Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

Bachelor's Thesis

Reversing the Microarchitecture with Unikernels

submitted by

Mikka Rainer

on October 04, 2023

Reviewers

Dr. Michael Schwarz

Prof. Dr. Jan Reineke

Advisor

Lukas Gerlach

Supervisor

Dr. Michael Schwarz

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Saarbrücken, October 04, 2023,

(Mikka Rainer)

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, October 04, 2023,

(Mikka Rainer)

Abstract

The microarchitecture of modern CPUs is largely undocumented. However, knowledge of inner CPU mechanisms allows for finding novel attack vectors, creating new defenses, and building high-performance applications. While there is an ongoing effort to reverse engineer the inner mechanisms of modern processors, researchers are largely unable to observe individual microarchitectural events.

In this thesis, we investigate how we can create a noise-free measurement environment for microarchitectural reverse engineering by leveraging the power of unikernels. In a case study, we show that we can significantly improve the accuracy of address-to-slice mappings in comparison to previous techniques, taking the example of the addressing function of last-level cache slices. Contrary to previous work, we can measure microarchitectural events up to a single instruction granularity. This enables us to speed up reverse engineering of last-level cache slices by a factor of 260. We further reverse engineer one known and one previously unknown slice-addressing function. In this work, we make the first step towards a unified framework for microarchitectural reverse engineering by proposing a specialized research kernel.

Acknowledgements

I have been fortunate to receive great support during the development of this bachelor's thesis. First and foremost, I would like to thank my supervisor, Michael Schwarz, and my advisor, Lukas Gerlach, for their efforts. You introduced me to the fascinating world of microarchitectures, the most inner mechanisms of processors. You always took time to answer all of my questions and came up with new ideas when I was stuck.

Moreover, I would like to thank the other members of the Rootsec group. You were always happy to help, and it was a pleasure working with you. I also want to thank Robert Pietsch, who worked with me on the first prototype of the research kernel. It was very nice to tackle this project with you. You are an incredibly talented researcher.

Finally, I would like to acknowledge the support of my family and friends. Special thanks go to my girlfriend, Katharina, for her ongoing support throughout this special time. You are always there to motivate and encourage me.

Contents

Chapter 1

Introduction

Due to the need for high-performance computers, the microarchitecture of modern CPUs is getting more and more complex with every new generation. However, the microarchitecture of modern CPUs is highly security-critical. With the demise of Moore's Law comes the need for new optimization methods like speculative execution, branch prediction and adaptive memory management. With increasing complexity, it gets harder and harder to evaluate the security of modern CPUs. In recent years, it has been shown that the microarchitecture of modern CPUs contains flaws that provide a surface for attacks. Attacks like Rowhammer [1], Spectre [2] and Meltdown [3] have shown that the microarchitecture of modern CPUs is vulnerable and needs extensive security research. To detect and mitigate those flaws, it is necessary to understand and describe the inner mechanisms of the CPU.

An example of such a mechanism is the behavior of caches. The last-level cache (LLC) is split into multiple pieces, which are called *slices*. To decide to which slice of the LLC a memory address maps, there exists a hardware circuit inside the CPU that calculates a simple logical hash function. This hash function is present in most modern CPUs from manufacturers like AMD and Intel, but is entirely undocumented. Previous work was able to reverse engineer the addressing function of many CPUs [4–9]. To reverse engineer these hash functions, we need a way to measure individual microarchitectural events like cache lookups. We can measure those events using timing-differences [4, 6, 7, 10] or hardware monitoring features like *performance counters* [5, 9]. The function of CPUs, where the number of cores is not a power of two, is non-linear, which makes it harder to reverse engineer [7, 9] as it requires far more measurements. The number of measurements that are needed for reverse engineering grows exponentially with the number of bits that are used as input for the function. The state-of-the-art techniques require triggering an LLC lookup 10,000 or even 100,000 times to achieve an acceptable performance [7, 9]. When the number of input bits is high, the measurements take a lot of time. Gerlach et al. reported that the measurements for reverse engineering a $32\,\mathrm{bit}$ function take $5\,\mathrm{d}$ [9].

We identify several reasons that make microarchitectural reverse engineering difficult. First, the OS and processes that are running on other cores introduce environmental noise, which makes low-level measurements inaccurate. This increases the number of required measurements and the overhead of single measurements. Second, many low-level measurements require special privileges that measurement applications do not have in classical settings. While there are ways to circumvent these restrictions, they introduce runtime overhead and environmental noise. Lastly, measurements require accessing specific physical memory addresses or large regions of contiguous physical memory, which is difficult to achieve on classical systems.

We propose to set up a measurement environment using a special kernel design, a *unikernel*. A unikernel is a single-use operating system where the application that is supposed to run on the kernel is compiled together with the kernel [11–15]. This means that we can 1) completely remove all runtime overhead and corresponding noise from the operating system, and 2) directly access arbitrary kernel functionality. This enables us to set up a noise-free environment for microarchitectural measurements, where all measurement code is executed at the highest privilege level. We create a specialized research unikernel that can boot bare-metal. We also provide libraries to measure microarchitectural events using performance counters or timing differences. Our kernel is based on Unikraft [12], an open-source development kit for unikernels. Unikraft provides support for C and C++, including a reduced version of the C standard library. Our kernel maps virtual memory 1:1 to physical memory and allows access to arbitrary addresses available to the kernel.

We evaluate our microarchitectural research kernel by reverse engineering the LLC-addressing function of modern Intel CPUs and show that we can significantly improve the measurement accuracy and efficiency. We can pinpoint LLC lookups to single instructions with $100\,\%$ accuracy, which removes the need for successive accesses. This enables us to speed up measurements from $982\,260$ cycles to $25\,692$ cycles, which is 39 times faster. We reverse engineer the LLC-addressing function of an Intel Core i7-8700 CPU that uses a non-linear hash function. Our results show that our approach is 264 times faster than the state-of-the-art approach, as our measurements are more accurate and efficient. We further reverse engineer the previously unknown function of an Intel Core i7-9700K CPU. Our results prove that unikernels are a useful tool for microarchitectural reverse engineering.

This research kernel can be the first step towards a unified measurement framework for microarchitectural reverse engineering, side channel prototyping and CPU benchmarking. As it is fast and efficient, researchers can use it to perform fine-grained measurements. As the Unikraft core is actively maintained by a growing number of developers, our research kernel will likely get more and more features in the future.

Contributions. To summarize, this thesis makes the following contributions:

1. We identify the main challenges that microarchitectural reverse engineering is facing, which are environmental noise, missing privileges, and restricted memory access.
2. We show how we can overcome these issues by leveraging the power of unikernels and design a research kernel that can be used to set up a noise-free measurement setup for microarchitectural reverse engineering.
3. We evaluate our research kernel by reverse engineering non-linear and linear last-level cache addressing functions of modern Intel CPUs, and show that we can measure cache lookups with single-instruction granularity and improve the speed of the state-of-the-art approach by a factor of 264.

Outline. Chapter 2 provides background. Chapter 3 identifies the challenges of microarchitectural reverse engineering and shows how we overcome these by using unikernels. Chapter 4 is a case study in which we reverse engineer LLC-addressing functions. Chapter 5 discusses approaches and limitations of previous methods for LLC analysis and other research kernels. Chapter 6 discusses limitations and future work. Chapter 7 concludes.

Chapter 2

Background

2.1 Microarchitectural Hash Functions

Microarchitectural hash functions are responsible for distributing data to different components in the CPU [16]. The microarchitecture of modern CPUs contains many low-level functions that are implemented in circuits as part of the CPU [9]. Those low-level functions perform simple arithmetic operations like addition, multiplication, and other simple operations that the CPU needs to perform to execute instructions. For example, a modern processor contains many addition circuits that are used for different purposes, like address offset calculation, jump destination calculation or incrementing the instruction counter. One group of these low-level functions are microarchitectural hash functions. Microarchitectural hash functions can be used to distribute accesses to multiple components that implement the same functionality. E.g., the CPU wants to distribute physical address accesses to multiple DRAM columns and banks to store the data of that address [1, 17]. The main idea of this strategy is to address all components equally often, such that the CPU can reduce interference from successive accesses and uneven resource pressure on components [18].

$$H : \{0,1\}^n \mapsto \{0,1\}^m = \begin{cases} h_0 : \{0,1\}^n \mapsto \{0,1\} \\ h_1 : \{0,1\}^n \mapsto \{0,1\} \\ \vdots \\ h_m : \{0,1\}^n \mapsto \{0,1\} \end{cases} \tag{2.1}$$

A binary hash function H maps an input with n bits deterministically to a fixed set of buckets addressed with m bits, as we see in the left part of Equation (2.1) [19]. The first structural requirement for microarchitectural hash functions is that they distribute uniformly over all

buckets. Another structural requirement for microarchitectural hash functions is that they have low latency, i.e., the depth of the corresponding circuit is small. Therefore, microarchitectural hash functions can usually be expressed as a small sequence of logical operations [9, 16]. Opposed to cryptographical hash functions, microarchitectural hash functions do not fulfill any cryptographic properties like collision resistance or preimage resistance, as the primary goal is performance [16, 18].

When reverse engineering microarchitectural hash functions, we can simplify the process by handling each bit of the output separately [6, 9]. We therefore split the function H into multiple functions $h_0, h_1, \ldots, h_m$, one for each bit in the output, as we can see on the right side of Equation (2.1).

2.2 Caches

The Cache Hierarchy. Computers use caches as an intermediate storage for data that is frequently accessed to improve general performance by exploiting spatial and temporal locality. Most modern processors use i-way caches. The cache is divided into cache sets containing i ways, each with a size of 64 bytes. In most modern x86 processors, there are three main caches that are structured hierarchically in three levels (L1, L2, LLC) of different sizes (Figure 2.1). The higher the cache is in the hierarchy, the faster it is, but the less storage it has. On recent Intel processors, the L1 cache is further divided into data-cache and instruction-cache, while the L2 and LLC are unified caches, i.e., they can contain instructions and data.

Last-Level-Cache Slicing. Whereas the L1 and L2 caches exist per core, the LLC is shared between all cores. The LLC is split into slices on Intel machines, as this allows parallel access to the LLC from different cores at the same time if the memory addresses map to different slices [6]. In this way, contention is reduced. There is one slice in the LLC per core. In newer processors, where multiple logical cores can share the same physical core, there usually exists one slice per logical core. [20, 21]

Slice-Addressing. To decide to which slice of the LLC an address maps, there is a microarchitectural hash function that maps physical addresses to cache slices. This function is called the LLC-addressing function. Previous work has shown that on Intel systems, the function can be expressed as a sequence of XORs if the function is linear in the number of bits, i.e., the number of cores/slices in the system is a power of two [5]. On AMD systems or when the number of cores is not a power of two, the function is more complex [9].

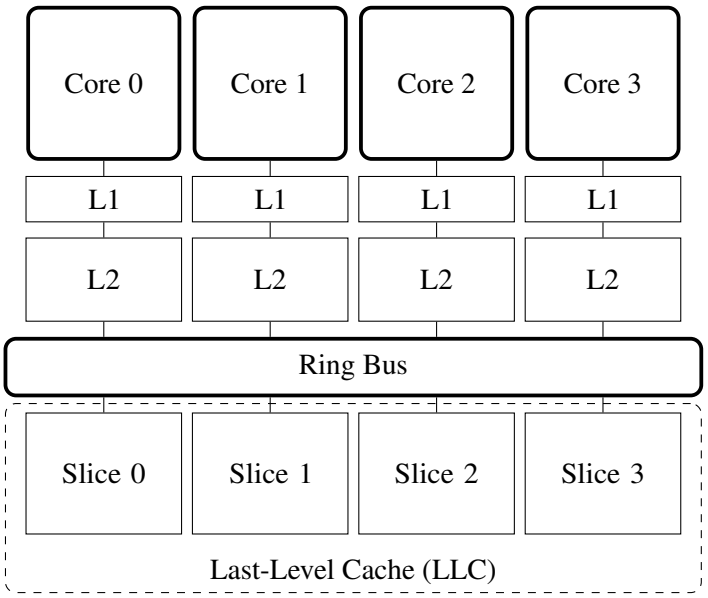


Figure 2.1: Cache hierarchy of a modern quad-core intel CPU

Last-Level-Cache Attacks. As the LLC is shared between cores, an attacker who exploits the structure of the LLC can spy on processes that run on other cores of the same machine. As this increases the attacker's capabilities, the LLC is an interesting target for cache attacks [10, 22, 23]. The Flush+Reload attack [22] was one of the first cross-core attacks, directly targeting the LLC. The original attack strategy of Flush+Reload is to observe the presence of data in the cache that is shared between processes, either in the form of shared memory or due to the copy-on-write mechanism. If the read-access to the page is fast after flushing the address, it was likely accessed by the victim process. Additionally, Liu et al. [23] have shown that the Prime+Probe attack is possible against the LLC without relying on shared memory. Later work by Kayaalp et al. [10] shows that this attack can be improved by reverse engineering the LLC-addressing function.

2.3 Microarchitectural Reverse Engineering

The goal of microarchitectural reverse engineering is to understand the inner mechanisms of components of the microarchitecture. Components that are interesting targets for reverse engineering are, e.g., cache addressing [4–7, 9], cache replacement policies [24], the memory controller [25] or undocumented model-specific registers [26].

Knowledge of the inner mechanisms of the microarchitecture allows researchers to 1) uncover security flaws, 2) propose mitigations against existing vulnerabilities, and 3) build specialized high-performance software. For example, knowledge of the LLC-addressing function can be used to build novel high-resolution cross-core cache attacks [10], create protected partitions of

the LLC for security-critical processes [7], and improve the performance of applications with a slice-aware memory management scheme [27].

When reverse engineering a functionality of the microarchitecture, we need to observe the behavior of the target component with low-level measurements. For example, if the target component is the LLC-addressing function (Section 2.2), we have to observe the mapping between a physical address that is accessed in the LLC and the slice that was accessed for this address. From these measurements, we can then infer how this mapping is created. To perform these measurements, we need one or multiple measurement processes that trigger the mechanism in the target component and then observe the resulting behavior, e.g., a process triggers a lookup in the LLC and then observes the accessed slice.

To measure how a component of the microarchitecture behaves in a specific context, we need a way to monitor microarchitectural events. This can be done, e.g., with timing differences or by using performance counters. Performance counters are a special type of model-specific register (MSR) that can count microarchitectural events. In the x86 instruction set architecture (ISA), there exist multiple types of performance counters. In contrast to fixed-function performance counters that count specific events, general-purpose performance counters can be programmed to monitor specific events. Additionally, there exist *uncore performance counters*, that count events in the LLC.

2.4 The Concept of Unikernels

Unikernels are a new approach for building single-application, high-performance operating systems. A unikernel is compiled together with the application into a standalone image. In this section, we explain the concept of unikernels in a bottom-up approach, starting with a general definition of operating systems (Section 2.4.1) and then explaining the concepts of library operating systems (Section 2.4.2) and unikernels as a special form of the previous (Section 2.4.3).

2.4.1 Operating Systems

An operating system (OS) can be defined as software that manages hardware. The main part of the OS is the kernel, which is responsible for allocating physical resources such as execution time on the central processing unit (CPU), memory, I/O devices, and storage to applications running on the system. As these operations are security-critical, the code of the kernel runs in privileged *kernel mode* on the CPU. Applications run in a less privileged *user mode*. If an application needs to perform a privileged action, it has to request the kernel to execute it with a syscall. An application can run directly on the CPU but is regularly forced to hand control over to the kernel.

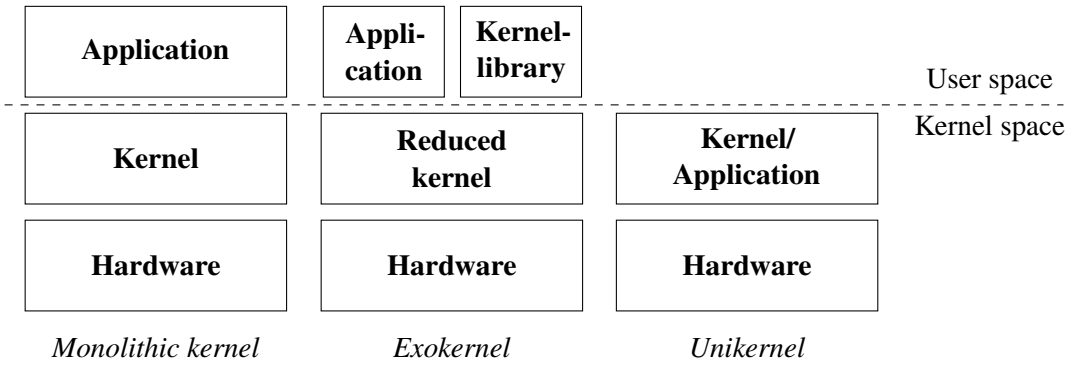


Figure 2.2: Comparison: OS structure of different architectures

This introduces a division of labor between the application and the kernel and introduces a layer of abstraction. The application runs on top of this layer of abstraction (Figure 2.2) and can rely on the kernel for low-level management. [28]

Monolithic Kernels. Linux builds upon monolithic kernels. Monolithic kernels provide a general-purpose, high-level abstraction of all hardware resources. An application running on the kernel does not need to care about other applications running on the system, device drivers or the details of memory management. Each application has its own virtual address space and does not need to worry about potential conflicts. This high-level abstraction of the underlying hardware makes developing applications for those systems very comfortable, but comes at the cost of performance. [29]

2.4.2 Library Operating Systems

The design philosophy of library operating systems is to reduce the responsibilities of the kernel to enable better performance. An inherent problem with high-level abstractions is that an implementation of kernel features cannot be efficient for all applications [30]. For example, database management systems require memory management capabilities like buffering or consistency control while retaining a high efficiency. However, always providing such primitives may decrease performance for applications that do not require them [31]. Due to this problem, Engler et al. proposed a new kernel design called *exokernel* [32]. This design was the first in a new group of operating systems and is also referred to as *library operating systems* [33]. The main idea of library operating systems is to lower the abstraction introduced by the kernel close to the hardware level. Important abstractions, like memory allocator, scheduler or device drivers that were previously managed by the kernel, now run in a library fashion as part of the user space. As we see in Figure 2.2, parts that were originally part of the kernel run in the same address space as the application on top of the privileged reduced kernel.

The system design is fully modular. The kernel is solely used to multiplex, allocate and deallocate hardware. A developer can choose between different implementations for scheduler, memory allocator, or threading in a library fashion. This enables the developer to choose the implementation of the operating system functions that are best suited for the application they implement. If the developer does not find an efficient implementation for their purpose, they can even implement one themselve. This design allows creating efficient operating systems for targeted use cases, running 10-100 times faster than systems with comparable monolithic kernels at that time [32].

The concept of library operating systems is closely related to the concept of *microkernels*. Like in library operating systems, the goal is to reduce the size of the kernel. However, instead of exposing the full low-abstraction hardware functions, microkernels provide a reduced set of high-level abstraction functions. The focus of microkernels lies on flexibility and security rather than efficiency. [30, 32]

2.4.3 Single Purpose Operating Systems: Unikernels

Unikernels are special library operating systems that only provide one single application. The main difference between unikernels and classical library operating systems is that the application is compiled together with the kernel into one binary. Compiling the application as part of the kernel allows for several design changes. First, we do not need context switches. As the application is part of the kernel, it can perform arbitrary operations without requesting permission from the kernel. This allows the system to remove any overhead that is introduced by context switches [34]. Second, the application runs in the same address space as the kernel, as we see in Figure 2.2. [11, 12]

Normally, unikernels are intended to be used on top of a hypervisor in a cloud environment [13], which is also the idea of early unikernel designs like mirage [11]. The main goal in this setting is to increase efficiency and reduce the size of cloud images. In this setting, portability can be guaranteed as the real hardware is abstracted by the hypervisor. However, deploying unikernels bare-metal is possible.

There exist several unikernel solutions that support different platforms, application languages [12, 13] or are specialized for specific tasks like network routing[14]. Recently, there is also an ongoing effort to implement unikernel functionality for Linux [15].

2.5 Unikraft

Unikraft [12] is an open-source development kit for unikernels. It allows building unikernels for different hypervisors, including KVM, Xen, HyperV and VMware, and different instruction set architectures (ISA) like Arm, x86 and RISC-V. Currently, Unikraft does not support booting bare-metal, but this is planned for future versions. The key design decisions of Unikraft kernels align with the idea of unikernels and library operating systems:

1. Single address space
2. Fully modular system design
3. Single protection level

The goal of the Unikraft architecture is to make building specialized applications as easy as possible, which they achieve through their modular design and a configurable build system.

For every kernel function, like networking, memory allocation, scheduling and booting, they define a low-level API. These APIs are then implemented by different micro-libraries so that developers can choose the modules that best suit their application and so that it is easy to implement a custom version for these APIs. The key concept is that there exist multiple microlibraries that implement a certain kernel API. E.g., applications can use the *ukalloc* API to communicate with the memory allocation subsystem. However, the *ukalloc* API can be served by specialized allocator implementations like tinyalloc, mimalloc or the oscar secure memory allocator. Another advantage of microlibraries is that developers can plug in custom implementations at different abstraction levels. E.g., for networking, developers can use the standard socket-based API or implement a custom version of the lower-level *uknetdev* API. While low-level kernel functionality is provided, it is even possible to entirely disable components like scheduling or multicore execution.

For compatibility, Unikraft provides a ported version of *musl*[1], an implementation of the C standard library. To make porting existing applications to Unikraft easier, Unikraft provides a syscall-shim-layer, which enables applications that require syscalls to run on a system without a kernel. As the libraries are statically linked, the abstraction boundaries between the application and low-level kernel functionality can be short-circuited during compilation. This increases the performance of the compiled kernel. Kuenzer et al. showed that building an application with Unikraft can improve the performance to $170\,\%$-$270\,\%$. The image size of a compiled kernel can be as small as $1\,\mathrm{MB}$, boot in $1\,\mathrm{ms}$ and use less than $10\,\mathrm{MB}$ of random access memory (RAM). [12]

[1] `https://musl.libc.org/`

Chapter 3

Unikernels for Reverse Engineering

In this chapter, we show how unikernels can be used for microarchitectural reverse engineering. At first, we identify the main challenges of microarchitectural reverse engineering (Section 3.1). We then show how we can use unikernels to face these challenges (Section 3.2). Lastly, we explain how this can be achieved technically (Section 3.3).

3.1 Challenges in Microarchitectural Reverse Engineering

In microarchitectural reverse engineering, researchers have to face several challenges. First, environmental noise makes low-level measurements inaccurate and also decreases the efficiency of reverse engineering methods, as more measurements are needed to filter out noise (Section 3.1.1). Second, researchers often need to modify or read structures that are only accessible in kernel mode, which is challenging in classical measurement setups (Section 3.1.2). Lastly, researchers often need to access large contiguous memory regions or modify paging structures, for which classical kernels often do not provide suitable features (Section 3.1.3).

3.1.1 Environmental Noise

Environmental noise is interference that is produced by 1) other processes running on the measurement system and 2) the underlying OS itself.

As mentioned in Section 2.3, researchers use a measurement process to trigger specific microarchitectural mechanisms. However, the behavior of the target component is not only dependent on the measurement process but may also be affected by other processes running on the system, as these might also trigger the mechanism that a researcher wants to observe. This can produce

errors in the measurements and reduce the accuracy. E.g., the LLC is shared between cores and therefore also influenced by processes that run on different cores than the measurement process. If these other processes trigger an LLC lookup during the measurement, the measurement process observes the corresponding slice access and cannot distinguish if the slice access was triggered by itself or by another process.

Furthermore, the measurement process is regularly interrupted and can also be interrupted by hardware peripherals, such that the OS can take control over the CPU. This is called a *context switch*. The complete state of the CPU of the measurement process is saved to memory, and the OS CPU state is loaded. Now the OS can perform scheduling, resolve requests or manage hardware events. If the OS is finished, it restores the CPU state of the measurement process. The measurement process can now continue and is unaware of the interruption, if it was not issued by itself via a syscall. The operations done by the OS also affect the state of the microarchitecture and leave it in an inconsistent state. Especially, the interrupts can influence the results reported by performance counters, which leads to measurement errors [35, 36]. E.g., the OS also triggers lookups in the LLC, which increment performance counters that monitor cache lookups. This leads to inconsistent results for the measurement process.

3.1.2 Missing Privileges

Performing low-level measurements often requires special privileges that user-space applications do not have in classical settings. While some of these requirements can be circumvented, they make low-level reverse engineering challenging in classical measurement setups based on common monolithic kernels like the Linux kernel.

In Section 2.3, we explained that microarchitectural events can be observed by using performance counters. However, interacting with most model-specific registers requires kernel privileges. Model-specific registers can be read with the *rdmsr* and written with the *wrmsr* instruction [37]. Both of these instructions can only be executed in kernel mode and otherwise generate a general protection exception. While there exists an instruction (*rdpmc*) to read performance counters in user mode, it cannot access all performance counters [37]. For example, uncore performance counters cannot be read in user mode. If researchers want to monitor lookups in the LLC to reverse engineer the LLC-addressing function, they need kernel privileges to program and read uncore performance counters. While it is possible to read and write performance counters on Linux using tools like *libpfc*[1], this introduces measurement overhead and noise.

Furthermore, the behavior of the CPU can be modified, which can be very helpful for measurements but is only possible in kernel mode. First, the behavior of the CPU can be controlled

[1] GitHub: `https://github.com/obilaniu/libpfc`

with special instructions. These instructions can only be executed in kernel mode, which is not possible for a user-space process. E.g., there are instructions like *cli* and *sti* to disable and enable hardware interrupts. These instructions can be useful to reduce environmental noise. Second, the behavior of the CPU can be modified by changing values in MSRs. For example, MSR 0x48 in Intel processors can be used to specify where the processor is allowed to speculate, or MSR 0x6a0 can disable or enable security features like the shadow stack. Often, such MSRs cannot be overwritten in classical operating systems, as this would break functionality that the operating system depends on.

3.1.3 Restricted Memory Access

Reverse engineering of memory-related components of the microarchitecture like caches, the memory controller, or prefetchers for data often requires access to special physical addresses or a specific paging structure. As physical memory is managed completely by the kernel on monolithic operating systems, this is a restriction for reverse engineering. [9, 24, 25]

Each application runs in its own virtual address space. Virtual addresses are mapped to physical addresses via page tables. Page tables are managed by the kernel. Therefore, a measurement process that wants to do measurements based on physical addresses needs to find the corresponding physical address for every virtual address. Sometimes researchers even need to access specific physical addresses or large, contiguous physical memory regions. On Linux, contiguous physical regions can be requested with *kmalloc*, which is limited to a size of $4\,\mathrm{MB}$ [24]. This size is often not sufficient for a detailed analysis. E.g., to reverse engineer the LLC-addressing function, researchers want to measure an LLC lookup for an arbitrary set of addresses to get a representation that is correct for all areas of the physical memory.

Another way of getting access to specific physical addresses or large physical memory regions is to directly modify the page table itself. However, free regions in physical memory are fragmented. While it is also possible to perform measurements on pages that are used by other processes, this might introduce measurement errors as their microarchitectural state might be modified by the owner process. [9]

3.2 Unikernels as Research Environment

In this section, we show why unikernels (Section 2.4.3) are useful for microarchitectural reverse engineering and how they can be used to overcome challenges identified in Section 3.1: Environmental noise (Section 3.1.1), missing privileges (Section 3.1.2) and restricted memory

management (Section 3.1.3). Additionally, unikernels run very stable, which enables new analysis techniques.

First, unikernels do not produce microarchitectural noise. The design goal of unikernels is to remove any overhead introduced by abstraction layers to improve performance and efficiency [11]. However, this goal directly aligns with our goal of noise reduction, as removing execution overhead also removes the corresponding noise. The most important abstraction layer that is not present in unikernels is the division between kernel and application. As the measurement process is the only process running on the system and is never interrupted by the OS, we remove every OS noise that would be present on monolithic kernels. Second, we do not have to care about privileges when using unikernels. As there is no separation between kernel and application, the application has kernel privileges. This means that the measurement code can contain arbitrary privileged instructions. It can also read and write any MSRs and performance counters. Furthermore, a custom kernel allows the researcher to directly modify the paging behavior or might allow him to disable paging completely.

Further advantages of unikernels are their high stability and deterministic execution. The minimal nature of unikernels allows changing the CPU behavior arbitrary, as there is no underlying OS that depends on certain CPU functionalities. Moreover, the kernel executes measurement code deterministically, as there is no interference with other processes. This allows for reproducible measurements and can also allow for special analysis techniques like cycle-by-cycle execution [38].

3.3 Building a Measurement Environment

To evaluate unikernels for microarchitectural reverse engineering, we create a research unikernel that can boot bare-metal on modern Intel processors, which we can use as a low-noise research environment. We also introduce libraries that make microarchitectural reverse engineering easier and a network protocol to export measurement data to other computers.

Our kernel is based on the Unikraft version 0.10.0 (Phoebe) [2]. The original version of our kernel was created in collaboration with Robert Pietsch within the scope of a cybersecurity project. We reworked PCI enumeration, added support for Multiboot v2, and created a custom network card driver. We also implemented a library to read and write model-specific registers and functions that can be used for fast cache-attack prototyping. In the following, we call our modified Unikraft Phoebe kernel 'research kernel'.

[2] https://unikraft.org/blog/2022-08-20-unikraft-releases-phoebe/

To make our measurements as fine-grained as possible, we removed as many layers of abstraction as possible. The original Unikraft core (Section 2.5) is intended to be used on top of a hypervisor. However, we need the research kernel to be able to boot bare metal to remove the overhead that is introduced by the compatibility and scheduling layer of the hypervisor. To be able to boot directly on hardware, it was necessary to remove the dependency of the Unikraft Phoebe core on the hypervisor.

As we solely use the research kernel to perform and display measurements, it is necessary to export data to another medium. Unikraft does not provide drivers for USB storage. To transfer data efficiently, we use the custom network driver to implement a transfer protocol for binary data using the low-level *lwip*[3] library. This protocol can then be used to exchange data with other computers in the same network.

Custom Page-Tables. To make microarchitectural measurements easier, we modify the page table of the Unikraft kernel such that the virtual memory is mapped 1:1 to the physical memory for all available RAM. This allows us to work with virtual addresses as if they were physical addresses when performing measurements. To achieve that, we fill the hard-coded page table of the unikernel with $2\,\mathrm{MB}$ pages to be compatible with systems that do not support $1\,\mathrm{GB}$ pages. While we could also completely disable paging and only use $32\,\mathrm{bit}$ addressing, this makes it impossible to address memory larger than $4\,\mathrm{GB}$.

[3] `https://www.nongnu.org/lwip/2_1_x/index.html`

Chapter 4

Case Study: Reverse Engineering of Last Level Cache Addressing

In this first case study, we show that we can reverse the LLC-addressing functions of multiple processors using our custom unikernel with $100\,\%$ accuracy and 39 times fewer cycles than the state-of-the-art approach. Combining the advantages of accuracy and efficiency, we can speed up the total time for measurements by a factor of 264. Our reverse engineering approach is based on the work of Gerlach et al. [9] and adapted for use in our unikernel.

At first, we show how we can measure to which slice a physical address maps (Section 4.1). This mapping can then be used to find a small representation of the LLC-addressing function (Section 4.2). Using our measurement setup (Section 4.3), we evaluate the correctness, accuracy, and efficiency of our method (Section 4.4).

4.1 Mapping Address to Slice

In this section, we show how addresses can be mapped to slices. Our approach (Algorithm 4.1) is based on the algorithm by Maurice et al. [5] and slightly modified for the setting in our unikernel. At first, we set up the uncore performance counters corresponding to the LLC slices to monitor *LLC_LOOKUP* events. The concrete registers and configuration values to program the performance counters are CPU-specific. After the setup of all required performance counters, we can enable all counters simultaneously via a single *wrmsr* instruction. During our measurement, we disable all interrupts with the *cli* instruction and enable them afterward with *sti*.

Our goal is now to trigger an LLC lookup event and measure it using performance counters. As the LLC contains code and data, executing measurement code can trigger LLC lookups even if

Algorithm 4.1 Mapping Address to Slice

Input: physical address $paddr$
Output: slice index s

```
1: for each slice do
2:     setup monitoring via LLC_LOOKUP event
3: end for
4: warmup_code()
5: start_counters()                          // Start of the monitoring session
6: clflush(paddr)
7: stop_counters()                           // End of the monitoring session
8: for each slice do
9:     read performance counter
10: end for
11: s ← get slice with most LLC_LOOKUP events
12: return s
```

the code does not access memory, which leads to measurement errors. To avoid measurement errors, we execute warm-up code (l. 4) that loads all required data into the higher-level L1 and L2 caches. In this case, we need to execute the code that is part of the monitoring session (l. 6-7) before performing the measurement. Note that the *wrmsr* instruction to stop the performance counters should be part of the warm-up code, as it needs to be fetched before the CPU can execute it.

To trigger a lookup in the LLC for a specific physical address, we use the *clflush* instruction (l. 6). *clflush* invalidates the cache line of a given address in every cache level of the cache hierarchy and therefore causes a lookup in the LLC even when the line is not present or the address is not readable [37]. Note that *clflush* expects a virtual address as an input, but we want to access a physical address. However, we can use the physical address as an input in our setting, as the virtual memory in our research unikernel is mapped 1:1. Each virtual address is equal to the corresponding physical address.

After this measurement, we immediately stop all performance counters (l. 7) with a single *wrmsr* to prevent measurement errors. Then we read the contents of the corresponding counter for all slices. We can then assume that the slice with the most traffic is the slice that the physical address maps to (l. 11).

4.2 Reverse Engineering LLC Addressing

The key idea of our reverse engineering approach is to measure a significant number of mappings between physical addresses and cache slices while taking the structural requirements of the

architectural implementation into account. Using these mappings, we can then infer a logical function.

As mentioned in Section 2.1, we split the hash function H into multiple functions h_b per bit b in the output of H (Equation (2.1)). The reverse engineering pipeline is visualized in Figure 4.1. The first step, described in Section 4.2.1, is to identify which bits of the physical address are used in the function h_b. If the microarchitectural hash function is linear, we can directly infer the function, as we just need to XOR all relevant bits. Otherwise, we build a query set of relevant addresses by choosing one address per possible combination of relevant bits. Using this query set, we measure the slice for each address as described in Section 4.2.2 and build a table of addresses and slice indices. This table can be viewed as the truth table of a binary function h_b. Using this table, we infer h_b using logic minimization techniques as described in Section 4.2.3.

4.2.1 Identifying Relevant Address Bits

For our measurements, relevant bits of the physical address are defined as bits that influence the outcome of the LLC-addressing function. In our approach, we identify relevant bits for each function h_b separately because this makes the reverse engineering process easier.

Assuming that the LLC-addressing function is linear, the approach is as follows. Algorithm 4.2 shows how we decide if bit p of the physical address is relevant for the slice-bit function h_b. We choose a pair of addresses that differ in bit p. Then we measure the slices of both addresses and check if the slices differ in bit b. If the slice differs in bit b, we know that the physical address bit p is XORed in the function h_b, as it changed the result. Otherwise, we can conclude that p should not be XORed. We do this for every function bit b and can directly obtain h_0 to h_m, and the reverse engineering process is finished.

If the LLC-addressing function is non-linear, we cannot directly conclude that bit p is irrelevant for h_b, as p might only be relevant in a certain combination of bits. However, it is in the interest of chip manufacturers to maximize the influence of each bit in the function, as this means distributing the data uniformly. We therefore measure the relevance of bit p for multiple random address pairs. If the bit influenced the result in at least one case, it is considered relevant. We observe that it suffices to query the slice function 100 times, as a bit does not have an influence in few cases, which aligns with the design goal of the slice-addressing function, that all buckets should be addressed uniformly for a random set of addresses. We do this for every function bit b and can obtain the relevant bits for the functions h_0 to h_m.

Algorithm 4.2 Identify if bit p of the physical address is relevant for function h_b

Input: slice bit b, physical address bit p
Output: $true$, if bit p is relevant vor h_b

```
addr_1 ← choose random address
addr_2 ← flip bit p of addr_1
slice_1 ← measure slice of addr_1
slice_2 ← measure slice of addr_2
if slice_1 and slice_2 differ in bit b then
    return true                          // Bit p is relevant for h_b
else
    return false                         // Bit p is not relevant for h_b
end if
```

4.2.2 Measuring Relevant Addresses

The goal of this step is to find a relation between the relevant bits of the function h_b and the output of h_b. To accomplish this, we can build a truth table of all possible combinations of relevant bits and the slice bit b, which is already a representation of the hash function h_b. To get the bit b of a combination of relevant bits, we create a representative address containing this combination of bits and fill the remaining bits with zeros. When we now measure the slice that the representative address maps to, we can obtain bit b for this combination of relevant bits. We do this for every possible combination of relevant bits and get a truth table for h_0 to h_m. While these truth tables are a valid representation of the functions, they are large and inconvenient.

4.2.3 Logic Minimization

As it is in the interest of chip manufacturers that the LLC hash function has low latency, we assume that there exists a compact form of the function h_b. To find this compact representation of the function, we use the logic minimization techniques developed by Gerlach et al. [9].

The truth table can be transformed into a logic formula in disjunctive normal form (DNF) by applying the canonical algorithm. The DNF can then be minimized using the ESPRESSO algorithm [39]. The next step is to 1) translate the boolean function to a system of polynomial equations, 2) find a smaller representation of this system by calculating the Gröbner basis, and 3) translate the resulting smaller system of polynomial equations back to a boolean formula. These three steps can be applied recursively until the boolean representation converges.

This algorithm can be used to find compact representations of all functions, h_0 to h_m, which then gives us a compact representation of H for non-linear functions.

Figure 4.1: Pipeline for reverse engineering the LLC-addressing function

4.3 Measurement Setup

We perform measurements on two different machines and reverse engineer one linear and one non-linear cache slice function. For our measurements, we use our research kernel (Section 3.3). We further configure our kernel to use Multiboot v2 for booting, run on one core, and disable threading. For our measurements, we exclude memory ranges where we suspect memory mapped I/O, as flushes in those regions lead to incorrect lookups in the LLC.

For the *non-linear* slice function, we perform our measurements on a Dell OptiPlex 7060 with an Intel Core i7-8700 CPU, Family 6, Model 158, stepping 20. This model has 6 physical cores and therefore 6 cache slices. To address 6 slices, the LLC-addressing function needs 3 output bits. With this setup, we then identify the relevant bits for h_0, h_1, and h_2. We identify all non-linear bits in the function and create a truth table for each of them, while the function for all linear ones can directly be obtained. We verify our measurements with the closed form of this function that was obtained by Gerlach et al. [9]. To compare the accuracy and efficiency of our approach to the approach of Gerlach et al., we install the framework of Gerlach et al. on our test machine and perform all measurements in both setups.

For the *linear* slice function, we perform measurements on an HP Z1 Entry Tower G5, equipped with an Intel Core i7-9700K CPU, Family 6, Model 158, stepping 13. It has 8 physical cores and 8 cache slices. The slice addressing function therefore has 3 output bits. In our measurements, we observe that none of the known slice-addressing functions describes the behavior of this CPU. We verify our measurements by checking the results for $1\,000\,000$ random addresses.

4.4 Evaluation

To evaluate our research kernel, we assess the accuracy and efficiency of our reverse engineering approach and compare it to the state-of-the-art approach by Gerlach et al. [9]. To assess the accuracy, we measure the amount of noise that is registered by the performance counters. We show that we can measure individual LLC lookups that are triggered by a single instruction without encountering environmental noise. This means that it suffices to flush the physical address 1 time with our method, in contrast to $10\,000$ flushes with the state-of-the-art approach. It follows

	Gerlach et al.	**Our Method**
Slice 0	21 028	0
Slice 1	20 824	0
Slice 2	22 556	0
Slice 3	20 671	0
Slice 4	23 358	0
Slice 5 (*)	1 024 663	1 000 000
A (random)	90 %	100 %
A (10,000 samples)	97 % (+- 9 %)	100 % (+- 0 %)

*: Correct slice

Table 4.1: Exemplary number of slice accesses and accuracy A (Intel i7-8700 CPU)

that an individual measurement is much faster with our new approach. To evaluate the increase in efficiency, we measure the number of cycles required to map a physical address to a slice. Our results show that our method is 39 times faster than the method by Gerlach et al. in terms of cycles for individual measurements. To show that our method is faster in terms of total runtime, we compare the runtime of performing the measurements to reverse engineer a non-linear hash function. Our results show that our method is 264 times faster than the state-of-the-art approach. We also report the hash functions that we reverse engineered.

Single Access Accuracy. At first, we evaluate the accuracy at performance counter level. We set up uncore performance counters for all slices present in the system and measure LLC lookup events. We then choose a random address in the addressable address space and flush this address 10 000 times. The expected result is 10 000 lookups on the slice that the address maps to and 0 lookups on the other slices, which can be seen in the third column of Table 4.1. The state-of-the-art technique records plenty of cache accesses on different slices and too many accesses on the expected slice due to environmental noise. While the measurements with the state-of-the-art technique record environmental noise, the measurements with our method are noise-free. To calculate the accuracy A, we use the formula in Equation (4.1). Let y_i denote the expected access on slice i, y'_i denote the measured accesses on slice i and S the set of slices.

$$A = 1 - \frac{\sum_{i \in S} |y_i - y'_i|}{\sum_{i \in S} y'_i} \tag{4.1}$$

Using our measurement of accuracy A, we observe that we take the accuracy of a random measurement from 97 % accuracy of the state-of-the-art technique to 100 % accuracy on performance counter level. Note that the accuracy of a random measurement (90 %) is lower than the average accuracy of 10 000 samples. This is because multiple measurements in short succession provide an implicit warm-up of the cache state, which improves the overall accuracy of the measurements. All in all, our method can measure individual microarchitectural events like LLC lookups with

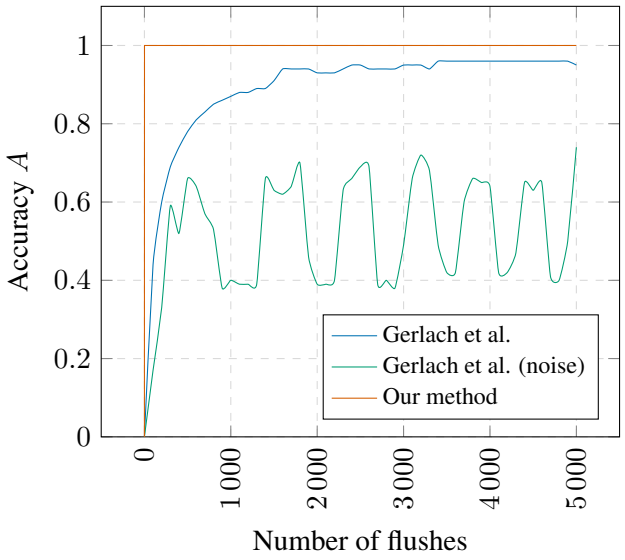


Figure 4.2: Accuracy A of performance counter measurements (Intel i7-8700)

perfect accuracy.

Number of Flushes. As there is no noise in measurements on our kernel, it suffices to flush 1 time for measurements on our kernel. This is a significant improvement from 10 000 flushes that are needed in the approach by Gerlach et al. In Figure 4.2 we show the dependency between the number of flushes and the accuracy A of the performance counter measurements. We see that our method only requires 1 flush to achieve perfect accuracy, whereas the approach by Gerlach et al. requires 3000 measurements to achieve an accuracy of at least 95 %. To evaluate how much the measurements of the approach by Gerlach et al. are influenced by system traffic, we perform another measurement in the same setup but with a high-overhead application running in parallel. We observe that the measurements with the approach by Gerlach et al. are highly influenced by system noise, which leads to unstable and hard-to-reproduce results.

Measurement Cycles. Due to the environmental noise, previous methods [5, 9] that rely on performance counters flush the address 10 000 times, whereas it suffices to flush the address 1 time in our method. This improves the speed of our method compared to the previous approach. Further, we can directly execute the *wrmsr* and *rdmsr* instructions natively instead of using the Linux kernel MSR mechanism, which gives us additional speedup. To assess the speedup, we measure the CPU cycles that a single measurement takes in both approaches and compare the results. We report our results in Table 4.2. We observe that the number of cycles for a single measurement in our method ($\approx$ 25 700) is significantly lower than the cycles of the method by Gerlach et al. ($\approx$ 1 000 000). This means that measurements with our method are 39 times faster than measurements with the state-of-the-art approach. Note that if our method required 10 000 flushes per measurement, it would be around 40 % slower. These results show that our new

	Gerlach et al.	Our Method
#Cycles	982 260 (+- 123 309)	25 692 (+- 123)
#Flushes	10 000	1

Table 4.2: Comparison of single measurement speed (10 000 samples)

	Gerlach et al.	Our Method
Time	14 280 s	54 s

Table 4.3: Time needed for reverse engineering a slice addressing-function (Intel i7-8700)

approach using unikernels is far more efficient than the traditional approach using Linux because of the noise-free measurement environment and direct access to kernel functionality.

Time for All Measurements. As a single measurement in our method is 39 times faster than the method of Gerlach et al., we can reduce the time that is needed for all measurements that are required for reverse engineering. We can further reduce the total measurement time as we do not have measurement errors and therefore do not need to measure multiple times per bit. We reverse engineer 28 bit of the non-linear LLC-addressing function of an Intel Core i7-8700 CPU (former Coffee Lake) and report the required time in Table 4.3. Both tools reported the same 22 relevant input bits for each slice bit. In comparison to the time of the method of Gerlach et al. (14 280 s) our method is 264 times faster (54 s). Note that the number of addresses that we need to measure rises exponentially with the number of input bits n of the hash function. This means that the time to measure these addresses also rises exponentially to n. Measuring a function with 32 relevant input bits would take approximately 169 d for the method of Gerlach et al. in comparison to 15 h with our method. This shows that our method can be used to reverse engineer functions with more input bits than previous methods (up to ca. 32 relevant bits in one day).

Reported Functions. We reverse engineer the non-linear functions of an Intel Core i7-8700 (former Coffee Lake) and the previously unknown linear function of an Intel Core i7-9700K (former Coffee Lake). The logic representations and logic circuit for the non-linear function are presented in Appendix A. We evaluate the correctness of our function by probing the slices of 1 000 000 random addresses and comparing the resulting slice to the output of the logic function, obtained by reverse engineering. With an accuracy of 100 %, our results indicate that the functions we obtained are correct. We were able to reproduce the results of 1 known function (i7-8700) and found 1 new function (i7-9700K). We observe that the newly obtained function is a permutation of slice bit functions of the known function of another 9th Gen CPU (i9-9980HK), first reverse engineered by Gerlach et al. [9].

Chapter 5

Related Work

5.1 Low Noise Reverse Engineering

Nanobench. Abel and Reinecke [24] designed a tool for micro-benchmarking of small assembly code snippets that can also be used for microarchitectural reverse engineering. They measured the latency, throughput and port usage of many x86 instructions and reversed several cache replacement policies. To create a privileged low-noise environment for their measurements, they implemented parts of the tool as a Linux kernel module. Similar to the strategy in this thesis, they use performance counters to observe microarchitectural events. As their tool can directly operate in kernel mode, they can also use privileged instructions and can therefore lower the system noise by disabling preemption and interrupts. However, as the tool runs on top of the Linux kernel, it is limited by the available Linux kernel APIs for resource allocation. For example, reverse engineering low-level memory components of the microarchitecture requires access to specific physical memory addresses or even large contiguous memory regions. For physical memory regions, the Linux kernel provides the *kmalloc* function, which is limited to return at most $4\,\mathrm{MB}$ of contiguous physical memory. Furthermore, the measurement environment is less controllable than in a custom kernel. E.g., while it is possible to pin a measurement process to a specific core that it does not share with other processes, we cannot prevent changes in the microarchitectural state that are triggered by other cores.

Custom Kernels for Reverse Engineering. There exist multiple works that use custom kernels for microarchitectural reverse engineering [38, 40, 41]. Falk [38] developed a strategy to monitor microarchitectural events on a cycle-by-cycle basis, which means that we can pinpoint the exact cycle when a microarchitectural event happens. To monitor microarchitectural events, this strategy also relies on performance counters. To measure microarchitectural events with cycle granularity, the measurement code has to be executed multiple times. Therefore, the key

design principle of the kernel is to behave as deterministically as possible to be able to reproduce the same state of the microarchitecture multiple times. This strategy is implemented in the closed-source sushi-roll kernel. Koppe et al. [40] reverse engineer the internals of the x86 microcode semantics using a custom low-noise measurement environment (angry-OS[1]). This low-noise environment for AMD x86 CPUs supports interrupt and exception handling, virtual memory, paging, serial connection, microcode updates, and the execution of streamlined machine code. Different from our measurement kernel, it does not support programming languages like C and C++ or networking. Furthermore, Ravichandran and Na [41] reverse engineered the M1 TLB hierarchy using a custom kernel (PACMAN-OS[2]) to attack hardware-supported pointer authentication. Their kernel is written in Rust and can set up a minimal noise-free execution environment on M1 processors.

5.2 Analysis of Last Level Cache Addressing

To reverse engineer the LLC-addressing function, it is required to 1) find a way to map physical addresses to slices and 2) find the corresponding function by analyzing tuples of address and slice, which involves some kind of logic solving.

Mapping Addresses to Slices. In previous work, there are two main approaches for mapping physical addresses to slices. The first method relies on an eviction-based strategy [4, 6, 10], similar to the Prime+Probe attack. The first step is to find an eviction set for a specific cache set in each slice of the LLC. Once we have an eviction set for each slice, we can populate the LLC cache set of a specific slice with the corresponding eviction set. Then we access the address to which we want to map the slice. Note that this address has to reside in the same LLC cache set as the evictions set. Then we populate the cache again with our eviction set. If this takes long, we know that some addresses in the eviction set have been replaced, and the target address maps to the same slice as the eviction set. Otherwise, the address maps to a different slice than the current eviction set, and we proceed with the other slices. Another method [5] relies on the usage of hardware performance counters that count accesses to LLC slices. This method is described in detail in Section 4.1, as this is the method we used for our measurements.

Logic Minimization. To reverse engineer the addressing function of the LLC, we also need a solution to obtain a closed form of the addressing function from a mapping table of addresses to slices. As previous work has shown, the addressing function of Intel cores, where the number of cores is a power of two, just consists of XORs of physical address bits [4–6, 9]. While Irazoqui et al. [6] claim that their method should also work for nonlinear functions, this has not been

[1] GitHub: `https://github.com/RUB-SysSec/Microcode/tree/master`
[2] GitHub: `https://github.com/jprx/PacmanOS`

done so far. Yarom et al. [7] were the first to find a compact representation for a non-linear hash function by searching for patterns in the slice mappings. This work was later extended by McCaplin [8] which allowed for the reverse engineering of more non-linear functions. The first unified approach to reverse engineering nonlinear functions was presented by Gerlach et al. [9]. Their approach makes use of the Gröbner basis to minimize the truth table of slice mappings and is explained in Section 4.2.3.

Chapter 6

Discussion

Limitations. Our kernel is the first step to a unified microarchitectural reverse engineering framework. The current research kernel does boot on many Intel machines, but is, at the time of this writing, incompatible with some newer machines. Our kernel did not boot on an Intel i9-12900K and an Intel i9-13900K, both attached to an MSI board. Further, our measurements on the i7-9700K are currently only $99\,\%$ accurate, as the LLC occasionally shows traffic. It is left to future work to investigate this issue. Lastly, there are still some address ranges that are inaccessible for our kernel, as some ranges are reserved by the BIOS. These ranges are usually present in the *PCI memory hole*, which lies between $3\,\mathrm{GB}$ and $4\,\mathrm{GB}$ of RAM. It is left to future work to automatically detect those regions and circumvent this mechanism.

Future Work. Unikernels can be useful in microarchitectural reverse engineering in many settings, not only for reverse engineering cache-slice functions.

- First, our research kernel can be extended to reverse other microarchitectural hash functions, like the DRAM addressing functions, TLB-mapping or cache-way prediction, with high accuracy. The increased performance might be useful for measurement problems where the measurement time rises exponentially, and previous methods run out of time.
- Second, our research kernel can be used for side-channel prototyping, as it provides a highly configurable and noise-free environment. This enables researchers to test potential side channels in an ideal setting before trying to build them on classical systems. E.g., the research kernel can be useful for power-based side-channels as, at testing time, the power consumption of microarchitectural components is only influenced by the measurement process.

- Third, our kernel might be useful for fuzzing model-specific registers. The low boot times of Unikernels (ca. $5\,\mathrm{ms}$) can be useful to recover from complete crashes when accessing MSRs.
- Lastly, our kernel can be used for CPU benchmarking. As we can easily modify our kernel and the code runs with unrestricted privileges, it is possible to monitor CPUs with high detail.

To improve the transferability of our kernel, future work is to adapt our research kernel for a broader spectrum of x86 machines or to other instruction set architectures like ARM. The kernel described in this thesis is the first step towards a unified measurement framework for accurate and efficient low-level measurements, and can be used as a base system for more complex analysis tools. Unikraft is maintained and continuously improved by a large base of developers. These improvements will further extend the capabilities of our research kernel in the future.

Chapter 7

Conclusion

In this thesis, we have shown that unikernels can be used for microarchitectural reverse engineering, surpassing the accuracy and efficiency of previous methods. Due to the unikernel design, the measurement application has complete control over all resources. Furthermore, we can completely remove all environmental noise, as there is no underlying operating system or other processes that can interfere with the measurements.

We illustrated the usefulness and improved accuracy in our case study, in which we reverse engineer linear and non-linear last-level cache addressing functions. We show that our approach can measure cache lookups with single-instruction granularity, and can therefore drastically reduce the number of required measurements. Furthermore, we show that we can reverse engineer a known non-linear addressing function in a much shorter time and recover a previously unknown linear addressing function.

Our research kernel is the first step towards a unified CPU measurement framework and can be used to reverse engineer the microarchitecture of modern CPUs, investigate potential side-channels in a noise-free environment, or benchmark CPU components.

List of Figures

List of Tables

List of Algorithms

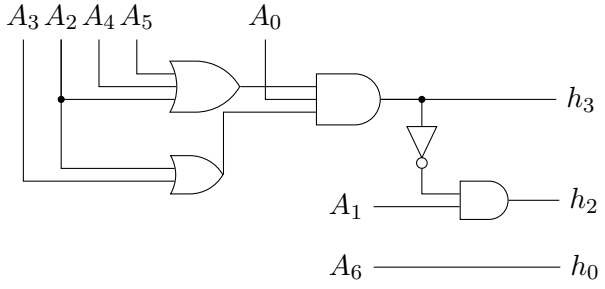
Appendix A

Reported Functions

In this section, we report the full closed-form representation of the reverse engineered hash functions. Figure A.1 shows the previously unknown cache-slice function for an Intel i7-9700K. Figure A.2 shows the logic circuit for the slice-addressing function for an Intel i7-8700K, which was first reverse engineered by Gerlach et al. [9].

$$
\begin{aligned}
h_0 = b_6 \ & \oplus b_{10} \oplus b_{12} \oplus b_{14} \oplus b_{16} \oplus b_{17} \oplus b_{18} \oplus b_{20} \oplus b_{22} \oplus b_{24} \oplus b_{25} \oplus b_{26} \oplus b_{27} \oplus b_{28} \\
& \oplus b_{30} \oplus b_{32} \\
h_1 = b_9 \ & \oplus b_{12} \oplus b_{16} \oplus b_{17} \oplus b_{19} \oplus b_{21} \oplus b_{22} \oplus b_{23} \oplus b_{25} \oplus b_{26} \oplus b_{27} \oplus b_{29} \oplus b_{31} \oplus b_{32} \\
h_2 = b_{10} & \oplus b_{11} \oplus b_{13} \oplus b_{16} \oplus b_{17} \oplus b_{18} \oplus b_{19} \oplus b_{20} \oplus b_{21} \oplus b_{22} \oplus b_{27} \oplus b_{28} \oplus b_{30} \oplus b_{31}
\end{aligned}
$$

Figure A.1: Hash function for 8-core 9th Gen CPU (Intel i7-9700K)



$$
\begin{aligned}
A_0 &= b_6 \oplus b_{11} \oplus b_{12} \oplus b_{16} \oplus b_{18} \oplus b_{21} \oplus b_{22} \oplus b_{23} \oplus b_{24} \oplus b_{26} \oplus b_{30} \oplus b_{31} \\
A_1 &= b_7 \oplus b_{12} \oplus b_{13} \oplus b_{17} \oplus b_{19} \oplus b_{22} \oplus b_{23} \oplus b_{24} \oplus b_{25} \oplus b_{27} \oplus b_{31} \\
A_2 &= b_8 \oplus b_{13} \oplus b_{14} \oplus b_{18} \oplus b_{20} \oplus b_{23} \oplus b_{24} \oplus b_{25} \oplus b_{26} \oplus b_{28} \\
A_3 &= b_9 \oplus b_{14} \oplus b_{15} \oplus b_{19} \oplus b_{21} \oplus b_{24} \oplus b_{25} \oplus b_{26} \oplus b_{27} \oplus b_{29} \\
A_4 &= b_{10} \oplus b_{15} \oplus b_{16} \oplus b_{20} \oplus b_{22} \oplus b_{25} \oplus b_{26} \oplus b_{27} \oplus b_{28} \oplus b_{30} \\
A_5 &= b_{11} \oplus b_{16} \oplus b_{17} \oplus b_{21} \oplus b_{23} \oplus b_{26} \oplus b_{27} \oplus b_{28} \oplus b_{29} \oplus b_{31} \\
A_6 &= b_6 \oplus b_8 \oplus b_9 \oplus b_{10} \oplus b_{14} \oplus b_{15} \oplus b_{17} \oplus b_{18} \oplus b_{20} \oplus b_{23} \oplus b_{27} \oplus b_{30} \oplus b_{31}
\end{aligned}
$$

Figure A.2: Hash function for 6-core 8th Gen CPU (Intel i7-8700)

Bibliography

[1] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[2] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[3] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom *et al.*, "Meltdown: Reading kernel memory from user space," *Communications of the ACM*, vol. 63, no. 6, pp. 46–56, 2020.

[4] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 191–205.

[5] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering intel last-level cache complex addressing using performance counters," in *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings 18*. Springer, 2015, pp. 48–65.

[6] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Systematic reverse engineering of cache slice selection in intel processors," in *2015 Euromicro Conference on Digital System Design*. IEEE, 2015, pp. 629–636.

[7] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the intel last-level cache," *Cryptology ePrint Archive*, 2015.

[8] J. D. McCalpin, "Mapping addresses to l3/cha slices in intel processors," Tech. Rep., 2021.

[9] L. Gerlach, S. Schwarz, N. Faroß, and M. Schwarz, "Efficient and generic microarchitectural hash-function recovery," *S&P*, 2024.

[10] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[11] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 461–472, 2013.

[12] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ş. Teodorescu, C. Răducanu *et al.*, "Unikraft: fast, specialized unikernels the easy way," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 376–394.

[13] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "Includeos: A minimal, resource efficient unikernel for cloud services," in *2015 IEEE 7th international conference on cloud computing technology and science (cloudcom)*. IEEE, 2015, pp. 250–257.

[14] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[15] A. Raza, T. Unger, M. Boyd, E. B. Munson, P. Sohal, U. Drepper, R. Jones, D. B. De Oliveira, L. Woodman, R. Mancuso *et al.*, "Unikernel linux (ukl)," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 590–605.

[16] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Transactions on Computers*, vol. 46, no. 12, pp. 1378–1381, 1997.

[17] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 32–41.

[18] H. Vandierendonck and K. De Bosschere, "Xor-based hash functions," *IEEE Transactions on computers*, vol. 54, no. 7, pp. 800–812, 2005.

[19] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," in *Proceedings of the ninth annual ACM symposium on Theory of computing*, 1977, pp. 106–112.

[20] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C, & 3D): System Programming Guide*. Intel, 2023.

[21] Intel, "Improving real-time performance by utilizing cache allocation technology," *Intel Corporation, April*, 2015.

[22] Y. Yarom and K. Falkner, "{FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack," in *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 719–732.

[23] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[24] A. Abel and J. Reineke, "nanobench: A low-overhead tool for running microbenchmarks on x86 systems," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 34–46.

[25] C. Helm, S. Akiyama, and K. Taura, "Reliable reverse engineering of intel dram addressing using performance counters," in *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2020, pp. 1–8.

[26] A. Kogler, D. Weber, M. Haubenwallner, M. Lipp, D. Gruss, and M. Schwarz, "Finding and exploiting cpu features using msr templating," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1474–1490.

[27] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić, "Make the most out of last level cache in intel processors," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–17.

[28] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. Laurie Rosatone, 2018.

[29] B. Roch, "Monolithic kernel vs. microkernel," *TU Wien*, vol. 1, 2004.

[30] D. R. Engler and M. F. Kaashoek, "Exterminate all operating system abstractions," in *Proceedings 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*. IEEE, 1995, pp. 78–83.

[31] M. Stonebraker, "Operating system support for database management," *Communications of the ACM*, vol. 24, no. 7, pp. 412–418, 1981.

[32] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr, "Exokernel: An operating system architecture for application-level resource management," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 251–266, 1995.

[33] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library os from the top down," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011, pp. 291–304.

[34] L. Soares and M. Stumm, "{FlexSC}: Flexible system call scheduling with {Exception-Less} system calls," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[35] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?" in *2008 IEEE International Symposium on Workload Characterization*. IEEE, 2008, pp. 141–150.

[36] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 215–224.

[37] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z.* Intel, 2023.

[38] B. Falk, "Sushi roll: A cpu research kernel with minimal noise for cycle-by-cycle micro-architectural introspection," August 2019. [Online]. Available: https://gamozolabs.github.io/metrology/2019/08/19/sushi_roll.html

[39] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*. Springer Science & Business Media, 1984, vol. 2.

[40] P. Koppe, B. Kollenda, M. Fyrbiak, C. Kison, R. Gawlik, C. Paar, and T. Holz, "Reverse engineering x86 processor microcode," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1163–1180.

[41] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "Pacman: attacking arm pointer authentication with speculative execution," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 685–698.