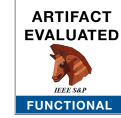

Fractal: An Operating System Designed for Microarchitecture Reverse Engineering

Joseph Ravichandran
MIT CSAIL
jravi@mit.edu

Mengjia Yan
MIT CSAIL
mengjiay@mit.edu

***Abstract*—Modern microarchitecture security research requires a deep understanding of the microarchitecture designs of commodity hardware, specifically how the hardware is isolated between privilege domains. Since these details are usually undocumented, researchers must conduct reverse engineering experiments to learn the microarchitecture parameters of various processors. Currently, the state of the art approach is to modify commodity operating systems in an ad-hoc manner to enable these experiments. Our objective is to systematize the requirements of microarchitecture security research workflows, and to create new lightweight system software precisely tailored to these requirements.**

We present Fractal, a new operating system kernel built from the ground up to enable practical low-noise microarchitecture reverse engineering research at the user to kernel hardware boundary. Fractal enables seamless concurrency between privilege levels and a user-controlled scheduler for fine-grained thread ordering to enable new microarchitecture reverse engineering workflows with minimal noise. We ported Fractal to a variety of real systems and evaluate Fractal by using it to analyze the Apple M1 CPU. We uncovered a number of new insights about the branch predictor on M1, demonstrating for the first time evidence that limited Phantom speculation is present on Apple Silicon.

1. Introduction

Microarchitecture attacks such as Meltdown [1], Spectre [2], Prefetch side channels [3], PACMAN [4], FLOP [5], Inception [6], RETBLEED [7], Phantom [8], Spectre-BHB [9] and others have demonstrated security domains, such as the user/ kernel privilege barrier or address space ID barrier, to be critical boundaries relevant to microarchitecture security research. In response to these attacks, CPU designers have begun implementing hardware defenses to isolate security domains from one another, such as Intel's Enhanced Indirect Branch Restricted Speculation (eIBRS) [10], [11], [12] and ARM's Cache Speculation Variant 2 (CSV2) [9], [13]. These mitigations are intended to prevent code running one security domain from influencing code in another: user code should not interfere with kernel code, and code in one ASID should not interfere with code in another. We use the term *security domain* to represent both the hardware privilege level (user/ kernel) and address space ID (ASID).

The discovery of every one of these microarchitecture attacks began the same way: using the process of microarchitecture reverse engineering to understand the hardware's behavior. A microarchitecture reverse engineering experiment, henceforth referred to as a *µtest*, is a test program designed to measure the hardware to learn key microarchitecture parameters, such as the branch predictor structure and its defense mechanisms. [4], [9], [14], [15].

µtests consist of placing specific carefully chosen assembly instructions, such as branches, loads, or stores, at specific virtual addresses with specific targets, then measuring the latency to perform these instructions using performance counters. These experiments are highly sensitive to subtle changes in all microarchitecture parameters, such as the address space layout, branch history, or cache contents. For instance, if an interrupt occurs while running an experiment measuring the branch predictor, branches taken by the interrupt handler may influence the results of the µtest. In fact, any CPU instructions executed on the same core that are not part of the µtest may influence the results. Researchers must factor this into their experiment designs, typically by repeating measurements many times and using statistics to infer hardware behavior.

Studying isolation mechanisms like eIBRS or CSV2 that introduce isolation between security domains, investigating whether previously unknown isolation mechanisms exist, or reverse engineering how hardware is shared across security domains requires running µtest code on both sides of the isolation barrier.

1.1. Today's Operating Systems

Today, these experiments are normally performed under an existing general-purpose operating system (GPOS) like Linux, or for the case of Apple Silicon, macOS. However, these general-purpose systems present a number of challenges at the system software level that cannot be solved with application code alone:

- On GPOSes, studying user/ kernel hardware defenses necessitates interacting with the driver programming model or patching the kernel. Moving a

µtest from user mode to kernel mode may uncontrollably modify experiment variables such as the address space layout, branch history, and return stack buffer contents.
- µtests have little control over how threads are scheduled or when they are preempted as the scheduling policy is at the sole discretion of the OS. Fine-grained interleaving of user and kernel code or code in different ASIDs within a µtest is challenging.
- GPOSes introduce significant microarchitectural noise due to background processes and kernel threads which cannot be disabled.

Rather than building µtests on top of existing GPOSes, another approach is to build an ad-hoc single-purpose bare metal program that can run directly on a specific piece of hardware to conduct individual experiments. This approach allows greater control over the hardware than a GPOS, but at the cost of requiring new system software for each platform being studied, and the lack of useful abstractions for outputting experiment data or launching tasks. We believe the problem isn't that experiments are performed under an operating system; rather, the problem is we don't have the *right* operating system.

Instead, we aim to characterize the requirements of µtest workloads, identify the role system software must play to enable them, and develop reusable practical system software implementing these requirements. We study the tricks that state of the art microarchitecture security researchers use, develop our own reverse engineering experiments, and iteratively systematize these requirements into a custom lightweight operating system kernel tailored to this workflow. Our solution combines the raw control over hardware of a bare metal program with the ease of use of a GPOS.

1.2. Fractal

FRACTAL is a lightweight custom operating system kernel implementing our proposed set of abstractions for microarchitecture reverse engineering workflows. Fractal borrows concepts from GPOSes where appropriate and introduces new abstractions where required. Like a GPOS, Fractal provides a UNIX-like API, filesystem, drivers for common peripherals, and task abstractions. Like a bare metal program, Fractal provides flexible control over hardware, tailored to the µtest workflow.

Property 1

Fractal allows varying only the security domain while keeping all other microarchitecture parameters constant.

Reverse engineering experiments that measure security domain isolation mechanisms must find some way to host experiment code on both sides of the boundary. If a µtest demonstrates an attack succeeding when all parts of it are run in one security domain, but fails to demonstrate that attack after moving part of the code into a different domain, it is not clear whether this difference is due to hardware defenses blocking the attack, or some parameters of the attack changing due to the act of moving code from one domain to another.

Fractal addresses this by introducing a flexible privilege programming model we call *multi-privilege concurrency*. Multi-privilege concurrency allows a single task to have concurrent threads sharing the same code and memory, but executing in configurable security domains. This allows µtests to vary only the security domain, leaving all other program parameters such as address space layout, branch history, return stack buffer contents, and which cache/ TLB entries are accessed unchanged. If a µtest succeeds when the program is run entirely in one security domain, but fails when part of it is moved to a different security domain, on Fractal it can be stated with a high degree of certainty that hardware partitioning of structures by privilege level is the root cause.

Property 2

Fractal allows expressing exactly which microarchitecture events occur, from what security domain, and in what order.

A common pattern in microarchitecture security attacks is the prime $\rightarrow$ modulate $\rightarrow$ probe pattern. An attacker thread first primes a shared resource, a victim thread utilizes it, then the attacker probes it to see what changed. Coordinating between the attacker and victim can be challenging on existing GPOSes due to their preemptive multitasking model, especially when conducting cross-domain experiments. Preemptive multitasking is also a significant source of asynchronous noise.

Fractal provides primitives for thread ordering and extends them across security domains. µtests can dictate exactly which threads are scheduled when, from which security domain, and in what order. Interrupts or system services never preempt running µtests to avoid introducing noise to experiment data.

Property 3

Fractal introduces first-class support for common µtest memory usage patterns.

µtests frequently require large sparse address mappings, the capability for placing instructions at specifically chosen virtual addresses, and huge pages for more control over physical address bits. On Apple Silicon, these workflows are poorly served by macOS, which places limitations on how memory can be allocated in the virtual address space and lacks support for huge pages. Even compared to Linux, the workflow can be improved by first-class support for a new paging algorithm.

Fractal introduces the `gmap`, which is an approach to virtual memory management specifically tailored to this pattern. The `gmap` provides a mechanism for mirroring entire sparse instruction chains across threads or tasks in dif-

ferent security domains at a consistent location in memory, reducing the need for ad-hoc hacks when creating µtests.

1.3. Example µtest on Fractal

Listing 1 shows a sample µtest on Fractal demonstrating how to launch two threads of configurable privilege level, and how to use the scheduler to interleave them at the sub-function granularity to measure hardware isolation of some structure. The privilege level of either thread can be independently varied, allowing threads to move between security domains without adjusting control flow or any other unrelated microarchitecture parameters.

Listing 1. A Fractal task with two cooperative different privilege threads.

```
fthread_t th_recv, th_send;

void main() {
  // Create a user and kernel thread in this task
  fthread_create(&th_recv, FTHREAD_KERN, recv, NULL);
  fthread_create(&th_send, FTHREAD_USER, send, NULL);
  fthread_switch(th_recv); // Start utest experiment
}

void recv() {
  while(true) {
    setup_uarch_state();
    fthread_switch(th_send); // Switch to send thread
    measure_uarch_state();
  }
}

void send() {
  while(true) {
    modulate_uarch();
    fthread_switch(th_recv); // Switch back to recv
  }
}
```

1.4. Contributions

We make the following contributions:

1) We analyze state of the art microarchitecture reverse engineering approaches on a variety of platforms, identify what requirements these workflows have of system software, and analyze where existing GPOSes fall short.
2) We implement FRACTAL, a lightweight custom operating system kernel implementing new abstractions to address these requirements.
3) We evaluate Fractal by using it to study the Apple M1 branch predictor, discovering novel insights about speculation behavior across privilege levels, including evidence of limited Phantom speculation for the first time.

1.5. Open-Source Release

The Fractal project is open-source. You can find the source code here: https://fractal-os.com.

2. Background

In this section we provide background on CPU microarchitecture, microarchitecture attacks, and OS kernels.

2.1. Microarchitecture Structures

The CPU's microarchitecture refers to all structures and design parameters that are abstracted away by the ISA. This includes structures such as the branch predictor, cache, translation lookaside buffer, and load-store queues [16], [17], [18], [19], [20], [21], [22], [23]. The process of *microarchitecture reverse engineering* is used to learn the implementation details of these structures. This involves executing carefully crafted sequences of instructions that trigger certain behavior, then using performance counters to read metrics from the hardware such as how many cycles a specific instruction took to execute. From this information, the hardware's behavior can be inferred. In this work, we focus on the reverse engineering of branch predictors.

There are two kinds of branch predictor: the indirect branch predictor (IBP) and conditional branch predictor (CBP), used for indirect and conditional direct branches respectively. Branch predictors index their internal storage using the current instruction's program counter (PC) value and the addresses and targets of prior branches from the path history register (PHR) [23], [24]. At prediction time, the current PC and PHR are used to look up the target previously observed for this branch, and this location is speculatively executed. According to prior work [14], [15], the Apple M1 PHR holds information about the most recent 100 branches.

2.2. Microarchitecture Attacks

Microarchitecture attacks such as Spectre [2] and Meltdown [1] exploit these microarchitecture structures via side channels to leak secret information from different security domains, such as userspace learning kernel memory contents. Spectre V2 [2], [9], also known as Branch Target Injection (BTI), is an attack where the attacker can inject chosen targets into the IBP. These injected targets can be later used during predictions made by victim code, allowing the attacker to redirect speculative control flow to a gadget that reveals victim memory contents. The most serious case of BTI is the out-of-place cross-privilege attack, where an attacker in userspace can influence how branches in kernelspace are predicted without triggering the victim branch at all by using branches that alias in the IBP.

To defend against BTI, CPU designers have begun implementing defenses like Intel eIBRS [11] and ARM CSV2 [13] that isolate IBP targets based on the security domain that generated them. CSV2 defines a hardware context to include "security state + exception level + VMID + ASID" [13]. For the purposes of Fractal, we focus on solely understanding the exception level and ASID security domains, and defer other domains for future work.

The Phantom Attack [8] demonstrated that on recent Intel and AMD CPUs, non-branch instructions may be misspeculated as branches, resulting in a transient fetch and decode of the mispredicted branch target. This primitive can be used to reveal the KASLR slide, which is the secret random location where the kernel is loaded in memory [25]. Until now, it was not known whether Phantom is present

on Apple Silicon; in our evaluation, we find that limited Phantom speculation is present on M1.

2.3. Operating Systems

Operating system design has been a research question for decades. Many kernel designs have been proposed, ranging from small microkernel systems [26], [27], [28], [29], [30], [31], to monolithic kernels [32], [33], [34], to exokernels and unikernels [35], [36], [37], [38], [39], [40]. Fractal, filling the role of an operating system except only for microarchitecture reverse engineering workflows, has some common elements with these existing system architectures.

Modern CPUs have several privilege modes. User mode is where normal programs run and is the least privileged, and supervisor or kernel mode is more privileged. The operating system kernel runs in kernel mode and handles CPU resources such as page tables, interrupt and exception handlers, and interacts with hardware such as the interrupt controller. Fractal, being an operating system, also does all these things.

Linux and XNU (the kernel used by Apple's macOS on Apple Silicon devices) are monolithic, making use of a large codebase running in kernel mode [33], [34], [41], [42]. The kernel lives in its own address space separate from user processes. Drivers are extensions that are loaded into the kernel's address space.

Exokernels [36], [37] provide applications with direct access to hardware resources, implementing OS abstractions as needed via libraries in userspace. Fractal shares a design philosophy about resource management with exokernels; namely, that apps should be able to have direct hardware access as needed.

Unikernels [38], [39] and single-purpose bare metal assembly programs bundle OS logic and application logic into a single executable running in kernel mode. Each task on Fractal is given free reign to act like its own unikernel using kernel threads, where Fractal provides common abstractions and multiplexing support.

The Asahi Linux `m1n1` bootloader [43], [44] is a bare metal bootloader for Apple Silicon Macs. Fractal utilizes the USB-VDM UART driver stack and optionally chainloading support from `m1n1` for interacting with the Apple Silicon UART device.

3. Current Approaches

First, let us investigate the role system software currently plays in the microarchitecture reverse engineering process. System software includes the OS kernel, libraries, and background processes unrelated to a given µtest.

3.1. Kernel Patches

If a µtest wishes to study user/ kernel hardware defenses such as eIBRS or CSV2, it needs the capability of running specific instructions (such as branches, loads, or stores) at specific virtual address locations in both user and kernel mode. We call such µtests *cross-privilege* since they involve multiple privilege levels. Since the GPOS kernel controls the kernel address space, inserting µtest instructions into the kernel requires either writing a kernel extension or modifying the kernel itself, usually in an ad-hoc manner. Removing the kernel entirely is not possible as some amount of system software is required to make the hardware run. To investigate Intel eIBRS, Barberis et. al modified the Linux kernel by inserting an additional system call that performs an indirect branch [9], and used userspace code to trigger this branch.

3.2. Kernel Extensions

While the ad-hoc approach of modifying the kernel itself can be sufficient, such an approach is not always possible. On Apple Silicon devices, Linux support is poor (especially for recent SoCs), sometimes necessitating the use of macOS. On macOS, kernel extensions are deprecated [45] and the open-source XNU kernel provided by Apple is unstable due to Apple redacting much of the source code [46]. Instead, the state of the art approach for Apple Silicon devices is to use binary patches to modify the release version of the XNU kernel to configure hardware registers to enable access to the performance counters (which is normally disallowed by macOS) or other adjustments, and to add µtest code via kernel extensions [47]. Patching the core kernel and adding a kernel extension is the approach used by Ravichandran et. al [4], Tuby et. al [14], Jang et. al [48] and Kim et. al [5] when studying Apple Silicon devices. However, when Apple removes support for kernel extensions, this workflow will no longer be possible. Additionally, interacting with kernel extensions requires following opaque extensive paths through kernel code, which limits the ability for µtests to express fine-grained interleaving at the assembly level between user and kernel code.

3.3. Frameworks

Prior attempts at creating reusable microarchitecture reverse engineering frameworks under a GPOS exist. On Apple Silicon, PacmanKit [49] allows experiments to load or execute specific addresses in kernel mode and reports back performance counter measurements. Similarly, the µARF framework for `X86_64` Linux developed by Graf et. al for VMScape [50], [51] allows running arbitrary instructions and measurement routines in kernel mode. These frameworks provide reusable components, but remain limited by the operating system they run on; complete address space mirroring and fine-grained scheduling are still not possible.

Even if creating a kernel extension or modifying the kernel source is possible, certain experiments are rendered impossible by existing GPOS abstractions because kernel extensions and user code cannot mirror all microarchitecture parameters exactly. For example, in a kernel extension, the address space layout will be different than in userspace, the branch history path will be influenced by the system call

handler path which is not present in user code, and the return stack buffer will include kernel pointers instead of user pointers.

Existing work has also identified µtests as being sensitive to background noise. Dai et. al observed that increasing background activity affected their ability to measure the mesh interconnect [52]. Current approaches involve repeating experiments and using statistical methods to manage the effect of noise on data [4], [52], [53]. A µtest running on a GPOS cannot control background activity on its own.

4. Design

Given the reliance on cooperation with system software in the microarchitecture reverse engineering workload, we propose constructing new system software. Unlike prior frameworks, we perform a *clean-slate* design of the entire system from top to bottom. This allows us to create a lightweight portable solution with complete control over the entire software stack. Figure 1 provides an overview of our system. In this section, we will walk through how we arrived at this design. First, we start by understanding the role of the hardware privilege level (user/ kernel) boundary, and later extend our design to address the ASID boundary.

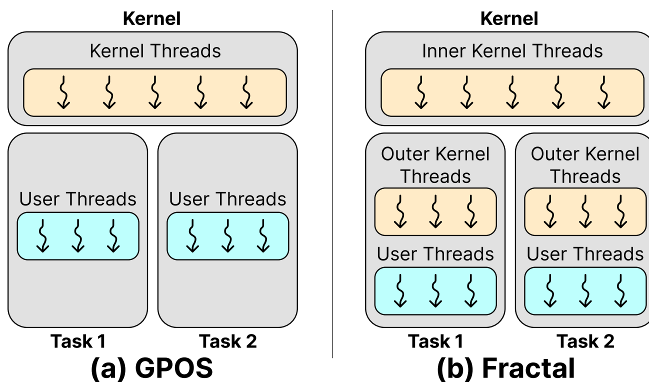

Figure 1. Comparison of System Architectures. a) GPOS, b) Fractal.

We begin with the following insight learned in Section 3 from studying how current microarchitecture reverse engineering is performed:

Insight 1

µtests use kernel interfaces because they are the only way to gain higher privileges on a GPOS.

The only reason µtest authors interact with kernel programming interfaces (either kernel patches or extensions) is to change the hardware privilege level with which their experiment code runs or to disable system protections. µtests do not make use of driver abstractions, kernel workqueues, or other kernel features beyond the bare minimum to execute code with kernel privileges. However, moving between user and kernel mode on a GPOS uncontrollably changes a number of microarchitecture parameters.

Insight 2

To make the privilege level an independent variable, µtests must be able to change it without changing anything else.

In an ideal world, a µtest author would have a single function that could somehow change the privilege level of the current program in-place without requiring a new kernel extension, kernel patches, or changing the address space. This way, experiments that vary only the privilege level and nothing else can be conducted. Our first approach was to create such a function.

4.1. Single-Threaded Privilege Switches

We began by implementing enough of Fractal to boot to userspace, handle system calls, and run C programs. From there, we introduced a system call that allows a user program to change its hardware privilege level at runtime. This allows all of virtual memory, the instructions being executed, and other microarchitecture parameters within the µtest to remain constant while the privilege level changes.

Implementing this requires deep changes to how the kernel manages low-level CPU structures which will be elaborated on in Section 5. Every time a task wishes to switch its privilege mode, the following must happen: 1) The page table must be walked and all permission bits updated with the new privilege level, 2) the TLB must be flushed, 3) the saved privilege register should be adjusted, 4) special system call return paths must be used, as some architectures assume system calls always return to unprivileged mode, 5) special logic is required for handling the kernel stack.

This approach allows user tasks to change their privilege level, but is expensive as it comes with the cost of destroying much of the microarchitecture state. The page table walk uses many branches and loops, and a TLB flush is required every time the privilege level changes, destroying or modifying significant parts of the microarchitecture state. Importantly, there is significant control flow divergence depending on whether or not a µtest varies its privilege level. We would like every instruction executed on the CPU to run in exactly the same order whether or not a privilege switch occurs.

4.2. Multi-Privilege Concurrency

Insight 3

Changing the privilege level is expensive; switching between already running tasks is cheaper.

Instead of using a single task that changes its own privilege mode at runtime, we allow µtests to fork into several tasks, and utilize concurrency to perform the experiment. Each forked task can adjust its privilege level independently before beginning the µtest. Now, instead of adjusting the privilege level during the µtest, the µtest switches between

tasks that already have configured their privilege level, meaning the same instructions are executed regardless of the privilege level, or whether privilege level switches are needed. This task switching should be performed in such a way that recreates the single-threaded control flow. This way, all operations in the µtest are independent of the privilege level:

1) The initial task forks into two tasks.
2) Both tasks configure their privilege levels.
3) The µtest runs, switching between tasks.

However, a different problem emerges: how do we synchronize these tasks so that they run in a deterministic fine-grained order?

4.3. User-Defined Scheduler

Fractal defaults to a GPOS-like preemptive multitasking model, which means tasks can be interrupted and switched at any time. This approach is insufficient for reconstructing a fine-grained interleaving between multiple tasks since tasks have no control over preemption. Switching tasks requires the intervention of the core kernel, as it owns the exception handlers and task data structures, so we require some mechanism to allow µtests to inform the core kernel when and how they should be preempted.

Our solution is to introduce a *cooperative* multitasking model. When cooperative scheduling is enabled, user tasks must explicitly inform the kernel when a task switch should occur. Tasks can specify a default successor task which the kernel will attempt to schedule, or explicitly select which successor task should be run. Cooperative tasks run with interrupts disabled, making this mode helpful for removing background noise due to interrupts as well.

4.4. Multithreading

Both PACMAN [4] and SysBumps [48] performed cross-privilege reverse engineering of the Apple Silicon TLB using kernel extensions. Our current approach of using multiple tasks would not help with TLB reverse engineering, as switching between tasks requires flushing the TLB, destroying the state being measured. Instead, analysis of the TLB requires some mechanism for switching the privilege level without expensive page table walks *and* without TLB flushes. Such a mechanism could also be useful for other µtest experiments beyond the case of the TLB.

To handle this, we introduce support for µtests to use multiple threads within the same task. Threads within a task share an address space, so TLB flushes are not required on thread switches like they are on task switches. However, to support multi-privilege concurrency with multiple threads, a new category of thread is required.

Insight 4

Fractal needs a new category of thread, halfway between traditional kernel and user threads.

On a GPOS, userspace threads always run with user privileges, access the system via system calls, and are mapped in the lower half of memory. Kernel threads run with kernel privileges, access the system through kernel APIs, and are mapped in the higher half of memory. In order to support multi-privilege concurrency, Fractal needs a third category of thread. These new threads will share instruction and data memory with regular user threads, and therefore also use system calls and exist in the lower half of memory, but will run with kernel privileges.

Figure 1 illustrates the difference between multithreading on a GPOS vs Fractal. To clarify the difference between in-kernel and these per-task kernel threads, we define the term *inner kernel thread* to refer to threads that live within the Fractal core kernel, and *outer kernel thread* to refer to these new kernel threads that reside within a task. In general, the term "kernel thread" within the context of Fractal refers to these outer kernel threads. Supporting outer kernel threads requires solving a number of challenges, described in Section 5.

Outer kernel threads share memory with user threads and can be switched without TLB flushes. Due to ISA restrictions, this requires utilizing a shadow memory map within the per-task address space (which will be described in Section 5). Because of this shadow region, some of the upper virtual address bits must differ between user and kernel threads within a task. This is not a problem when using multiple tasks instead of threads, where full 64 bit mirroring *is* possible, at the cost of TLB flushes between task switches.

To summarize: using multitasking (one thread per task, multiple tasks) grants full 64-bit address space mirroring, but requires TLB flushes on context switches; multithreading (many threads in one task) grants mirroring of only some of the virtual address bits, but no TLB flushes are required. This trade-off is why Fractal supports both multitasking and multithreading for µtests; researches can pick which model suits their experiment best, or try both.

4.4.1. Address Space IDs. The address space ID (ASID) is a tag (usually around 8-16 bits) added to the page table to identify a given address space. The ASID is not the same as the page table; two separate page tables may have the same ASID. The hardware may use the ASID alongside the privilege level for partitioning. Like the privilege level, we allow tasks and threads to configure their ASID to investigate this privilege boundary.

5. Implementation

In this section, we discuss the challenges solved to implement Fractal. Fractal has three goals: 1) primitives for expressing multi-privilege concurrency (tasks and threads), 2) a user defined flexible scheduling policy to order tasks and threads, and 3) data structures for common microarchitecture reverse engineering workflows. Fractal supports the `X86_64`, `AARCH64`, and `RISCV-64` ISAs. Fractal has been tested on Qemu [54], a variety of Intel/ AMD

PCs, Raspberry Pi, and various Apple Silicon Macs. Fractal consists of just over 31,000 lines of assembly/C/C++ code, and the core kernel was built from zero.

5.1. Implementation Challenges

Put simply, Fractal uses the hardware in ways it was never designed for. Supporting our features across different architectures is challenging due to differences in virtual memory implementations, memory protection modes, and how traps are implemented by the hardware.

ISAs were never designed to support running the same instruction memory with configurable privilege level. Direct user and kernel code sharing, while possible on `X86_64` by disabling supervisor mode execution prevention (SMEP), is not possible on `AARCH64` or `RISCV-64`, requiring a workaround to enable code sharing without TLB flushes. This approach sometimes introduces exceptions caused by the ISA if one thread uses a function pointer to another thread's memory. Section 5.2 presents our approach of using duplicate shadow memory maps and how to automatically handle these exceptions.

ISAs disagree on how to handle stack switching during traps (eg. an interrupt, exception, or syscall). For normal programs, these differences are not an issue; however, when introducing threads that can change their privilege mode at runtime or multithreaded programs with different privilege levels, management of the CPU stack becomes non-trivial. In Section 5.3, we present a stack aliasing solution that handles all these cases across all ISAs neatly.

In Section 5.4, we outline Fractal's software support for μtests, including scheduler APIs, branchless context switching, and memory management approaches.

5.2. Shadow Memory Maps

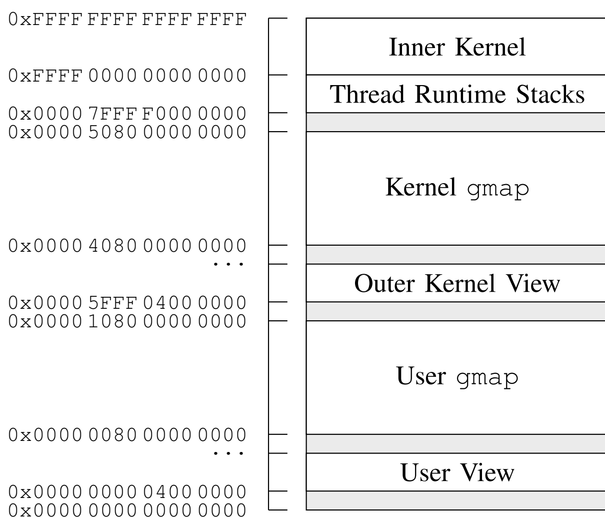

Figure 2. The Memory Layout of a Fractal Task

Fractal allows user and kernel threads in the same task to share memory, including instruction memory. This allows the hardware privilege level to act as an independent variable in a μtest, as the same code can run with either user or kernel privileges. Figure 2 presents a simplified view of memory while running a Fractal task. For security purposes, the ISA restricts user and kernel code from sharing instruction memory, requiring a workaround. The key mechanism enabling multi-privilege code sharing is to map all task memory twice: once for user threads ("User View"), and again as a shadow map for kernel threads ("Outer Kernel View"). Both the user and kernel views of memory refer to the same physical pages, but using different virtual pages and permission bits. Since `gmaps` may have executable code placed in them, each privilege level gets its own `gmap` as well.

Fractal divides virtual memory into two halves: the higher half (inner kernel memory) and lower half (task memory). The higher half is reserved for the core kernel and is never unmapped. The lower half is switched in and out on context switches as it belongs to the current task. Outer kernel threads are mapped in the lower half of memory alongside user memory, rather than in the higher half, as they belong to tasks and not the core kernel.

Sometimes, a thread may try to jump to code in the wrong privilege level, eg. by executing a function pointer, which happens frequently during C library code due to its use of function pointers. Fractal automatically handles these exceptions by detecting attempts to execute memory from the wrong privilege level and quickly fixing the program counter to point to the correct view of memory.

5.3. Stack Aliasing

The introduction of outer kernel threads introduces issues due to ISA management of the stack during traps. When running user code, a trap into the core kernel requires switching the stack from the runtime task stack to a separate interrupt stack to respond to the event. While this approach works for user mode threads, problems emerge when an interrupt or exception occurs while running an outer kernel thread.

Some ISAs such as `X86_64` interact with the stack differently when a trap occurs depending on the hardware privilege level the CPU was previously running in. For user threads, the stack will automatically be switched from the runtime stack to an interrupt stack by the CPU. But for outer kernel threads, this stack switch does not happen, meaning the core kernel will begin servicing the trap using the current runtime stack from task memory. This happens on `AARCH64` as well due to the use of one stack pointer per privilege level. If a trap into the core kernel from an outer kernel thread requires a context switch to a different task, the page table will be switched, removing the current runtime stack from memory, causing the system to crash.

The solution is upon entry to the core kernel, if a kernel thread is running, Fractal moves the stack pointer out of the task's virtual address space to the location where the

stack page exists in the physical memory map. This allows a single physical stack page to be used for running both task code and core kernel code, allows for interrupts to push and pop metadata to the kernel stack asynchronously (as these pushes and pops happen on the runtime virtual location of the stack), and supports context switching as the physical map never goes out of scope during page table switches.

Most importantly, this approach is what allows user threads to promote to kernel mode and kernel threads to demote to user mode. The promotion / demotion process cannot alter the virtual location of the runtime stack because all virtual addresses must remain the same after the mode switch. When promoting a user thread to kernel mode, its runtime stack is used for everything and its interrupt stack is ignored. This works since outer kernel stacks are expected to reside in lower memory. If a kernel thread demotes to user mode, a new interrupt stack is allocated if it does not already exist, and the old runtime stack continues to be used for task code, but now running with user privileges.

5.4. Software Support for µtests

Fractal features common POSIX system calls such as `fork`, `execve`, `read`, `pipe`, and `poll` [55]. In userspace, Fractal's C library is based on newlib, providing support for C library routines like `printf`, `fopen`, and `malloc` [56]. Fractal also ships with ports of many familiar software projects such as GNU `binutils`, `coreutils`, `findutils`, `gcc`, `vim`, and the `dash` shell [57], building a cohesive development experience within the OS. This allows µtests to use familiar POSIX-style APIs and eases the difficulty of migrating existing experiment code to Fractal.

5.4.1. Scheduler APIs. Fractal allows threads to select whether they are scheduled preemptively (interrupts enabled) or cooperatively (interrupts disabled), and to explicitly define the order in which threads should be scheduled. Rather than scheduling based on time deadlines, this scheduling policy is oriented around task completion. Tasks can inform the scheduler of a preferred successor task, which the scheduler will always run after this task if possible. Threads can explicitly switch to threads within the same task using `fthread_switch` which takes the thread ID of the successor. Threads can chain these primitives together to create a fine grained interleaving of experiment control flow based on "what runs when, and in what order."

5.4.2. Branchless Context Switches. When performing a context switch, branches will inevitably be encountered at some point due to the complex logic involved. This is true of both Fractal and GPOSes. If a µtest wants to eliminate all branches from the system, it needs a mechanism for switching the privilege level without branching. Leveraging the ability for threads to switch their privilege mode, Fractal includes an optional scheduling mode which is capable of performing context switches *without* branching.

Branchless context switching works by unmapping the core Fractal trap handlers and replacing them with different ones from within task memory mapped with kernel privileges. Now, every time a trap occurs, rather than trapping into the core Fractal kernel, control flow is transferred to a small stub in the C library that only does one thing: performs a context switch. Since no logic is required, no branch instructions are needed. When this mode is enabled, Fractal becomes unreachable. System calls, interrupts, and exceptions simply trigger context switches, which is why this mode is not the default. This mode can be exited with the use of a library call that restores the original Fractal trap handlers, restoring access to the rest of the kernel.

5.4.3. The gmap. Fractal provides a memory region called the `gmap` which occupies 2^{44} bytes of RWX virtual memory. Every page in `gmap` points to the same physical 2MB huge page. Sections of the `gmap` can be replaced with different unique physical pages (called "carving out"). The `gmap` is designed to minimize memory usage and simplify writing µtests, as every virtual address in the `gmap` is fully controlled.

6. Workflow

We present an overview of the Fractal approach for learning about the user/kernel boundary by automatically transforming simple single-threaded microarchitecture experiments into cross-privilege ones. Figure 3 presents an overview of the Fractal workflow.

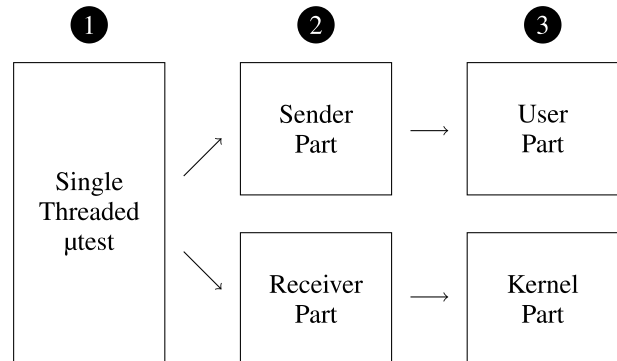

Figure 3. Workflow Overview

First, a simple measurement test should be created where an experiment modulates some hardware structure and measures it. This could mean porting a pre-existing experiment into Fractal or writing a new experiment. For a branch predictor, this could be a method that 1) places a branch at a specific virtual address, 2) trains that branch to jump to a specific address, and 3) uses a side channel to confirm the CPU speculates to that address.

Second, once a reliable simple measurement test for the structure is working, the experiment should be broken up into individual components, such as a sender and receiver. These sections should be either separate threads in the same task or separate tasks. The Fractal scheduler should be used to configure the threads/tasks to run in the correct order

to restore the control flow of the simple single threaded measurement test. When all threads/tasks are ran in the same privilege level, the experiment should return the same results as it did when single threaded.

Finally, the author can vary the hardware privilege level or process ID of the various components of the μtest to learn how the hardware privilege boundary interacts with the experiment. For the example of the branch predictor, if the hardware utilizes privilege-based partitioning, the experiment will succeed if both threads have the same privilege mode, but will fail if they have different privilege modes.

7. Evaluation

We evaluate Fractal by applying it to study the Apple M1 indirect and conditional branch predictors (IBP and CBP). Specifically, we study how the branch predictor state is isolated between privilege levels and ASIDs, and check for evidence of Phantom speculation. Prior work has investigated the conditional branch predictor on Apple Silicon [14], [15], but the indirect branch predictor internals and Phantom speculation have remained unstudied until now. Fractal runs on `X86_64`, `AARCH64`, and `RISCV-64` systems, but we choose to focus on studying `AARCH64` Apple Silicon as this platform has had notoriously poor software support for microarchitecture security research, and therefore is where Fractal offers the biggest improvement to the state of the art.

- We uncover new details about the indirect branch predictor (IBP) on M1, including the existence of user/ kernel partitioning of IBP targets, and discover control of speculative fetches is still possible despite this partitioning.
- We demonstrate the capability of cross-training the conditional branch predictor from userspace to kernelspace on both the performance and efficiency cores, contradicting prior work (Tuby et. al [14]).
- We demonstrate Phantom [8] speculative fetches for the first time on Apple Silicon.

We perform our evaluation on a 2020 M1 Mac Mini 8GB Model J274AP. Fractal is booted directly on the hardware bare metal without any other code running. The Fractal image includes the kernel and a ramdisk which contains our μtest experiment code as a Fractal program. The M1 supports ARM CSV2 as reported by `sysctl` [1].

7.1. Indirect Branch Predictor

We begin by constructing a simple single-threaded μtest that attempts to train and measure the IBP. This μtest: 1) creates an indirect branch (the "dispatch branch") with the `br` instruction, 2) trains it to jump to a specific target, 3) calls that branch with a different target, and 4) uses a side channel to check whether the original trained target was speculatively fetched or executed.

1. Checked with `sysctl -a | grep CSV2` on macOS.

Before training or testing the branch, we perform 128 constant branches to flush the state of the path history register (PHR) to remove the influence of branch history from the experiment. We use a load from the last-level cache (called the SLC on Apple Silicon) to delay the resolution of the dispatch branch target to maximize speculation.

7.1.1. Measuring Speculation. We measure speculative fetches using Flush+Reload [58] of the instruction cache line of the original trained target. If the target instruction line was loaded into the L1 instruction cache during the test run, we know it was speculatively fetched, since this cache line would not be loaded otherwise. We measure this by timing how long it takes to *call* a `ret` instruction in the same cache line as the target using the Apple performance counters.

To measure whether the instruction that was fetched was speculatively executed, we need to make the target perform some action under speculation, then later measure whether that action occurred. We configure the target to attempt to load a data cache line that is flushed to the SLC. Later, we measure how long it takes to *load* this address. If we observe this cache line in the L1 data cache, we know the target was speculatively executed because the load occurred before we measured it, and the only way the load would occur is if the target was speculatively executed.

7.1.2. Introducing Multi-Privilege Concurrency. We configured the experiment to randomly train the indirect branch to jump to one of two targets. Once we had a working single-threaded experiment that could control speculation (both fetch and execute), we broke the experiment up into two cooperative tasks in different address spaces, and used the Fractal scheduler to interleave them to reconstruct the single-threaded order. We utilize multiple tasks rather than threads because the branch predictor utilizes virtual addresses, so full 64-bit mirroring is desirable, and the TLB flush benefits of multithreading are not applicable. Both tasks use a `gmap` to ensure their virtual layouts are identical, but the physical pages are different, allowing us to vary the contents of their memory independently while maintaining full address space mirroring.

The first task is the *sender* task which randomly chooses to either train the branch to jump to the target ("taken"), which is several cache lines away in the `gmap`, or to a `ret` instruction immediately after the branch ("not-taken"). The sender trains the branch 128 times before handing control to the receiver task. The *receiver* task runs its own copy of the branch in its address space and measures whether its copy of the target was speculatively fetched / executed or not. The receiver's branch always jumps to the `ret` directly after it; it never runs the target directly, so the only way the target would be fetched or executed is due to the IBP being influenced by the sender.

Figure 4 provides an overview of the sender and receiver experiments. 'N' represents a not-taken branch, and 'T' represents a taken branch. The receiver's branch is labeled

as either a branch or a `nop` as later we will replace it with a `nop` when testing for Phantom.

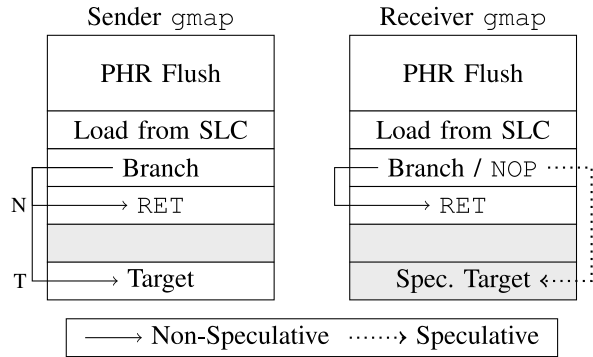

Figure 4. The sender and receiver experiments. T = Taken, N = Not-Taken.

7.1.3. Varying the Privilege Level. We configured the sender to send a sequence of 5000 random branches, either taken or not-taken. We ran the receiver and verified we were able to observe this sequence via speculative fetches and executes when both the sender and receiver were run in user mode. The receiver only performs one measurement per data point; no statistical averaging is used. We performed these experiments on the performance core (p-core).

We then varied the hardware privilege level of the receiver to measure how the hardware implements CSV2 partitioning of the IBP. We hypothesized that a user-mode receiver would be able to retrieve the signal, but a kernel-mode receiver would not. Figure 5 plots the first 160 samples, comparing a user receiver to a kernel one. The ground truth of whether the sender trained the branch to be taken (T) or not-taken (N) is overlaid on top of the samples.

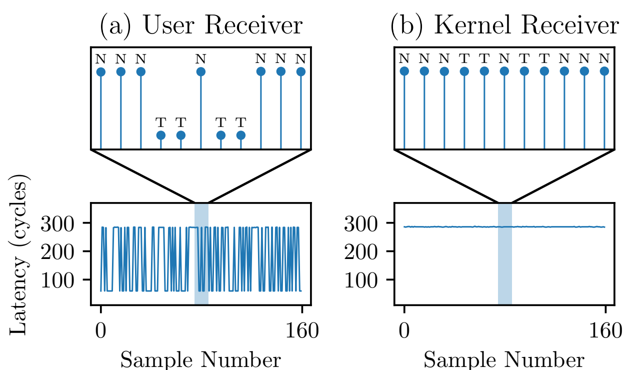

Figure 5. Measuring Speculative Execution, User vs. Kernel Receiver.

Result 1

Targets inserted by user code will not be speculatively executed by the IBP in kernel mode.

In Figure 5, when the receiver is run in user mode, the signal can be clearly observed. When the sender trains the branch to be taken, we observe a L1D latency for the data value loaded by the target, indicating it was speculatively executed. When the sender trains the branch not to be taken, we observe an SLC latency, indicating speculative execution did not occur. However, when the receiver is run in kernel mode, the signal disappears. Every access returns SLC latency, indicating the target was never speculatively executed. This is consistent with CSV2's property that kernel code should not speculate using targets injected by user code.

7.1.4. Exploring IBP Partitioning. From Figure 5, we conclude the IBP is partitioned by privilege level, as the signal disappears when the privilege level changes. However, it is not clear how this partitioning is implemented. The IBP could use a different hash function for indexing its internal structures depending on the privilege level, or it could tag each target with the privilege level that generated it, or something else.

To find out, we repeated the experiment, except measuring speculative fetches instead of executions. Our hypothesis is that speculative fetches will succeed if the receiver is in user mode, and once again fail if the receiver is in kernel mode, assuming the IBP is properly partitioned. Figure 6 presents our findings.

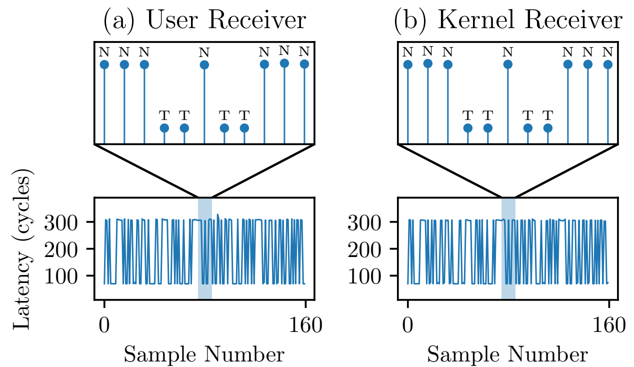

Figure 6. Measuring Speculative Fetches, User vs. Kernel Receiver.

Result 2

Targets inserted by user code can be speculatively *fetched* by the IBP in kernel mode.

Surprisingly, the fetch side channel succeeds in both cases, even for a kernel-mode receiver. We hypothesize there is a race condition in the pipeline; the IBP begins fetching the target before checking whether the target should be executed. Later, the IBP realizes the target was inserted by a different privilege level and prevents speculative execution, but the fetch has already occurred. This also demonstrates that the IBP hash function is independent of the privilege level; user and kernel code must share the same IBP entries, as otherwise speculative fetches would not succeed.

We summarize these four findings in Figure 7 which plots the latency distribution for each of these cases over all 5000 samples. The only case where the attack is blocked is the user to kernel speculative execution case, seen by both taken and not-taken latencies overlapping.

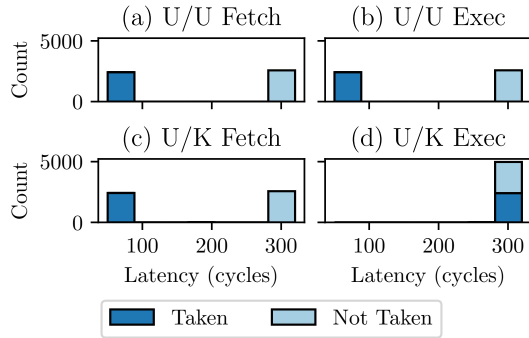

Figure 7. Latency Distributions for A) User to User Fetch, B) User to User Execution, C) User to Kernel Fetch, D) User to Kernel Execution.

7.1.5. ASID Partitioning. From the prior two experiments, we observe that speculative fetch succeeds but execution is blocked when the sender and receiver run with different privilege levels. CSV2 defines the hardware context to include both exception level (ARM's name for privilege level) and ASID. We hypothesize that two user threads with different ASIDs will behave like the user/ kernel case, blocking speculative execution due to CSV2 protections.

Result 3

Targets inserted with one ASID will be fetched, but not executed, by a different ASID.

We repeated the above two experiments on the indirect branch predictor, except set both the sender and receiver to user mode, and configured Fractal to give them each a different ASID value inserted into the `TTBR0_EL1` register. We observed identical behavior to the user to kernel case. That is, when the ASIDs differ, speculative fetches occur, but the fetched instructions were not executed.

7.1.6. Performance and Efficiency Cores. Apple Silicon is a heterogeneous system with both performance and efficiency cores [59]. So far, our experiments have been performed on a performance core (p-core). We repeated our IBP experiments on the efficiency core (e-core) and found no difference in behavior.

7.2. Conditional Branch Predictor

Next we investigate the conditional branch predictor. We modify our experiments from Section 7.1 to use a conditional dispatch branch (`beq`) in place of an indirect one. Like before, a "taken" branch indicates the sender trained the branch to jump to the target. For conditional branch experiments, "not-taken" indicates the sender trained the conditional branch not to be taken, falling through into the `ret` immediately following it. Figure 4 remains accurate for the CBP experiments.

Result 4

The CBP has no protection on M1.

We found that we were able to reliably mistrain the CBP across privilege levels and ASIDs on both the p-core and e-core, including the case of a kernel mode receiver tracking speculative executions. That is, the IBP behavior where cross-privilege execution was blocked is not present on the CBP; we were able to always influence the CBP via both speculative fetches and execution no matter the privilege mode. We conclude there is no privilege isolation of the CBP.

Prior work (Tuby et. al [14]) observed by running code on a macOS system and pinning to a p-core, they were able to cross train the CBP across privilege levels, but were not able to do so when pinned to the e-core. They concluded privilege isolation only exists for the e-core, but not the p-core. Our experimental evidence contradicts their claim, demonstrating that on M1 user code can in fact mistrain the CBP across privilege levels on both the p-core and e-core. We hypothesize this difference to be due to macOS not respecting the core pinning when executing kernel extension code, which would explain why Tuby et. al observed cross-training succeeding on certain cores, but not others [14].

7.3. Phantom

Lastly, we modify our experiments to test for the existence of Phantom speculation. Phantom speculation occurs when a non-branch instruction is speculatively executed as a branch, resulting in speculation that should not have occurred [8]. We modify our experiments by replacing the dispatch branch with a `nop` instruction; the rest of the parameters remain unchanged. Figure 4 remains accurate for Phantom experiments.

Result 5

Phantom fetches occur, but the fetched instructions are not executed.

We find that for both the IBP and CBP on both p-core and e-core, Phantom fetches succeeded in the user to user, user to kernel, and different ASID configurations. However, Phantom execution never succeeded. We conclude that Phantom speculation is present on M1 in the form of Phantom fetches, where user code can cause speculative fetches during kernel code without branches present in the instruction stream. However, these instructions can only be fetched, not speculatively executed.

Configuration		Normal Speculation Simple Fetch	Simple Exec	X-Priv Fetch	X-Priv Exec	X-ASID Fetch	X-ASID Exec	Phantom Speculation Simple Fetch	Simple Exec	X-Priv Fetch	X-Priv Exec	X-ASID Fetch	X-ASID Exec
CBP	P-Core	✔	✔	✔	✔	✔	✔	✔	✘	✔	✘	✔	✘
	E-Core	✔	✔	✔	✔	✔	✔	✔	✘	✔	✘	✔	✘
IBP	P-Core	✔	✔	✔	✘	✔	✘	✔	✘	✔	✘	✔	✘
	E-Core	✔	✔	✔	✘	✔	✘	✔	✘	✔	✘	✔	✘

TABLE 1. SUMMARY OF OUR FINDINGS ON M1. ✔= ATTACK SUCCEEDS, ✘= ATTACK IS BLOCKED BY HARDWARE.

7.4. Summary

Table 1 presents a summary of our findings. A ✔ was placed if the receiver was able to reconstruct the sender's signal and differentiate taken vs not-taken branches by observing L1 vs SLC latencies. A ✘ was placed if the attack was blocked by the hardware and the receiver only observed SLC latencies. "Simple Fetch" and "Simple Exec" refer to the case where both the sender and receiver are in user mode with the same ASID. "X-Priv" refers to experiments where the receiver is in kernel mode rather than user mode. "X-ASID" refers to experiments where the sender and receiver have different ASIDs.

8. Related Work

There have been a variety of systems designed to grant increased privileges to user processes, such as Exokernels, Dune, and other techniques such as BPF filters and DTrace [36], [37], [40], [60], [61], [62], [63]. Exokernels provide applications with direct access to hardware resources, allowing them to implement operating system abstractions as needed via libraries in userspace. Fractal shares a design philosophy about resource management with exokernels; namely, that apps should be able to have direct hardware access as needed. However, Fractal takes a more monolithic approach to delivering these features, allowing user apps to completely override the core kernel, rather than cooperate with it. Dune provides similar primitives to exokernels using virtualization extensions under Linux. Unlike Dune, Fractal runs directly on the bare metal without any other software on the machine, providing complete control over the hardware, eliminating any sources of noise from background tasks, and opening the door to future work which could apply Fractal to studying the virtualization extensions themselves.

Efforts have been made to systematize cross-privilege reverse engineering frameworks on existing GPOSes. In VMScape, Graf et. al [50], [51] constructed µARF, a Linux kernel extension for `X86_64` that enables execution of privileged instructions and measurement utilities. PacmanKit [49] performs similar tasks on Apple Silicon under macOS. However, these tools do not enable full user/ kernel microarchitecture parameter mirroring or fine-grained interleaving of user and kernel code like Fractal does.

The PACMAN project [4], [47] created a number of tools for microarchitecture reverse engineering on Apple Silicon. Pacman Patcher [47] patches the XNU kernel to enable access to the high resolution performance counters, which are normally disabled. On Fractal, these counters are always enabled, as XNU is not involved. PacmanOS [4] is a lightweight single-purpose bare metal execution environment for the Apple M1 CPU. Fractal improves upon PacmanOS by adding full user/ kernel microarchitecture parameter mirroring, among other features.

No other system besides Fractal provides the ability to independently vary the security domain (hardware privilege level and ASID) without changing any other microarchitecture parameters, and only some of them can run bare metal without the assistance of other system software (such as Linux). Table 2 provides an overview of these various systems, comparing their suitability for microarchitecture reverse engineering.

	Provides raw access to hardware	**Boots & runs without any other system software**	**Allows independently varying security domain**
BPF	✘	✘	✘
DTrace	✘	✘	✘
PacmanKit	✔	✘	✘
µARF	✔	✘	✘
Dune	✔*	✘	✘
Exokernels	✔	✔	✘
PacmanOS	✔	✔	✘
Fractal	✔	✔	✔

TABLE 2. COMPARISON BETWEEN OPERATING SYSTEMS.
*PROVIDES VIRTUALIZED HARDWARE, NOT BARE METAL.

Several works have directly studied the Apple M1 CPU by reverse engineering the system-level cache [64] and the branch predictor [14], [15]. These prior works focused on the conditional branch predictor, while we investigate both the conditional and indirect branch predictors. Other works have investigated the branch predictors on non-Apple systems, such as Intel or other ARM chips [65], [66], [67], [68], [69]. Prior work has investigated Phantom speculation on Intel and AMD systems [6], [8].

Spectre V2 [2] introduced the first branch target injection attack demonstrating the capability for userspace code to influence the indirect branch predictor during kernel code execution. Both Spectre-BHB [9] and VMScape [50] discovered new methods of manipulating indirect branch targets across privilege domains in the presence of mitigations like eIBRS [10], [11], [12] or CSV2 [13].

A number of attack papers have evaluated the security of Apple Silicon SoCs, such as PACMAN [4], SLAP [70], FLOP [5], Augury [71], GoFetch [72], SysBumps [48], and Branch Different [73].

9. Future Work

In this work, we evaluated Fractal on the `AARCH64` Apple Silicon platform with a focus on the user to kernel hardware privilege boundary. We selected Apple Silicon for our evaluation as historically this platform has had the least amount of software support of the major commercially available processor families. However, Fractal is supported on a variety of other platforms, including `X86_64` desktop PCs and `RISCV64` CPUs. In the future, Fractal could be used to study behavior on these systems. Additionally, while Fractal focuses on the user to kernel hardware boundary, other hardware boundaries exist, such as kernel to hypervisor or user to hypervisor. Fractal could be extended to support analysis of those boundaries as well.

10. Conclusion

Fractal is a new operating system kernel written from the ground up to address common issues with existing general-purpose operating systems while performing microarchitecture reverse engineering. Fractal improves upon the ability to study the user/kernel hardware privilege barrier by introducing new mechanisms for multi-privilege concurrency. It grants the capability of creating fine-grained interleaving between user and kernel code for precise microarchitecture reversing experiments. It provides abstractions for data access patterns common to microarchitecture workflows, enabling experiments to control larger parts of the address space, and extends this shared control across privilege levels. It delivers these features while maintaining a familiar UNIX-like programming environment and minimizing system noise by never preempting running experiments.

We used Fractal to reverse engineer the cross-privilege level behavior of the Apple M1 conditional and indirect branch predictors, presenting for the first time evidence that Apple Silicon is affected by the fetch variant of Phantom speculation. We demonstrated evidence of user/kernel IBP partitioning that affects the execute stage of speculation, but not the fetch stage. We demonstrated that the CBP is not privilege separated on either the P core or the E core.

We believe Fractal is a strong foundation for a new method of performing microarchitecture research, enabling new kinds of cross-privilege experiments while simultaneously improving accuracy by removing entire classes of noise, allowing researchers to focus on system understanding without fighting the software.

11. Acknowledgements

We would like to thank Apple for discovering and reporting a bug in Fractal to us and helping us fix it.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. (2141064). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

This work was supported in part by the Air Force Office of Scientific Research (AFOSR) under grant FA9550-22-1-0511, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

12. Ethics Considerations

We have disclosed our findings to Apple's product security team.

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[2] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[3] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 368–379. [Online]. Available: https://doi.org/10.1145/2976749.2978356

[4] 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*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3470496.3527429

[5] J. Kim, J. Chuang, D. Genkin, and Y. Yarom, "FLOP: Breaking the Apple M3 CPU via false load output predictions," in *USENIX Security*, 2025.

[6] D. Trujillo, J. Wikner, and K. Razavi, "Inception: Exposing new attack surfaces with training in transient execution," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 7303–7320. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/trujillo

[7] J. Wikner and K. Razavi, "RETBLEED: Arbitrary speculative code execution with return instructions," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3825–3842. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/wikner

[8] J. Wikner, D. Trujillo, and K. Razavi, "Phantom: Exploiting decoder-detectable mispredictions," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 49–61. [Online]. Available: https://doi.org/10.1145/3613424.3614275

[9] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch history injection: On the effectiveness of hardware mitigations against Cross-Privilege spectre-v2 attacks," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 971–988. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/barberis

[10] Intel, "Speculative execution side channel mitigations," 2021. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html

[11] ——, “Branch history injection and intra-mode branch target injection,” 2025. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/branch-history-injection.html

[12] ——, “Branch target injection,” 2018. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/branch-target-injection.html

[13] ARM Limited, “Spectre-BHB: Speculative Target Reuse Attacks.” [Online]. Available: https://developer.arm.com/documentation/102898/0108/?lang=en

[14] A. Tuby and A. Morrison, “Reverse Engineering the Apple M1 Conditional Branch Predictor for Out-of-Place Spectre Mistraining,” 2025. [Online]. Available: https://arxiv.org/abs/2502.10719

[15] J. Chen, P. Qu, and Y. Zhang, “Dissecting Conditional Branch Predictors of Apple Firestorm and Qualcomm Oryon for Software Optimization and Architectural Analysis,” 2024. [Online]. Available: https://arxiv.org/abs/2411.13900

[16] P.-Y. Chang, E. Hao, and Y. N. Patt, “Target prediction for indirect jumps,” in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA ’97. New York, NY, USA: Association for Computing Machinery, 1997, p. 274–283. [Online]. Available: https://doi.org/10.1145/264107.264209

[17] A. Seznec and P. Michaud, “A case for (partially) tagged geometric history length branch prediction,” *Journal of Instruction-level Parallelism - JILP*, vol. 8, 02 2006.

[18] S. McFarling, “Combining Branch Predictors,” Technical Report TN-36, Digital Western Research Laboratory, Tech. Rep., 1993.

[19] T.-Y. Yeh and Y. N. Patt, “Two-level adaptive training branch prediction,” in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, ser. MICRO 24. New York, NY, USA: Association for Computing Machinery, 1991, p. 51–61. [Online]. Available: https://doi.org/10.1145/123465.123475

[20] J. E. Smith, “A study of branch prediction strategies,” in *Proceedings of the 8th Annual Symposium on Computer Architecture*, ser. ISCA ’81. Washington, DC, USA: IEEE Computer Society Press, 1981, p. 135–148.

[21] S. McFarling and J. Hennessy, “Reducing the cost of branches,” *SIGARCH Comput. Archit. News*, vol. 14, no. 2, p. 396–403, May 1986. [Online]. Available: https://doi.org/10.1145/17356.17402

[22] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn, “Virtual program counter (vpc) prediction: Very low cost indirect branch prediction using conditional branch prediction hardware,” *IEEE Trans. Comput.*, vol. 58, no. 9, p. 1153–1170, Sep. 2009. [Online]. Available: https://doi.org/10.1109/TC.2008.227

[23] J. Lee and A. Smith, “Branch prediction strategies and branch target buffer design,” *Computer*, vol. 17, no. 1, pp. 6–22, 1984.

[24] R. Nair, “Dynamic path-based branch correlation,” in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, ser. MICRO 28. Washington, DC, USA: IEEE Computer Society Press, 1995, p. 15–23.

[25] Jake Edge, “Kernel address space layout randomization,” https://lwn.net/Articles/569635/, 2013.

[26] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young, “Mach: A New Kernel Foundation for UNIX Development,” in *Proceedings of the USENIX Summer Conference, Atlanta, GA, USA, June 1986*. USENIX Association, 1986, pp. 93–113.

[27] K. Elphinstone and G. Heiser, “From L3 to seL4 what have we learnt in 20 years of L4 microkernels?” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 133–150. [Online]. Available: https://doi.org/10.1145/2517349.2522720

[28] A. S. Tanenbaum, “A comparison of three microkernels,” *J. Supercomput.*, vol. 9, no. 1–2, p. 7–22, Mar. 1995. [Online]. Available: https://doi.org/10.1007/BF01245395

[29] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, “The performance of μ-kernel-based systems,” in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’97. New York, NY, USA: Association for Computing Machinery, 1997, p. 66–77. [Online]. Available: https://doi.org/10.1145/268998.266660

[30] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “seL4: formal verification of an OS kernel,” in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP ’09. New York, NY, USA: Association for Computing Machinery, 2009, p. 207–220. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[31] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, “Comprehensive formal verification of an OS microkernel,” *ACM Trans. Comput. Syst.*, vol. 32, no. 1, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2560537

[32] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The design and implementation of the 4.3BSD UNIX operating system*. USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[33] L. Torvalds, “Linux: a portable operating system,” *Master’s thesis, University of Helsinki*, 1997.

[34] Apple, “Kernel Programming Guide,” https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/About/About.html.

[35] T. Anderson, “The case for application-specific operating systems,” in *Third Workshop on Workstation Operating Systems*, 1992, pp. 92–94.

[36] D. R. Engler, M. F. Kaashoek, and J. O’Toole, “Exokernel: An Operating System Architecture For Application-Level Resource Management,” *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, p. 251–266, Dec. 1995. [Online]. Available: https://doi.org/10.1145/224057.224076

[37] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceño, R. Hunt, and T. Pinckney, “Fast and flexible application-level networking on exokernel systems,” *ACM Trans. Comput. Syst.*, vol. 20, no. 1, p. 49–83, Feb. 2002. [Online]. Available: https://doi.org/10.1145/505452.505455

[38] 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,” *SIGARCH Comput. Archit. News*, vol. 41, no. 1, p. 461–472, Mar. 2013. [Online]. Available: https://doi.org/10.1145/2490301.2451167

[39] A. Madhavapeddy and D. J. Scott, “Unikernels: Rise of the Virtual Library Operating System: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework?” *Queue*, vol. 11, no. 11, p. 30–44, Dec. 2013. [Online]. Available: https://doi.org/10.1145/2557963.2566628

[40] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, “Dune: safe user-level access to privileged CPU features,” in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’12. USA: USENIX Association, 2012, p. 335–348.

[41] D. M. Ritchie and K. Thompson, “The UNIX time-sharing system,” *Commun. ACM*, vol. 17, no. 7, p. 365–375, Jul. 1974. [Online]. Available: https://doi.org/10.1145/361011.361061

[42] D. M. Ritchie, “The UNIX system: The evolution of the UNIX time-sharing system,” *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1577–1593, 1984.

[43] “Asahi Linux Documentation,” https://asahilinux.org/docs/.

[44] “m1n1: User Guide Boot Loader,” https://asahilinux.org/docs/sw/m1n1-user-guide/.

[45] Apple, "Deprecated kernel extensions and system extension alternatives." [Online]. Available: https://developer.apple.com/support/kernel-extensions/

[46] "darwin-xnu-build." [Online]. Available: https://github.com/blacktop/darwin-xnu-build

[47] J. Ravichandran, "PacmanPatcher." [Online]. Available: https://github.com/jprx/PacmanPatcher

[48] H. Jang, T. Kim, and Y. Shin, "SysBumps: Exploiting speculative execution in system calls for breaking KASLR in macOS for Apple Silicon," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 64–78. [Online]. Available: https://doi.org/10.1145/3658644.3690189

[49] J. Ravichandran, "PacmanKit." [Online]. Available: https://github.com/jprx/PacmanKit

[50] J.-C. Graf, S. Rüegge, A. Hajiabadi, and K. Razavi, "VMSCAPE: Exposing and Exploiting Incomplete Branch Predictor Isolation in Cloud Environments," in *2026 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2026, pp. 865–882. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP63933.2026.00046

[51] ——, "μ-Architecture Reverse Engineering Framework (uARF)." [Online]. Available: https://github.com/comsec-group/vmscape/tree/main/uARF

[52] M. Dai, R. Paccagnella, M. Gomez-Garcia, J. McCalpin, and M. Yan, "Don't mesh around: Side-Channel attacks and mitigations on mesh interconnects," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2857–2874. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/dai

[53] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring(s): Side channel attacks on the cpu on-chip ring interconnect are practical," 2021. [Online]. Available: https://arxiv.org/abs/2103.03443

[54] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 41.

[55] IEEE, "IEEE/Open Group Standard for Information Technology–Portable Operating System Interface (POSIX™) Base Specifications, Issue 8," *IEEE/Open Group Std 1003.1-2024 (Revision of IEEE Std 1003.1-2017)*, pp. 1–4107, 2024.

[56] "Newlib," https://sourceware.org/newlib/, accessed June 2025.

[57] Free Software Foundation, "GNU Software." [Online]. Available: https://www.gnu.org/software/software.html

[58] Y. Yarom and K. Falkner, "Flush+Reload: a high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 719–732.

[59] Apple, "Optimize for Apple Silicon with performance and efficiency cores." [Online]. Available: https://developer.apple.com/news/?id=vk3m204o

[60] G. Heiser, "The seL4 Microkernel An Introduction," https://sel4.systems/About/seL4-whitepaper.pdf.

[61] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility safety and performance in the SPIN operating system," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, p. 267–283, Dec. 1995. [Online]. Available: https://doi.org/10.1145/224057.224077

[62] S. McCanne and V. Jacobson, "The BSD packet filter: a new architecture for user-level packet capture," in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, ser. USENIX'93. USA: USENIX Association, 1993, p. 2.

[63] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '04. USA: USENIX Association, 2004, p. 2.

[64] T. Xu, A. A. Ding, and Y. Fei, "Exam: Exploiting exclusive system-level cache in Apple M-Series SoCs for enhanced cache occupancy attacks," 2025. [Online]. Available: https://arxiv.org/abs/2504.13385

[65] L. Li, H. Yavarzadeh, and D. Tullsen, "Indirector: High-Precision branch target injection attacks exploiting the indirect branch predictor," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 2137–2154. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/li-luyi

[66] H. Yavarzadeh, M. Taram, S. Narayan, D. Stefan, and D. Tullsen, "Half&Half: Demystifying Intel's directional branch predictors for fast, secure partitioned execution," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 1220–1237.

[67] J. Wan, "Branch target buffer reverse engineering on arm," 2024. [Online]. Available: https://arxiv.org/abs/2412.05413

[68] M. Godbolt, "The BTB in contemporary Intel chips," https://xania.org/201602/bpu-part-three.

[69] A. Fog, "The microarchitecture of Intel, AMD, and VIA CPUs." [Online]. Available: https://www.agner.org/optimize/microarchitecture.pdf

[70] J. Kim, D. Genkin, and Y. Yarom, "SLAP: Data speculation attacks via load address prediction on Apple Silicon," in *S&P*, 2025.

[71] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, "Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1491–1505.

[72] B. Chen, Y. Wang, P. Shome, C. W. Fletcher, D. Kohlbrenner, R. Paccagnella, and D. Genkin, "GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers," in *USENIX Security*, 2024.

[73] L. Hetterich and M. Schwarz, "Branch different - spectre attacks on Apple Silicon," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 19th International Conference, DIMVA 2022, Cagliari, Italy, June 29 –July 1, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 116–135. [Online]. Available: https://doi.org/10.1007/978-3-031-09484-2_7

13. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

13.1. Summary

This paper introduces the design of Fractal, an operating system focused on enabling microarchitectural attack (uarch) development. The core motivation of the paper is to streamline the research workflow by removing noise-inducing elements from a general-purpose operating system, and by providing building blocks that simplify experimentation and data collection. The evaluation using Fractal helps confirms a uarch vulnerability within Apple M1 silicon

13.2. Scientific Contributions

- Creates a new tool to enable future science.
- Provides a valuable Step Forward in an Established Field.

13.3. Reasons for Acceptance

1) Creates a new tool to Enable Future Science. The authors build a completely new OS stack that allows uarch researchers to more efficiently investigate uarch CPU behavior.
2) Provides a Valuable Step Forward in an Established Field. The abstractions and architecture can help model future tools to enable research in this domain. This work can also provide building blocks to enable explanatory information on previous and future work in this domain.

13.4. Noteworthy Concerns

1) The current evaluation does not reflect the generality of Fractal to support a diverse category of uarch security research tasks. For instance, can users easily develop utest/exploits using Fractal for Meltdown and Spectre? How do the utest in Fractal differ from the ad-hoc ones? Even better, a user-study could be considered to further strengthen the evaluation.
2) The experimental hardware is rather dated (i.e., Apple M1), and authors should consider updating results to newer models (M2-M5), which were released prior to paper submission.
3) While the findings are useful to confirm previous results and showcase the utility of Fractal, it must be noted that the manuscript only reproduced previously known side-channel attack patterns on new platforms.