representation of a system of polynomial equations. To this end, we compute a Gröbner basis for the established system of polynomial equations H. Importantly, this transformation preserves the properties of our encoded function:

Theorem 1. *Over $\mathbb{F}_2$, a Gröbner basis transformation preserves the properties $\mathcal{P}1$ and $\mathcal{P}2$.*

A proof of this theorem can be found in Section E.

Our overall goal is to reduce the *size* of the formula. To this end, we expect our Gröbner basis to reduce the size of the system of polynomial equations. However, a Gröbner basis is always reduced with respect to a notion of *polynomial divisibility*. Note that this property does not necessarily align perfectly with the overall size of the system of equations. Still, a Gröbner basis is usually of small size. On all observed functions, the Gröbner base yields a significant reduction in terms of size. For example, a Gröbner basis of the previously established system of equations H is given by

$$\text{Gröbner}(H) = \left\{ \begin{array}{ll} 1 - x_1 & = 0, \\ x_2 + x_3 & = 0 \end{array} \right\}.$$

This output is more compact than the original H. In our implementation, we use the `Groebner` algorithm from the SINGULAR [15] computer algebra system in combination with the SageMath [16] computer algebra system.

4.3.4. Formula Reconstruction. In the last step, we reconstruct a boolean formula from the resulting Gröbner basis. This conversion from polynomials to boolean expressions is achieved by again using the correspondence from Table 2. Let φ denote this mapping. Then we build the following disjunction over all polynomials in our Gröbner basis:

$$h'(x_1, \dots, x_n) = \bigvee_{p \in \text{Gröbner}(H)} \varphi(p).$$

Note that if $h(x_1, \dots, x_n) = 0$, then $(x_1, \dots, x_n)$ fulfills all equations in H by $\mathcal{P}1$. Hence, by Theorem 1, all equations in $\text{Gröbner}(H)$ are fulfilled as well. Thus, $\varphi(p)(x_1, \dots, x_n) = 0$ for all equations $p \in \text{Gröbner}(H)$, which implies $h'(x_1, \dots, x_n) = 0$. Similarly, it follows from $\mathcal{P}2$ and Theorem 1 that if $h(x_1, \dots, x_n) = 1$, then $h'(x_1, \dots, x_n) = 1$. Overall, this entails $h(x_1, \dots, x_n) = h'(x_1, \dots, x_n)$. Hence, the recovered formula is equivalent to our original formula. In the above example, the resulting formula is

$$h'(x_1, x_2, x_3) = (\neg x_1) \vee (x_2 \oplus x_3).$$

Overall, the conversion of the Gröbner basis results in a formula of the following structure: The outermost connective is always a logical disjunction ($\vee$) over all polynomials in the Gröbner basis. Next, a polynomial is always a sum ($\oplus$) of monomials. A monomial itself is a product ($\wedge$) of variables. Hence, our resulting formula is always of the following shape:

$$\bigvee_{p \,\in\, \text{Gröbner}(H)} \quad \bigoplus_{\text{monomial } m \text{ in } p} \quad \bigwedge_{\text{variable } x \text{ in } m} x.$$

4.4. Recursive Solutions

The previous minimization pipeline creates an already simplified version of the formula. However, the overall structure is limited by the shape of the formula, as described

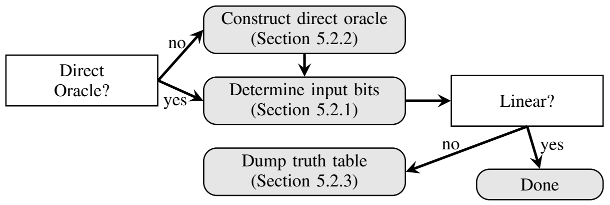

Direct Oracle?
no
yes
Construct direct oracle (Section 5.2.2)
Determine input bits (Section 5.2.1)
Linear?
no
yes
Dump truth table (Section 5.2.3)
Done

Figure 2: Overview of our unified measurement framework. First, an oracle, direct or indirect, is set up. Next, a linearity check is performed. If the function is non-linear, its truth table form is dumped. For linear functions, no further actions are needed.

above. To overcome this limitation, a *proper subterm* of the resulting formula can be further minimized by recursively applying the whole minimization pipeline. This starts with building a truth table for the formula induced by the subterm and ends with a reconstructed formula. If this formula is strictly smaller, the subterm can be replaced by it. This approach allows building even smaller formulas with more complex structures.

Choosing subterms for further simplification is implemented heuristically. Currently, the decision is influenced by the size and complexity of the respective subterm. Finding an optimal method here is subject to further work.

4.5. Runtime & Limits

The runtime of a non-recursive formula minimization is dominated on a case-by-case basis by either the ESPRESSO solver for DNF minimization or the computation of the Gröbner basis, depending on the structure of the minimized DNF. If ESPRESSO simplifies the formula significantly, the Gröbner basis can be computed fast, whereas for formulas with a complex minimized DNF, the Gröbner basis computation takes longer. For most evaluated functions, these steps take comparable time. Section 6.1 provides detailed measurements.

In general, this approach can reverse-engineer functions with up to 18–19 bits reliably within a few minutes. For more bits, a feasible runtime can only be achieved if the generating function is comparatively simple, as is the case for many microarchitectural hash functions. Still, runtime and memory grow at least exponentially in the number of input bits.

5. A Unified Measurement Framework

In this section, we propose a generic framework that allows gathering measurements to infer microarchitectural hash functions. As discussed in Section 3, inferring microarchitectural hash functions requires measurements. We present a two-step measurement strategy to infer microarchitectural hash functions reliably. Our framework is the only framework that correctly measures the cache slice if there are multiple slices per core, as is the case since the Intel Skylake microarchitecture. Moreover, it is the first framework that measures cache slices on AMD CPUs. Figure 2 shows an overview of our measurement framework.

5.1. High-level Overview

The framework supports *oracles* to add functionality for measuring a specific microarchitectural hash function. The framework supports two types of oracles, *direct* and *indirect* oracles. For direct oracles, the framework provides an input, e.g., an address, and the oracle returns the result of the hash function. Direct oracles can be used if the output is available, e.g., when using performance counters for cache slices [35], [36] or DRAM banks [24]. For indirect oracles, the framework provides two inputs, e.g., two addresses, and the oracle returns whether they result in the same hash. Indirect oracles can typically be constructed using timing side channels [41], [22], [21], [33]. We implement a direct oracle for cache slices using performance counters, indirect oracles for cache slices and DRAM banks using timing side channels, and an indirect oracle for the cache way μtag using performance counters.

5.2. Measurement

The measurement is a two-step process. First, the framework determines which bits are used by the microarchitectural hash function. Determining the bits that influence the output value significantly reduces the measurement and subsequently also the recovery time. Second, the framework iterates over all possible values of the relevant input bits and collects direct or indirect information regarding the output of the microarchitectural hash function.

5.2.1. Input Bits Determination. The first step of the measurement process is to perform the heuristic linearity check, as described in Section 4.1. The linearity check also determines which input bits influence the output of the hash function. This can be done by repeatedly picking a random input and mutating it on a single bit. In the case of physical or virtual addresses, this is done by flipping a single bit. We experimentally verify that 100 tries are sufficient to determine whether a bit is used as input for the microarchitectural hash functions considered in this paper. In case 100 tries are not sufficient, the number of tries can easily be increased.

5.2.2. Data Collection. The second step is to collect the output of the microarchitectural hash function. As the solver works using a truth table as input (cf. Section 4), the framework collects measurements for all possible assignments of the used input bits. The data collection depends on the oracle type.

Direct Oracle For a direct oracle, the framework can collect the output of the hash function directly, e.g., when using performance counters for measuring the cache slice of an address [35]. The framework simply collects the output of

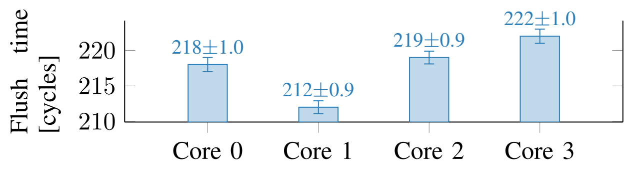


Figure 3: Time of flushing a cache line from different cores on Intel Core i7-8565U.

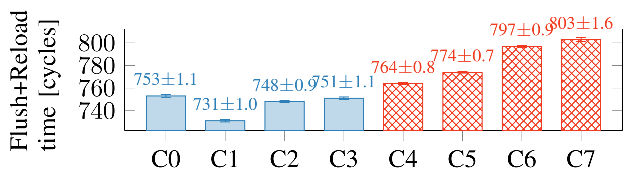


Figure 4: Time of a Flush+Reload cache conflict from different cores on AMD Ryzen 9 5900HX.

the hash function for every assignment of the relevant input bits.

Indirect Oracle For an indirect oracle, the framework performs multiple measurements to determine the number of output classes of the hash function. We leverage a generic algorithm to transform the indirect oracle into a direct oracle. We repeatedly measure different addresses until we find a witness for each output class. A new measurement can be compared with each witness by an indirect oracle. Therefore, we obtain a direct oracle with slightly higher complexity.

5.2.3. Noise Elimination. When noise is present, it needs to be accounted for by the measurement strategy. A practical and efficient strategy is to measure each input value multiple times and build a histogram over the measurements. If the most frequent bucket in the histogram reaches a predefined confidence threshold, we select this most frequently measured value. Our measurement framework uses this approach to quantify the noise present for each measurement and allows to add custom functionality to reduce the noise further when needed.

5.3. Measurement Oracles

This section introduces the proof-of-concept oracles we implement for evaluating the framework.

5.3.1. Cache Slices on Intel CPUs. Current measurement approaches for cache slices use performance counters or a timing side channel to measure the cache slice of an address (cf. Section 3.1). However, none of these approaches handles multiple slices per core, as has been the case since the Intel Skylake microarchitecture. Performance counters only count *all* slice accesses per core, and the timing does not differ for slices on the same core. Hence, to correctly infer a slice function that also considers multiple slices per core, we introduce a two-step measurement technique. First, our technique infers the CPU core associated with the slice. Second, it looks at the addresses mapping to one core in isolation, inferring a distinguishing function for mapping addresses on one core to the actual cache slice. These results are combined to the full cache-slice function.

Address to Core Mapping If available, we rely on performance counters [35], [36] for determining the core of the slice to which an address maps. In Table 4 (Section A), we provide the performance-counter configurations for microarchitectures we tested and on which they are available. For systems where the performance counters are unavailable, we use a timing side channel based on the observation of Gruss et al. [22]. The timing of the `clflush` instruction depends on the distance of the slice to the CPU core. Hence, for an address, we measure the time it takes to flush it from all CPU cores. For this, we pin the measurement application to a core and measure the time it takes to access and flush the address. By repeating this measurement for every core, we see the lowest timing if the measurement application runs on the core connected to the slice of the address. Figure 3 shows example timings for an address mapping to slice 1. The execution time increases the further away the core is from the slice of the address. Based on this direct oracle, we infer the function using the approach from Section 4.

Address to Slice Mapping In the second step, the measurement has to infer the actual slice of the core. We do not observe any timing difference for different slices on the same core. Hence, we use a contention-based approach to distinguish different slices on one core. For every cache set for which we have an address, we build a minimal eviction set using a timing side channel [58]. In this eviction set, all addresses map to the same cache slice, of which we know the respective CPU core. For every other address mapping to a slice on this core, we measure the time to access this address in addition to the addresses in the eviction set. If the address maps to the same slice, it conflicts with the eviction set, leading to cache misses. Otherwise, there is no conflict, resulting in only cache hits. We use this timing difference to split the addresses into two groups, each group mapping to a different slice. Note that this general approach is not limited to two slices per core. For more slices, one eviction set per set and slice can be created, and an address has to be tested against all these eviction sets. Based on this classification, we can again use the recovery approach from Section 4 to infer a microarchitectural hash function. Both functions can then be combined into one hash function, as the outputs are independent.

5.3.2. Cache Slices on AMD CPUs. For AMD CPUs, we demonstrate a novel technique to infer the cache slices, as there are no known approaches for detecting which slice an address maps to. In contrast to Intel CPUs, we are unaware of performance counters for slices on AMD. Moreover, the flush-based timing approach [22] that works on Intel CPUs does not provide conclusive results on AMD. Our approach relies on the timing differences of an L3 cache conflict, similar to the one exploited on Intel CPUs for the ZombieLoad attack [50]. Instead of measuring the time it takes to flush a cache line, as we do for Intel CPUs, we measure how long simultaneously-executing flush and load instructions to the same cache line take. Figure 4 shows these timings when accessing the same address from all 8 CPU cores (C0 to C7). In line with the timing side channel on Intel CPUs, the lowest timing can be observed if the address maps to the slice of the current core. Additionally, we observe a performance penalty if the address is in the slice of a different core complex (CCX). These timings are shown in red (crosshatched). We also verify this property using the CCX ID provided by the CPU via `cpuid` leaf 0x8000001e (bit 2 of `EBX` for 8-core CPUs). Hence, we can determine both the CCX and the slice within a CCX. As these functions are independent, we can solve both functions independently and combine them into the full hash function (cf. Section 6.2.1).

5.3.3. DRAM Addressing. To measure the DRAM-addressing function on a wide range of systems, our implementation relies on the timing side channel introduced by Pessl et al. [41]. However, in contrast to their approach, we introduce an optimized approach that relies on the surgical selection of physical addresses. Instead of randomly mapping memory and obtaining the physical addresses as done in previous work [41], we map the entire physical memory to a contiguous virtual memory region aligned to a multiple of the physical memory size. As a result, every physical address can be addressed by simply `or`-ing it with the start address of the virtual memory range. With this virtual memory range, we can directly choose accessible addresses that only differ in a specific bit. An additional optimization over previous work is marking this memory range as uncachable to ensure that every memory access goes directly to the DRAM. As we only have to read from memory, we do not have to pay attention to whether the accessed address is used by some other process or the operating system. However, as accessing memory-mapped devices could have unwanted side effects, such as system crashes, we exclude ranges that are not marked as system memory in `/proc/iomem`. The oracle itself is an indirect oracle, as it only returns whether two addresses map to the same bank (and different row), based on the execution time of reading from both addresses alternatingly.

5.3.4. Way Predictor. For the cache-way predictor, we use the approach described by Lipp et al. [33]. As virtual memory can be freely mapped, we can determine which address bits are relevant for the μtag computation by mapping addresses flipped in exactly one bit.

6. Evaluation

In this section, we evaluate our approach on different microarchitectural hash functions and demonstrate the correctness by means of case studies. Using our measurement framework (cf. Section 5) and the generic hash recovery (cf. Section 4), we also demonstrate results for previously unknown microarchitectural hash functions, such as the cache-slice function for the hybrid CPU architecture used in Intel Alder Lake CPUs, the cache-slice function of AMD Zen 3 CPUs, and the cache-way-predictor function for Zen 3(+).

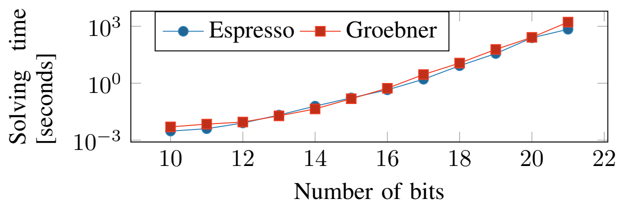


Figure 5: The number of input bits in relation to the time to solve for the ESPRESSO and Gröbner step on the data on an Intel Xeon E-2176M Coffe Lake processor.

6.1. Performance Evaluation

We evaluate the performance of our measurement framework and the solving approach regarding execution time. The execution time of the measurement framework is shown in Table 3 and depends on whether the function is linear or not. For the combination of linear functions and direct oracles, the framework does not need to dump the truth table. This can be seen for linear cache-slice functions, which are dumped within 1 min. For indirect oracles, the entire truth table needs to be dumped, which can be done in around 1 h for DRAM addressing functions and the AMD way predictor. For non-linear functions, the measurement time depends on the number of input bits. Additionally, the timing side channel for determining slices on AMD requires many measurements to obtain noise-free results, leading to a measurement of 23 h. Generally, as the measurement scales exponentially in the number of input bits, measurements for non-linear functions take up to several days.

The time for the logic-solving step depends on the time taken by the ESPRESSO logic minimizer step and the further minimization using Gröbner basis. The overall time taken for the logic-solving step is shown in Figure 5. Overall, the logic solving scales exponentially in both the Gröbner and ESPRESSO step. As stated in Section 4.5, these results indicate that ESPRESSO returns complex minimized DNF for the analyzed functions increasing the runtime in the Groebner step. Importantly, we also observe cache-slice functions such as the one for the Intel Core i9-12900K, where the Gröbner step is by a factor of 6 faster than the ESPRESSO step, indicating that the relation between the two runtimes strongly depends on the form of the analyzed function.

6.2. Hash Functions

To evaluate our approach, we apply it to 10 microarchitectural hash functions. These functions include 6 cache-slice functions, 2 cache-way predictor functions, and 2 DRAM-addressing functions. Out of these functions, 5 functions are linear, and 5 functions are non-linear. With this

TABLE 3: Measurement times for different hash functions.

CPU	Function	Relevant bits	Time
Intel i7-11800H	Cache slice	14	55 s
Intel Core i9-12900K	Cache slice	23	5 d
Intel Xeon E-2176M	Cache slice	22	23 h
AMD Ryzen 9 5900HX	Cache slice	3	23 h
AMD Ryzen 9 5900HX	Way predictor	16	55 min
AMD Ryzen 9 6900HX	Way predictor	16	117 min
Intel Core i5-2520M	DRAM address	8	18 h
Intel Core i3-7100T	DRAM address	8	16 h

$$\{h_0, h_1, h_2\} = \{b_8, \neg b_6 \wedge b_7 \wedge \neg b_8, b_6 \wedge \neg b_7 \wedge b_8\}$$

Figure 6: Cache slice function for the AMD Ryzen 9 5900HX.

diverse set of functions, we observe hash functions with input sizes from 3 to 23 bits and output sizes from 1 to 8 bits.

6.2.1. Cache Slices. In total, we analyze 6 cache-slice functions. These functions can be categorized into 3 previously unknown functions that we are the first to demonstrate, 2 previously known functions that we optimize into a more compact representation, and 1 previously known function that we reproduce.

New Functions We introduce 3 new cache-slice functions. These functions include a function for AMD Zen 3 and Zen 3+, and a function for the hybrid Intel Alder Lake microarchitecture. In contrast to most previous cache-slice functions, these functions are non-linear.

Figure 9 (Section B) illustrates the reverse-engineered cache-slice function for an Intel Core i9-12900K (Alder Lake) and Intel Xeon E-2176M (Coffee Lake). While the Intel Core i9-12900K has 16 cores (8 E- and 8 P-cores), it has only 10 slices, as each 4 E-cores share one slice. Hence, the resulting slice function is non-linear. Similarly, the Intel Xeon E-2176M has 6 cores and 12 slices resulting in a non-linear slice function, which differs from the 6 slice hash previously found by Yarom et al. [65]. We experimentally verify this slice function by comparing measurements of cache slices with the computed function. We sample 50 000 random addresses per function and compare the ground truth observed via performance counters to the function output. On both processors, the reverse-engineered slice function is 100 % correct. In addition, we experimentally verify the function of the Intel Xeon E-2176M in a Prime+Probe attack in Section 6.3. As the Intel Core i9-12900K has a non-inclusive L3, cross-core Prime+Probe is not directly possible. However, as the verification using performance counters is successful, we are highly confident that the function is correct.

Figure 6 illustrates the reverse-engineered function for an AMD Ryzen 9 5900HX (Zen 3). This CPU has 2 CCX with 4 cores each. As a result, the slice function has 3 outputs, where the least-significant bit, i.e., h_0 determines the CCX, and the other two bits determine the slice within the CCX. In contrast to the slice function on Intel CPUs, the slice is fully determined via low address bits, simplifying the eviction-set generation on AMD CPUs. For example, if 2 MB huge pages are available, an attacker can trivially determine both the cache set and the cache slice from the virtual address, as no bits above bit 20 are used.

In addition to the previously reverse-engineered slice functions for Intel Core and Xeon CPUs, where the number of cores is a power of two [35], we extended this function to 16 slices (from previously 8) and to newer Intel microarchitectures. Surprisingly, when we recover the slice function on an Intel Core i9-9980HK (Coffee Lake), it slightly differs from those recovered for older Intel Core and Intel Xeon CPUs [35]. This CPU has 8 cores, each with 2 hyperthreads, and thus a total of 16 slices. While the function for bit 0 and bit 1 are the same, the functions for bit 2 and bit 3 are previously unknown functions. For bit 2, we can rely on performance counters, while for bit 3, we need the indirect oracle based on a side channel. To demonstrate the correctness of our function, we evaluate it using a Prime+Probe attack in Section 6.3.

We show that while the performance-counter interface changed with Ice Lake (cf. Section A), the used function is still the same. We recover the function on an Intel Celeron CPU as well, specifically an Intel Celeron N4500 (Jasper Lake). Unsurprisingly, this function is also the same on Intel Celeron CPUs. However, on Intel Tiger Lake, which is based on a different microarchitecture, the cache-slice function is slightly different. For 4 slices, bit 0 is still generated by the same hash function as for previous generations [35], but bit 1 uses a previously unknown function that is also used for bit 3 on the i9-9980HK (cf. Figure 8 Section B).

Optimized Previous Functions We show that our approach has benefits even for previously (partially) inferred functions. The 2 non-linear functions found by McCalpin [36] can be minimized even further using our approach. We show this by finding more compact base sequences for these functions. All compact base sequences are provided in Section C.

6.2.2. Cache-Way Predictor. We recover the cache-way predictor function on AMD Zen 3 and Zen 3+, as it is currently only known up to Zen 2. Additionally, we verify that our approach produces the same result for the Zen and Zen 2 microarchitectures. The main differences between this hash function and other hash functions are that this one works on virtual instead of physical addresses and that it can only be observed indirectly. We experimentally verify that the reverse-engineered function is correct by reproducing the AES T-table attack from the original paper [33] in Section 6.4.

In line with the μtag hash function on Zen and Zen 2, the function consists of 8 linear components of the form shown in Figure 13 (Section D). The inferred functions are the same for Zen 3 and Zen 3+, and did not change from previous functions. We also reproduce the same function for Zen and Zen 2 as shown in the original paper [33].

6.2.3. DRAM Addressing. We recover the DRAM-addressing function on a system with DDR3 and a system with DDR4 to show that our optimized oracle works. For the DDR3 system, we use an Intel Core i5-2520M (Sandy Bridge), the closest to the system used by Pessl et al. [41] (Intel Core i5-2540M). We also use the same DRAM configuration as described in that paper, with one channel and one DIMM. Our framework recovers the same function within 18 h. For the DDR4 systems, we use an Intel Core i3-7100T. The recovered function matches the Coffee Lake function of Wang et al. [59]. We reverse-engineer this function within 16 h.

6.3. Case Study: Prime+Probe on Intel

To show that the inferred functions are correct, we perform a Prime+Probe attack on the LLC of the Intel Xeon E-2176M and the Intel Core i9-9980HK, for which no slice functions were previously known. Prime+Probe allows to monitor cross core cache accesses on the shared LLC without the requirement for shared memory. A common evaluation scenario for Prime+Probe is attacking the AES T-Table implementation of OpenSSL. To enable comparison with previous work, we attack OpenSSL 1.0.1e, as it is widely used as a benchmark for side-channel attacks [43], [19], [22], [32]. However, we note that the T-Table implementation did not change significantly in newer versions, and we also verify that the attack still works on the current version of OpenSSL 3.0.7. We base our attack on the implementation of Gruss et al. [22] but reduce the number of required encryptions from 1 000 000 to 1000 and use our reverse-engineered slice function.

On the i9-9980HK, out of the 1000 times we execute the attack, we recover, on average, 99.98 % of the key correctly. Every attack takes, on average, 2.4 s. On the Xeon E-2176M, we recover on average 97.65 % of the key correctly. Every attack takes 4.6 s on average. The brute-force complexity for finding the non-recovered bits is well within reach for even the most limited attacker, as the required number of encryptions can be performed in less than a second on commodity hardware, for example by using AES with hardware support [2].

6.4. Case Study: Take a Way on AMD

To verify the correctness of the inferred μtag hash function of the AMD way predictor, we reproduce the AES T-table attack from the original paper [33] on both the Zen 3 and Zen 3+ architectures. To evaluate our attacks, we use an AMD Ryzen 9 5900HX for the Zen 3 and an AMD Ryzen 9 6900HX for the Zen 3+ architecture. Figure 7 shows the result of the Load+Reload and Collide+Probe attacks on the first AES key byte. As the diagonal is visible, we can successfully extract the value of the key byte, i.e., 0 in this case. In line with the attack on older architectures [33], we successfully recover the AES key with Load+Reload and Collide+Probe on the Zen 3 and Zen 3+ CPUs.

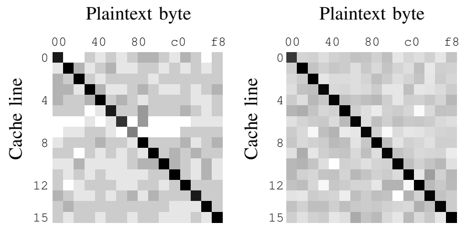


Figure 7: AES T-table cache access patterns (Ryzen 9 5900HX) with Load+Reload (left) and Collide+Probe (right).

7. Alternative Hash Functions

In this section, we propose alternatives to currently used microarchitectural hash functions that are more resistant to reverse-engineering. While knowing a microarchitectural hash function is generally not a strict necessity for mounting an attack, it can significantly simplify an attack. Without knowledge of the function, attackers have to resort to either less-effective attacks [52], [41], [22] or additional side channels [58]. Hence, we advocate the adoption of functions that do not impede the system's performance and are still resistant to currently-known reverse-engineering approaches. We suggest different approaches for such functions with different advantages and disadvantages.

Larger Input Space For our approach, the reverse-engineering time increases exponentially with the number of input bits of the hash function. As indicated by both our theoretical analysis in Section 4.5 as well as our practical benchmarking in Section 6.1, reverse engineering hash functions on a commodity computer reaches its limits at 21–25 input bits. Hence, increasing the number of input bits trivially hardens the functions against reverse engineering. However, while hash functions can be made more resistant to reverse engineering via our proposed methods by using a bigger input space, it only hardens these functions against our approach. Future methods could find ways to efficiently reverse engineer such hardened hash functions, e.g., by finding ways to parallelize the solving. While this could be a short-time solution, it likely does not provide long-term security.

Cryptographic Hashes From a security perspective, cryptographic primitives are an ideal candidate for replacing microarchitectural hash functions. Using cryptographic constructions, such as those proposed for secure cache designs [62], [48], [46], ensures that it is infeasible for an attacker to model and solve the connection between input and output. However, there are not many readily-usable low-latency cryptographic primitives, as this is still an open research problem.

Keyed Hashes We propose keyed hash functions that keep the load-balancing properties while offering an additional layer of randomization. Keyed hash functions allow integrating additional randomness to the hash function to make reverse-engineering more difficult, effectively resulting in a new hash function on every boot. Random bits can be generated by the CPU on system boot, e.g., using the `rdrand` instruction already present on x86 CPUs. In addition, this random value can be changed at will during runtime, at the cost of invalidating the targeted microarchitectural buffer. This requires an attacker to infer the hash function every time the randomness bits in the function are changed. The performance impact and implementation complexity of this mitigation depends on the underlying microarchitecture, and an evaluation of such an approach is subject to future work.

8. Discussion

Applicability and Limitations Our approach is designed for microarchitectural hash functions used in state-of-the-art microarchitectures. In contrast to cryptographic hash functions, these functions do not claim any security properties. Instead, they are designed for load balancing and improving the system's performance. Additionally, in contrast to cryptographic hash functions, the functions are secret, whereas the inputs and sometimes outputs can be observed. Hence, cryptographic hash functions are out of scope for our paper. This also includes cryptographic hash functions proposed for the use in the microarchitecture, e.g., for pointer authentication [44] or secure cache designs [46], [62], [48].

Use of the Hash Functions In line with previous work, we identify several use cases for the reverse-engineered microarchitectural hash functions. These functions can simplify microarchitectural attacks [51], [41], [33], especially for privileged attackers against, e.g., trusted-execution environments [49]. They can also be used to build more effective defenses. The operating system could use knowledge of the slice function to reduce interference on the interconnect, reducing the observable leakage from side-channel attacks on the ring bus [39] or mesh [14] connecting cache slices. For cache coloring [66], [53], knowing the slice function results in more available colors and, thus, more isolation domains [65]. Cache coloring is not only helpful in protecting against side-channel attacks but can also be used to improve the performance of applications by isolating them [61], [31]. McCalpin [36] also shows that knowing the cache-slice function can reduce hash conflicts and, as a result, improve the performance of specific applications up to 28 %. Finally, these functions can be implemented in CPUs and system simulators, such as gem5 [5], [56], to model existing CPUs even more accurately.

9. Conclusion

This paper introduced a generic approach for automatically reverse-engineering microarchitectural hash functions using a combination of techniques originally used for logic-gate minimization and in computer algebra. We inferred the hash functions based on input-output pairs observed via side channels, resulting in 3 previously-unknown non-linear hash functions on AMD and Intel CPUs, including the new Alder Lake hybrid-CPU architecture. We verified the correctness of the functions by showing the equivalence to some known hash functions and mounting successful side-channel attacks that rely on these functions. We discussed the need to design such functions with performance and security in mind to make microarchitectural attacks more difficult in future CPUs.

References

[1] 7-cpu. AMD Zen, 2019. URL: https://www.7-cpu.com/cpu/Zen.html.

[2] Kahraman D. Akdemir, Martin G. Dixon, Wajdi K. Feghali, and et al. Patrick G Fay. Breakthrough AES Performance with Intel® AES New Instructions, 2010. URL: https://www.intel.com/content/dam/develop/external/us/en/documents/10tb24-breakthrough-aes-performance-with-intel-aes-new-instructions-final-secure.pdf.

[3] Thomas Becker and Volker Weispfenning. *Gröbner bases: a computational approach to commutative algebra*. Springer Science & Business Media, 2012.

[4] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In *CHES*, 2016.

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 2011.

[6] R.K. Brayton, G.D. Hachtel, and A.L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 1990.

[7] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang. Mis: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1987.

[8] Robert K Brayton, Gary D Hachtel, Curt McMullen, and Alberto Sangiovanni-Vincentelli. *Logic minimization algorithms for VLSI synthesis*. Springer Science & Business Media, 1984.

[9] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *USENIX Security Symposium*, 2020.

[10] Bruno Buchberger. Ein Algorithmus zum Auffinden der Basiselemente des Restklassenrings nach einem nulldimensionalen Polynomideal. *Universitat Innsbruck, Austria, Ph. D. Thesis*, 1965.

[11] David Buchfuhrer and Christopher Umans. The Complexity of Boolean Formula Minimization. Lecture Notes in Computer Science, 2008.

[12] Christopher Condrat and Priyank Kalla. A gröbner basis approach to cnf-formulae preprocessing. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2007.

[13] David Cox, John Little, and Donal OShea. *Ideals, varieties, and algorithms: an introduction to computational algebraic geometry and commutative algebra*. Springer Science & Business Media, 2013.

[14] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. Don't mesh around: Side-Channel attacks and mitigations on mesh interconnects. In *USENIX Security Symposium*, 2022.

[15] Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann. SINGULAR 4-3-0 — A computer algebra system for polynomial computations, 2022.

[16] The Sage Developers, William Stein, David Joyner, David Kohel, John Cremona, and Burçin Eröcal. SageMath, version 9.7, 2022. URL: http://www.sagemath.org.

[17] Catherine Easdon, Michael Schwarz, Martin Schwarzl, and Daniel Gruss. Rapid Prototyping for Microarchitectural Attacks. In *USENIX Security*, 2022.

[18] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P*, 2020.

[19] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering*, 2016.

[20] Lukas Gerlach, Fabian Thomas, Robert Pietsch, and Michael Schwarz. A Large-Scale Rowhammer Reproduction Study Using the Blacksmith Fuzzer. In *ESORICS*, 2023.

[21] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, 2018.

[22] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[23] Martin Heckel and Florian Adamsky. Reverse-engineering bank addressing functions on amd cpus. In *DRAMSec*, 2023.

[24] Christian Helm, Soramichi Akiyama, and Kenjiro Taura. Reliable reverse engineering of intel dram addressing using performance counters. In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020.

[25] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*, 2013.

[26] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. *Cryptology ePrint Archive, Report 2015/898*, 2015.

[27] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *Euromicro Conference on Digital System Design*, 2015.

[28] Kevin Karplus. Using If-Then-Else DAGs for Multi-Level Logic Minimization. 1989.

[29] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[30] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs. In *EuroS&P*, 2020.

[31] Haifeng Li, Tianyue Lu, Yuhang Liu, and Mingyu Chen. Make page coloring more efficient on slice-based three-level cache. In *ICPADS*, 2019.

[32] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[33] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *AsiaCCS*, 2020.

[34] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, 2015.

[35] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Complex Addressing Using Performance Counters. In *RAID*, 2015.

[36] John D McCalpin. Mapping addresses to l3/cha slices in intel processors. Technical report, 2021.

[37] Edward J McCluskey. Minimization of boolean functions. *The Bell System Technical Journal*, 35(6), 1956.

[38] Thanh Hung Nguyen. *Combinations of Boolean Groebner Bases and SAT Solvers*. PhD thesis, Technische Universität Kaiserslautern, 2014.

[39] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. Lord of the Ring (s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *USENIX Security Symposium*, 2021.

[40] Colin Percival. Cache Missing for Fun and Profit. In *BSDCan*, 2005.

[41] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*, 2016.

[42] Klaus Pommerening. Lecture notes in cryptology, 1999. URL: https://www.staff.uni-mainz.de/pommeren/Cryptology/Bitblock/Fourier/ANF.pdf.

[43] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *CCS*, 2021.

[44] Qualcomm Technologies. Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions, 2017. URL: https://www.qualcomm.com/media/documents/files/whitepaper-pointerauthentication-on-armv8-3.pdf.

[45] Willard V Quine. The problem of simplifying truth functions. *The American mathematical monthly*, 59(8), 1952.

[46] Moinuddin K Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *IEEE MICRO*, 2018.

[47] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security Symposium*, 2016.

[48] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design. In *USENIX Security Symposium*, 2021.

[49] Michael Schwarz and Daniel Gruss. How Trusted Execution Environments Fuel Research on Microarchitectural Attacks. *IEEE Security & Privacy*, 2020.

[50] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[51] Mark Seaborn. Exploiting the DRAM rowhammer bug to gain kernel privileges, March 2015. Retrieved on June 26, 2015. URL: http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[52] Mark Seaborn. L3 cache mapping on Sandy Bridge CPUs, April 2015. Retrieved on June 26, 2015. URL: http://lackingrhoticity.blogspot.com/2015/04/l3-cache-mapping-on-sandy-bridge-cpus.html.

[53] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *DSN-W*, 2011.

[54] Anton Shilov. SK Hynix Shows Off 48GB and 96GB DDR5-6400 Memory Modules, 2022. URL: https://www.tomshardware.com/news/sk-hynix-shows-off-48gb-and-96gb-ddr5-6400-memory-modules.

[55] Mathias Soeken, Winston Haaswijk, Eleonora Testa, Alan Mishchenko, Luca Gaetano Amarù, Robert K. Brayton, and Giovanni De Micheli. Practical exact synthesis. In *DATE*, 2018.

[56] Fabian Thomas, Lukas Gerlach, and Michael Schwarz. Hammulator: Simulate Now – Exploit Later. In *DRAMSec*, 2023.

[57] Christopher Umans. The Minimum Equivalent DNF Problem and Shortest Implicants. *Journal of Computer and System Sciences*, 2001.

[58] Pepe Vila, Boris Köpf, and Jose Morales. Theory and Practice of Finding Eviction Sets. In *S&P*, 2019.

[59] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. DRAMDig: A knowledge-assisted tool to uncover DRAM address mapping. In *DAC*, 2020.

[60] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*, 2017.

[61] Xiaodong Wang, Shuang Chen, Jeff Setter, and José F Martínez. Swap: Effective fine-grain management of shared last-level caches with minimum hardware support. In *HPCA*, 2017.

[62] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium*, 2019.

[63] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *USENIX Security Symposium*, 2016.

[64] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *S&P*, 2019.

[65] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel Last-Level Cache. *Cryptology ePrint Archive, Report 2015/905*, 2015.

[66] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. Coloris: a dynamic cache partitioning system using page coloring. In *PACT*, 2014.

Appendix A. Cache Slice Performance Counter

As the performance counters used to infer cache slices change over microarchitectures, we provide a table containing all necessary MSR configurations for various microarchitectures. Table 4 lists these performance counters and configuration values. These values can be used in, e.g., the code of Maurice et al. [35].

Appendix B. Cache-slice Functions

This section contains the full equations for all newly inferred cache slice functions.

Figure 8 shows the cache slice function for an Intel Core i9-9980HK with 16 slices. Figure 9 shows the cache slice mapping for an Intel Alder Lake CPU with 16 cores and 10 slices. Listing 1 provides source code to compute the cache slice functions on both processors.

Appendix C. McCalpin's Cache Slice Mappings [36]

We show a minimized version of McCalpin's cache slice mappings [36]. Figures 10 to 12 show the minimized base sequence for a 16-, 20-, and 24-slice Intel Skylake CPU, respectively.

Appendix D. μtag hash for Zen 3 and Zen 3+

Figure 13 shows the μtag hash function of Zen 3 and Zen 3+ CPUs. The inferred function is identical to the previously reverse-engineered functions for older AMD microarchitectures.

Appendix E. Proof of Theorem 1

In the following, we provide a proof of Theorem 1 from Section 4.3.3. However, we first have to introduce ideals before we can prove the theorem. To simplify the notation, all polynomials are over $\mathbb{F}_2$, and we identify a polynomial p with the corresponding equation $p(x) = 0$.

Definition 1. *Let $H = \{p_1, \dots, p_k\}$ be a set of equations in a polynomial ring R. Then the ideal generated by H is*

$$\langle H \rangle := \left\{ \sum_{i=1}^{k} q_i p_i \ : \ q_1, \dots, q_k \in R \right\}.$$

The following proposition contains an elementary property of ideals, which states that a point $(x_1, \dots, x_n)$ satisfies all equations in H if and only if it satisfies all equations in the ideal $\langle H \rangle$. For convenience, we also include a proof of this statement.

Proposition 1. *Let $H = \{p_1, \dots, p_k\}$ be a set of equations in a polynomial ring R and let $x \in \mathbb{F}_2^n$. Then*

$$\forall p \in H \colon p(x) = 0 \quad \iff \quad \forall p \in \langle H \rangle \colon p(x) = 0.$$

Proof. First, assume $p_i(x) = 0$ for all $p_i \in H$ and let $p' \in \langle H \rangle$. By definition, p' has the form

$$p' = \sum_{i=1}^{k} q_i p_i$$

for some $q_1, \dots, q_k \in \mathbb{F}_2[x_1, \dots, x_n]$ and we obtain

$$p'(x) = \sum_{i=1}^{n} q_i(x) p_i(x) = \sum_{i=1}^{n} q_i \cdot 0 = 0.$$

Conversely, assume $p'(x) = 0$ for all $p' \in \langle H \rangle$ and let $p_i \in H$. Define

$$q_j := \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise} \end{cases}$$

and

$$p' := \sum_{j=1}^{k} q_j p_j.$$

Then $p_i = p'$ and $p_i(x) = 0$, since $p' \in \langle H \rangle$.

$\square$

Now, we can prove Theorem 1. Recall that we are given a function h and a system of polynomial equations H, which satisifes the properties:

- $\mathcal{P}1$: $h(x_1, \dots, x_n) = 0 \implies \forall p \in H \colon p(x_1, \dots, x_n) = 0$,
- $\mathcal{P}2$: $h(x_1, \dots, x_n) = 1 \implies \exists p \in H \colon p(x_1, \dots, x_n) \neq 0$.

We have to show that the properties $\mathcal{P}1$ and $\mathcal{P}2$ are preserved if we replace H by $\text{Gröbner}(H)$. Observe that both

TABLE 4: MSRs and their configuration values for inferring cache slices on Intel CPUs as shown by Maurice et al. [35].

Variable Name	**< Skylake**	**>= Skylake**	**>= Ice Lake**	**>= Alder Lake**
Counter MSR	0x706	0x706	0x702	0x2002
Counter MSR spacing	0x10	0x10	0x8	0x8
Control MSR	0x700	0x700	0x700	0x2000
Control MSR spacing	0x10	0x10	0x8	0x8
Control MSR value	0x408f34	0x408f34	0x408834	0x408834
Global Counter MSR	0x391	0xe01	0xe01	0x2ff0
Enable-counter mask	0x2000000f	0x20000000	0x20000000	0x20000000

$$h_0 = b_6 \oplus b_{10} \oplus b_{12} \oplus b_{14} \oplus b_{16} \oplus b_{17} \oplus b_{18} \oplus b_{20} \oplus b_{22} \oplus b_{24} \oplus b_{25} \oplus b_{26} \oplus b_{27} \oplus b_{28} \oplus b_{30} \oplus b_{32} \oplus b_{33} \oplus b_{35} \oplus b_{36}$$
$$h_1 = b_7 \oplus b_{11} \oplus b_{13} \oplus b_{15} \oplus b_{17} \oplus b_{19} \oplus b_{20} \oplus b_{21} \oplus b_{22} \oplus b_{23} \oplus b_{24} \oplus b_{26} \oplus b_{28} \oplus b_{29} \oplus b_{31} \oplus b_{33} \oplus b_{34} \oplus b_{35} \oplus b_{37}$$
$$h_2 = b_{10} \oplus b_{11} \oplus b_{13} \oplus b_{16} \oplus b_{17} \oplus b_{18} \oplus b_{19} \oplus b_{20} \oplus b_{21} \oplus b_{22} \oplus b_{27} \oplus b_{28} \oplus b_{30} \oplus b_{31} \oplus b_{32} \oplus b_{33}$$
$$h_3 = b_9 \oplus b_{12} \oplus b_{16} \oplus b_{17} \oplus b_{19} \oplus b_{21} \oplus b_{22} \oplus b_{23} \oplus b_{25} \oplus b_{26} \oplus b_{27} \oplus b_{29} \oplus b_{31} \oplus b_{32} \oplus b_{33} \oplus b_{34} \oplus b_{35}$$

Figure 8: Cache slice function for Intel Core i9-9980HK with 16 slices.

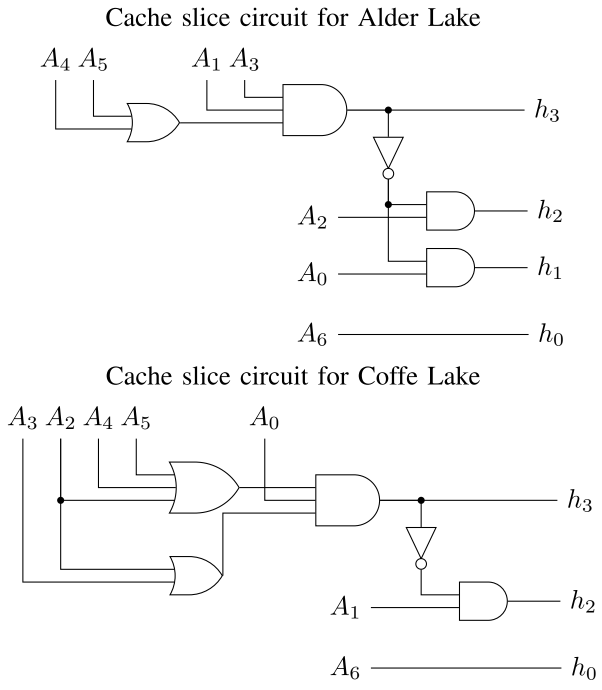
```
#include <stdint.h>
#define _B(x, pos) (((x) >> (pos)) & 1)

int compute_slice(uint64_t x) {
  uint64_t A_0 = _B(x, 6)  ^ _B(x, 11) ^ _B(x, 12) ^ _B(x, 16) ^ _B(x, 18) ^ _B(x, 21) ^
                 _B(x, 22) ^ _B(x, 23) ^ _B(x, 24) ^ _B(x, 26) ^ _B(x, 30) ^ _B(x, 31);
  uint64_t A_1 = _B(x, 7)  ^ _B(x, 12) ^ _B(x, 13) ^ _B(x, 17) ^ _B(x, 19) ^ _B(x, 22) ^
                 _B(x, 23) ^ _B(x, 24) ^ _B(x, 25) ^ _B(x, 27) ^ _B(x, 31);
  uint64_t A_2 = _B(x, 8)  ^ _B(x, 13) ^ _B(x, 14) ^ _B(x, 18) ^ _B(x, 20) ^ _B(x, 23) ^
                 _B(x, 24) ^ _B(x, 25) ^ _B(x, 26) ^ _B(x, 28);
  uint64_t A_3 = _B(x, 9)  ^ _B(x, 14) ^ _B(x, 15) ^ _B(x, 19) ^ _B(x, 21) ^ _B(x, 24) ^
                 _B(x, 25) ^ _B(x, 26) ^ _B(x, 27) ^ _B(x, 29);
  uint64_t A_4 = _B(x, 10) ^ _B(x, 15) ^ _B(x, 16) ^ _B(x, 20) ^ _B(x, 22) ^ _B(x, 25) ^
                 _B(x, 26) ^ _B(x, 27) ^ _B(x, 28) ^ _B(x, 30);
  uint64_t A_5 = _B(x, 11) ^ _B(x, 16) ^ _B(x, 17) ^ _B(x, 21) ^ _B(x, 23) ^ _B(x, 26) ^
                 _B(x, 27) ^ _B(x, 28) ^ _B(x, 29) ^ _B(x, 31);

  #ifdef COFFEE_LAKE
  uint64_t val2 = (((A_2) | (A_4) | (A_5)) & ((A_2) | (A_3)) & (A_0)) & 1;
  uint64_t val1 = (~val2 & A_1) & 1;
  uint64_t val0 = _B(x, 6)  ^ _B(x, 8)  ^ _B(x, 9)  ^ _B(x, 10) ^ _B(x, 14) ^ _B(x, 15) ^
                  _B(x, 17) ^ _B(x, 18) ^ _B(x, 20) ^ _B(x, 23) ^ _B(x, 27) ^ _B(x, 30) ^ _B(x,31);
  return val0 | (val1 << 1) | (val2 << 2);
  #elif defined(ALDER_LAKE)
  uint64_t val3 = ((A_4 | A_5) & (A_1 & A_3)) & 1;
  uint64_t val2 = (~val3 & A_0) & 1;
  uint64_t val1 = (~val3 & A_2) & 1;
  uint64_t val0 = _B(x, 6)  ^ _B(x, 8)  ^ _B(x, 9)  ^ _B(x, 10) ^ _B(x, 14) ^ _B(x, 15) ^
                  _B(x, 17) ^ _B(x, 18) ^ _B(x, 20) ^ _B(x, 23) ^ _B(x, 27) ^ _B(x, 30) ^ _B(x,31);
  return val0 | (val1 << 1) | (val2 << 2) | (val3 << 3);
  #endif
}


```

Listing 1: Code to compute the cache slice functions for the analyzed Alder Lake and Coffee Lake processors



Common slice circuit input

$$
\begin{aligned}
A_0 &= x_6 \oplus x_{11} \oplus x_{12} \oplus x_{16} \oplus x_{18} \oplus x_{21} \oplus x_{22} \oplus x_{23} \oplus x_{24} \oplus x_{26} \\
&\quad \oplus x_{30} \oplus x_{31} \\
A_1 &= x_7 \oplus x_{12} \oplus x_{13} \oplus x_{17} \oplus x_{19} \oplus x_{22} \oplus x_{23} \oplus x_{24} \oplus x_{25} \oplus x_{27} \\
&\quad \oplus x_{31} \\
A_2 &= x_8 \oplus x_{13} \oplus x_{14} \oplus x_{18} \oplus x_{20} \oplus x_{23} \oplus x_{24} \oplus x_{25} \oplus x_{26} \oplus x_{28} \\
A_3 &= x_9 \oplus x_{14} \oplus x_{15} \oplus x_{19} \oplus x_{21} \oplus x_{24} \oplus x_{25} \oplus x_{26} \oplus x_{27} \oplus x_{29} \\
A_4 &= x_{10} \oplus x_{15} \oplus x_{16} \oplus x_{20} \oplus x_{22} \oplus x_{25} \oplus x_{26} \oplus x_{27} \oplus x_{28} \oplus x_{30} \\
A_5 &= x_{11} \oplus x_{16} \oplus x_{17} \oplus x_{21} \oplus x_{23} \oplus x_{26} \oplus x_{27} \oplus x_{28} \oplus x_{29} \oplus x_{31} \\
A_6 &= x_6 \oplus x_8 \oplus x_9 \oplus x_{10} \oplus x_{14} \oplus x_{15} \oplus x_{17} \oplus x_{18} \oplus x_{20} \oplus x_{23} \\
&\quad \oplus x_{27} \oplus x_{30} \oplus x_{31}
\end{aligned}
$$

Figure 9: Cache slice functions for Alder Lake and Coffee Lake

$$
\begin{aligned}
h_0 &= x_0 \oplus x_2 \oplus x_3 \oplus x_4 \\
h_1 &= x_0 \oplus x_1 \oplus x_2 \oplus x_6 \\
h_2 &= (\neg x_5 \land \neg x_4 \land (x_2 \oplus x_7)) \\
&\quad \lor (\neg(x_1 \oplus x_6 \oplus x_7) \land (x_2 \oplus x_7)) \\
&\quad \lor (\neg x_3 \land (x_2 \oplus x_7)) \\
h_3 &= (\neg x_4 \lor \neg x_3 \lor \neg(x_1 \oplus x_6 \oplus x_7)) \\
&\quad \land (\neg x_5 \lor \neg x_3 \lor \neg(x_1 \oplus x_6 \oplus x_7)) \\
&\quad \land (x_0 \oplus x_5 \oplus x_6) \\
h_4 &= (x_5 \lor x_4) \land (x_1 \oplus x_6 \oplus x_7) \land x_3
\end{aligned}
$$

Figure 11: Function generating the base sequence for a 20-core Skylake CPU from McCalpin [36]

$$
\begin{aligned}
h_0 &= x_0 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_8 \\
h_1 &= x_0 \oplus x_1 \oplus x_2 \oplus x_6 \\
h_2 &= x_2 \oplus x_3 \oplus x_6 \\
h_3 &= (\neg x_4 \lor \neg(x_3 \oplus x_8) \lor \neg(x_0 \oplus x_5 \oplus x_6)) \\
&\quad \land (\neg x_5 \lor \neg(x_3 \oplus x_8) \lor (x_0 \oplus x_6)) \\
&\quad \land (\neg(x_2 \oplus x_7 \oplus x_8) \lor \neg(x_0 \oplus x_5 \oplus x_6)) \\
&\quad \land (x_1 \oplus x_6 \oplus x_7) \\
h_4 &= (x_5 \lor x_4 \lor (x_2 \oplus x_7 \oplus x_8)) \\
&\quad \land ((x_2 \oplus x_7 \oplus x_8) \lor (x_3 \oplus x_8)) \\
&\quad \land (x_0 \oplus x_5 \oplus x_6)
\end{aligned}
$$

Figure 12: Function generating the base sequence for a 24-core Skylake CPU from McCalpin [36]

$$
\begin{aligned}
h_0 &= x_6 \oplus x_{10} \oplus x_{12} \oplus x_{14} \oplus x_{16} \oplus x_{17} \oplus x_{18} \oplus 1 \\
h_1 &= x_7 \oplus x_{11} \oplus x_{13} \oplus x_{15} \oplus x_{17} \oplus x_{19} \oplus 1 \\
h_2 &= x_8 \oplus x_{12} \oplus x_{13} \oplus x_{16} \oplus x_{19} \oplus 1 \\
h_3 &= x_9 \oplus x_{12} \oplus x_{16} \oplus x_{17} \oplus x_{19} \oplus 1
\end{aligned}
$$

Figure 10: The partial 16-slice addressing function for a Xeon Skylake processor from McCalpin [36]. In the function provided by McCalpin, bits above bit 19 are missing.

$$
\begin{aligned}
h_0 &= b_{12} \oplus b_{27} \\
h_1 &= b_{13} \oplus b_{26} \\
h_2 &= b_{14} \oplus b_{25} \\
h_3 &= b_{15} \oplus b_{20} \\
h_4 &= b_{16} \oplus b_{21} \\
h_5 &= b_{17} \oplus b_{22} \\
h_6 &= b_{18} \oplus b_{23} \\
h_7 &= b_{19} \oplus b_{24}
\end{aligned}
$$

Figure 13: μtag hash function for Zen 3 and Zen 3+

properties can be combined into

$$h(x_1, \dots, x_n) = 0 \quad \iff \quad \forall p \in H \colon p(x_1, \dots, x_n) = 0,$$

because we are working over $\mathbb{F}_2$. By applying Proposition 1, we can replace H with $\langle H \rangle$ in the previous statement. A key property of Gröbner bases is that they generate the same ideal (see Cox et al. [13]), i.e., we have

$$\langle H \rangle = \langle \text{Gröbner}(H) \rangle.$$

Thus,

$$h(x_1, \dots, x_n) = 0 \iff \forall p \in \langle \text{Gröbner}(H) \rangle \colon p(x_1, \dots, x_n) = 0$$

By applying Proposition 1 again, we obtain

$$h(x_1, \dots, x_n) = 0 \iff \forall p \in \text{Gröbner}(H) \colon p(x_1, \dots, x_n) = 0$$

which is equivalent to $\mathcal{P}1$ and $\mathcal{P}2$ for $\text{Gröbner}(H)$.

Appendix F. Meta-Review

F.1. Summary

The paper considers the task of reverse engineering hash functions used to load balance microarchitectural structures, such as LLC cache slices, and DRAM addressing functions. As knowing the details of these hash functions is often needed for mounting side channel attacks, the authors show how to reverse engineer these functions using automated tools. Finally, the authors demonstrate their approach on several previously unknown hash functions, across different data structures on both AMD and Intel CPUs.

F.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field
- Establishes a New Research Direction

F.3. Reasons for Acceptance

- Being one of the first tools to automatically reverse engineer hash functions used in microarchitectural structures, the paper provides a novel tool to enable advances in side channel research. In particular, this has the potential to simplify various ad-hoc manual approaches used to recover such structures.
- Next, the framework presented in the paper is generic and works across multiple architectures, functions, and oracles.
- Finally, from a technical perspective, the papers techniques are clever. Resulting in being able to handle processors where the number of slices is not a power of two, as well as minimizing the extracted hash functions.

F.4. Noteworthy Concerns

The performance of the proposed framework is rather slow, requiring days in order to recover the targeted function. However, as the measurement needs to only be done once per processor, this is not considered to be a major limitation.