Cybersecurity Architecture: The Case for Teaching Computer Hardware and Computer Security Together

Bruce Jacob
United States Naval Academy
Annapolis, MD, USA
bjacob@usna.edu

William Casey
United States Naval Academy
Annapolis, MD, USA
wcasey@usna.edu

Tony Melaragno
United States Naval Academy
Annapolis, MD, USA
melaragn@usna.edu

Abstract

Historically, software engineers and hardware engineers represent disparate groups that don't talk to each other. Because the two don't interact much, neither knows much of the other's discipline, and therefore those they **teach** learn little of the other's discipline. This is problematic because computer design (hardware) and computer security (software) are intimately dependent on each other, and when system designers fail to understand both topics, catastrophe results. For example, critical vulnerabilities like buffer overflows and *Spectre/Meltdown* arose because designers inadvertently bypassed protections that could have ensured robust security. Moreover, teaching security and hardware as separate disciplines obscures the root causes of vulnerabilities, leading to solutions that fail to tackle the fundamental flaws. This paper shows how an understanding of both hardware and security will help one to recognize the dependences and dangerous gaps that exist between the two. Hopefully such awareness can help prevent the future undermining of system security via unintended design consequences.

CCS Concepts

• **Computer systems organization**; • **Security and privacy**;

Keywords

hardware, software, architecture, operating systems, security

ACM Reference Format:
Bruce Jacob, William Casey, and Tony Melaragno. 2025. Cybersecurity Architecture: The Case for Teaching Computer Hardware and Computer Security Together. In *International Symposium on Memory Systems (MemSys '25), October 07–08, 2025, Washington, DC, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3767110.3767135

1 Introduction

Jim Stevens, a Ph.D. candidate at the University of Maryland in the 2010s, focused his dissertation on computer hardware. He interned for several years at the Johns Hopkins Applied Physics Lab, doing R&D in security, and made the following wry observation:

> Hardware people don't know [much] about security,
> and security people don't know [much] about hardware.
>
> — Jim Stevens, U. Maryland Ph.D. student

Traditionally, computer hardware and computer security have been treated as separate domains, and this division has led to several disastrous results. For simplicity and convenience, system designers have undermined hardware and software primitives that would otherwise have ensured robust security. These decisions, likely made without an understanding of the repercussions, have left modern systems extremely vulnerable.

For example, in certain scenarios, modern operating systems grant superuser privileges to software rather than to users, a practice that enables privilege-escalation exploits which would not happen otherwise. Similarly, microprocessors allow user-level applications to generate physical memory addresses directly, paving the way for vulnerabilities like *Meltdown* and *Spectre.* All of these exploits could have been prevented simply by adhering to traditional computing models.

Understanding both hardware and security reveals that many vulnerabilities stem from deliberate design choices that weakened existing protections. This paper focuses on memory-system vulnerabilities, exploring buffer overflows and *Spectre/Meltdown.* It proposes that a re-thinking of how we teach these topics, plus a re-thinking of how the microprocessor handles internal memory structures, can ensure the security of future systems.

2 *Spectre/Meltdown* Is Entirely a Memory-Systems Issue

The Spectre/Meltdown vulnerabilities [5, 11, 12] are not fundamentally speculative-execution problems but instead stem from hardware engineers allowing user applications to generate memory addresses that do not belong to them. The protection otherwise guaranteed by virtual memory was inadvertently undermined to simplify I/O hardware and kernel operations.

2.1 Virtual Memory and I/O

Virtual memory, developed in the 1960s, automates data movement between main memory and disk, mapping virtual addresses to physical memory at the granularity of pages [2, 7, 8]. As shown in Figure 1, processes A, B, and C have virtual address spaces translated by the operating system and hardware into physical addresses. This enables sharing (e.g., shared libraries) while enforcing isolation.

Currently, the operating system leverages virtual memory to regulate software's access to physical resources. Applications, unless executing in privileged mode, cannot directly interact with physical memory or I/O devices. All memory operations, such as instruction fetch or load/store instructions, undergo a translation process controlled by the operating system, which restricts access to certain memory regions based on program or user identity. This enables the kernel to shield itself from user-level applications and to isolate user applications from one another.

This work is licensed under a Creative Commons Attribution 4.0 International License.
MemSys '25, Washington, DC, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2002-4/25/10
https://doi.org/10.1145/3767110.3767135

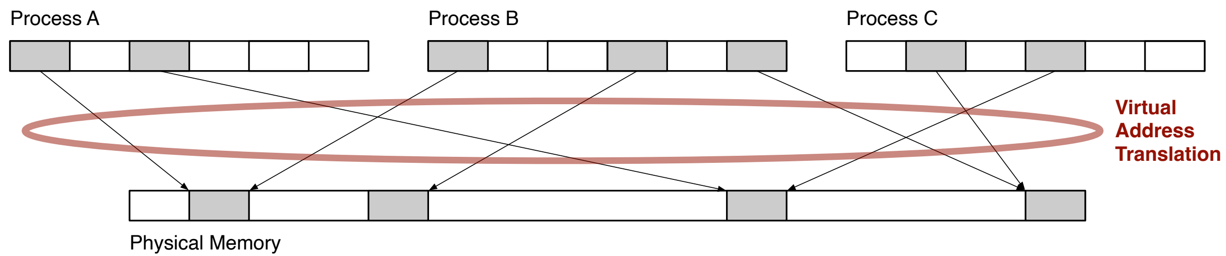


Figure 1: The general concept of virtual memory: translation from a virtual space onto a physical space

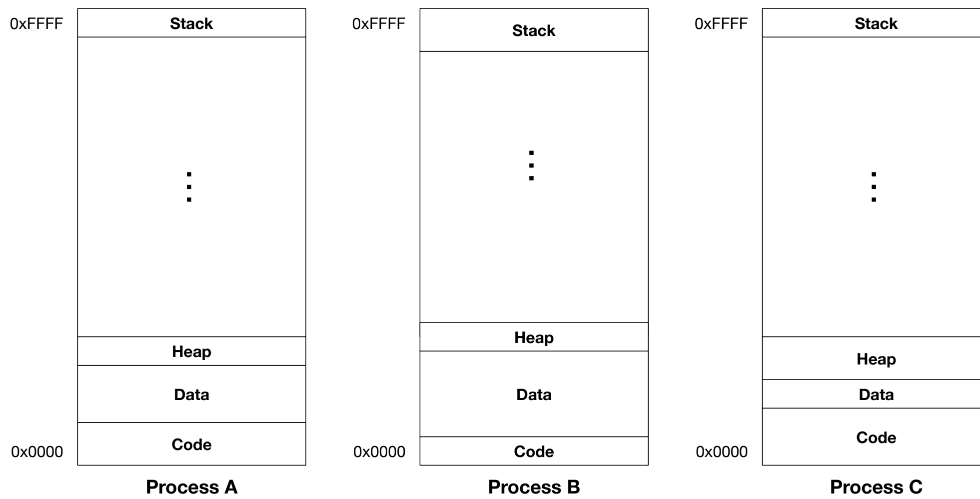


Figure 2: The VM theory: processes execute in virtual address spaces ranging from zero to MAX

In addition, modern systems employ memory-mapped I/O, in which access to hardware controllers for peripherals (e.g., disks, keyboards, monitors, networks, printers) is routed through the memory system using regular load/store instructions, as opposed to the specialized I/O instructions that were used for this many decades ago. Consequently, the same virtual memory protections that prevent unauthorized access by faulty or malicious user applications also secure the I/O subsystem.

The diagram in Figure 2 shows the same user-level processes as before (A, B, and C), just in more detail. Each has its own virtual environment and address space, ranging from 0x0000 to 0xFFFF (or whatever the maximum integer value is). Software uses numeric addresses to reference locations in its address space, and these numeric addresses are allowed to range over the entire space that an integer can represent, from zero to the largest integer value. These addresses only have meaning within the process's virtual address space and must be translated into physical addresses before they can be used for accessing memory or I/O.

2.2 Meltdown Vulnerabilities

Here is the problem: modern computer systems break this model. Instead of treating all numbers in the range [0 .. max] as virtual addresses to be translated, modern hardware and software, for purposes of convenience, divide the address space into separate **user** and **kernel** regions, as shown in Figure 3.

Addresses with the most significant bit set to '1' are reserved for the kernel, while addresses in the positive-integer range (from 0x0000 to 0x7FFF) are given to user applications. All modern processors do this, in some form; and they designate different ranges within the kernel region to have different side-effects, such as being cacheable, or non-cacheable, or virtual (translated through the virtual memory system), or non-virtual (mapped directly to physical memory), etc. This design, intended to enable memory-mapped I/O and to simplify kernel *copyin/copyout* operations, exposes kernel addresses to user applications, thereby undermining security. The following discussion explains.

As shown in Figure 2 (the "VM theory" illustration), user applications operate solely with virtual addresses, meaning that they are wholly unaware of physical memory. To an application, memory addresses outside its address space simply do not exist. However, in the subsequent diagram (Figure 3, the "VM reality" illustration), we see that, in reality, kernel addresses are exposed to user applications, directly. In modern systems, a user application **can** express an address representing something outside of its address space.

This shift severely undermines security and is the primary cause of the *Meltdown* vulnerability. Previously, a user process could not name anything outside of its individual virtual address space, and

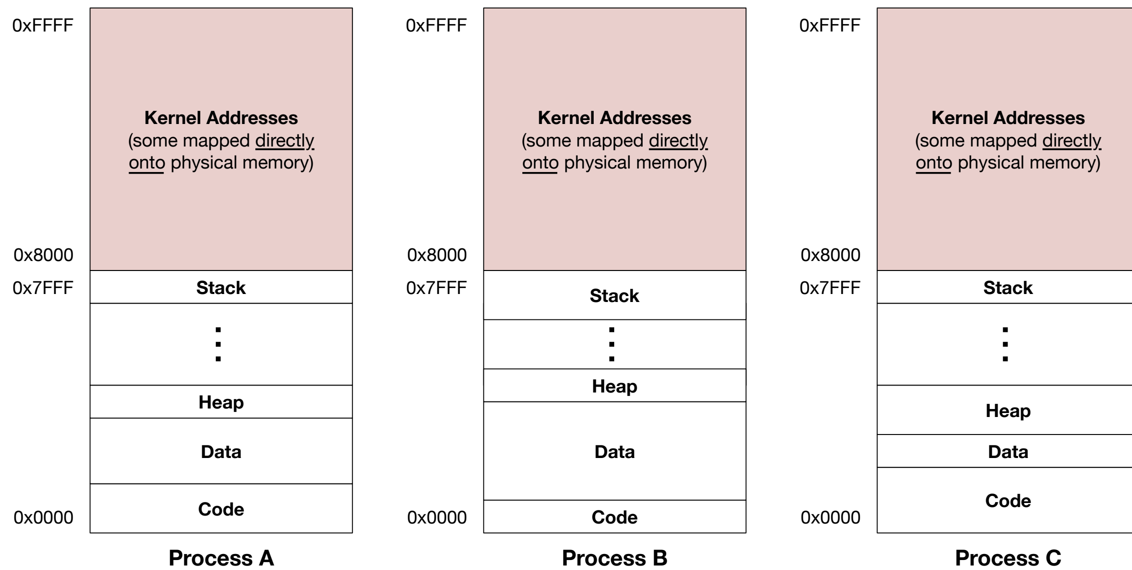


Figure 3: The VM reality: user processes have direct access to kernel addresses; they just aren't supposed to use them

this arrangement ensured the isolation of user applications from each other, the kernel, and all of I/O space. By revealing kernel addresses to user processes, our microprocessors now allow user applications to name physical memory directly, thereby eroding this isolation. The only remaining safeguard is the microprocessor's privilege check, which Meltdown bypasses using speculative execution to violate processor rules undetected.

The pseudocode below illustrates how Meltdown works.

```
char array[256 * CACHE_BLOCK];
for (PA=0x8000; PA<=0xFFFF; PA++) {
    flush cache
    if (0) {
        load-byte-via-phys-addr    r1 <- mem[PA]
        mul               r1 <- r1 * CACHE_BLOCK
        load-byte          r0 <- mem[r1 + array]
    }
    for (i=idx=0; i<256; i++, idx+=CACHE_BLOCK) {
        t0 = now();
        x = array[idx];
        t1 = now();
        if t1-t0 is low, we know that mem[PA]==i
    }
}
```

For each targeted physical address PA, the cache is flushed, a byte value at the physical address PA is read (**note:** *this is an illegal operation*), and the retrieved value is scaled by the cache block size and used to index into an array. The scaling ensures that each array index corresponds to a unique cache block.

After the second **load-byte** finishes, the array is swept through with timing checks. Since the cache was flushed beforehand, most array accesses are slow (cache misses), but one element — say, the nth — will be retrieved quickly, meaning that it will have been cached. It is this n^{th} element of the array that was brought into the cache by the second of the two **load-byte** operations. The value n is therefore the byte value that the first (illegal) **load-byte** instruction retrieved, which means that n is the byte value found at the physical address PA. This whole process is done for each physical address PA the attacker wishes to read.

Key questions arise. For starters: What is the purpose of the **if (0)** statement? Modern microprocessors overlook illegal operations if they are executed unintentionally. The code within the **if (0)** block isn't meant to execute, so if it runs due to a mispredicted branch, the application isn't penalized. Assuming that one can trick the branch predictor (and one certainly can … see for example [5, 11, 12]), then one can cause the processor to execute the illegal load instruction and the two instructions after it speculatively. When the processor later detects the misprediction, it discards the speculative instructions, but not before they impact the cache.

Another question: How is the physical address's value retrieved if such access is prohibited? Indeed, speculative instructions are not allowed to alter architectural state (e.g., registers, memory, or I/O), and hardware enforces this. The answer is that the value from the first load isn't retrieved directly but rather *indirectly*; it is passed to the second load within the speculative block, which uses it as an index into the array, thereby modifying the corresponding cache block. Crucially, the cache — as it is considered to be **non**-architectural state — *can* be modified by speculative instructions. The final **for()** loop uses side-channel analysis, measuring access times, to deduce what the loaded value was. The process indirectly extracts data that would be inaccessible through direct means.

Note that all of this hinges on one deeply problematic instruction being allowed in the first place:

```
load-byte-via-phys-addr    r1 <- mem[PA]
```

Why does this instruction even exist? Why are user applications *ever* allowed to generate physical addresses directly?

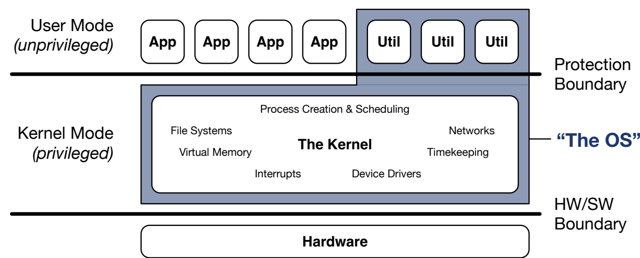


Figure 4: The (theoretical) Operating System comprises the kernel and a collection of user-level utilities

As noted earlier, this design decision was made to simplify system implementation. Unfortunately, this decision also introduced a massive security vulnerability. In a system where user applications can only generate virtual addresses, it is *guaranteed* that user applications cannot access memory outside their sandboxed space. Exposing foreign (e.g., kernel and unavailable) addresses to user applications may be convenient, but doing so obviously breaks this otherwise airtight security model. Allowing applications to generate physical addresses was a shortsighted decision, likely made due to a lack of cross-disciplinary knowledge between hardware designers and security experts. Meltdown exploited this fundamental flaw, which could have been avoided by maintaining strict virtual memory boundaries.

In other words, Meltdown only happened because we opened the door for it, by undermining virtual memory.

2.3 Spectre Vulnerabilities

Spectre similarly exploits speculative execution but targets memory addresses outside software-defined sandboxes, as opposed to the hardware-enforced sandbox of physical memory. Spectre exploits the fact that hardware cannot differentiate between the virtual addresses generated by different threads in the same address space.

Today, we use a hardware *address-space identifier (ASID)* to tag virtual addresses with a process ID and thereby isolate different process address spaces from each other. This also protects the operating system from user processes.

A natural extension of this concept would be to include a Thread ID within the ASID. This would allow a server to run third-party threads safely within its own address space, each thread limited to accessing only the addresses within its hardware-enforced sandbox. Such a mechanism would enable the server to protect itself against subservient threads within its address space, just as ASIDs enable the kernel to protect itself against processes running on the same machine, within the same physical memory system as the kernel. This could be done with new processor designs but could also be implemented entirely within the kernel, without requiring new hardware, as one could map multiple ASIDs to the same PID.

3 Buffer Overflows Are Not a Memory-Systems Issue

In a similar vein, while buffer overflows are presented as a memory-systems issue, with a memory-systems solution, this is not the case. Buffer-overflow exploits, including stack smashing [13] and

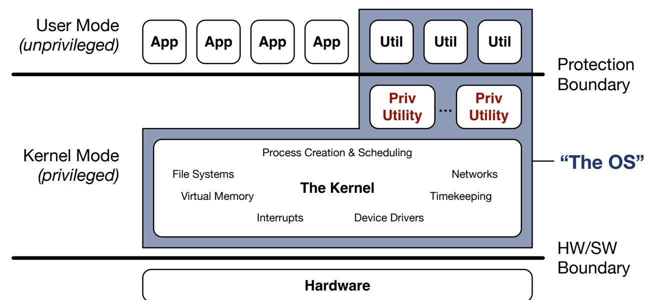


Figure 5: The Operating System comprises the kernel and a collection of user-level utilities, some of which are privileged

return-oriented programming [14], are the result of allowing certain applications to run with superuser privileges regardless of what user account invokes the application.

3.1 The Operating System vs. the REAL Operating System

Traditionally, the general organization of a system is presented as shown in Figure 4. All user-level applications run in unprivileged mode, even operating system utilities such as mail, messaging, the shell, windowing system, etc. Keeping all but the kernel out of privileged mode is one of the critical ways in which the operating system maintains security and protects itself from user applications.

This presented organization turns out not to be true. In reality, as implemented, all modern operating systems (but VMS, as far as we can tell) actually do what is shown in Figure 5. Modern operating systems have utilities that run in privileged mode, irrespective of the user account that starts up the application. This means that one need not be an admin to give these programs superuser privileges. Privileged utilities include many servers and administrative programs. One of the most well-known is the Linux CUPS facility (Common UNIX Printing System). As we will see, and as should be intuitively obvious, the existence of these privileged-level utilities causes significant security problems.

3.2 Stack Smashing

It is widely known that if we can deliberately trigger an application that uses *unsafe* system calls, we could gain control over the entire system. But how does it happen? What exactly are "unsafe" system calls, and why does this lead to compromising the *whole machine* rather than just the flawed application?

An unsafe system call typically refers to one that fails to perform proper checks, such as bounds checking on buffer sizes. Software developers, including those working on operating system code, sometimes neglect to verify a buffer's size before writing to it. This oversight makes the software susceptible to buffer-overflow attacks.

In a typical call stack, a function's local variables reside at lower memory addresses than the function's return address. Thus, if a program writes more data into a buffer than the buffer can hold (a *buffer overflow*), the extra data that extends beyond the end of the buffer can overwrite the function's return address. When the

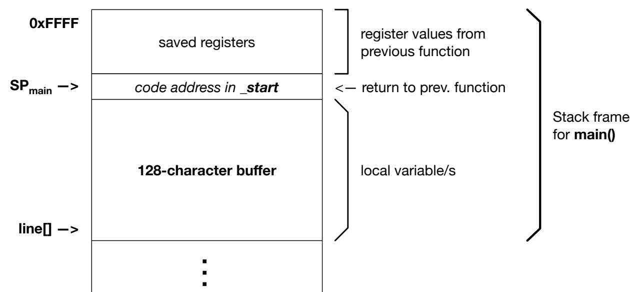


Figure 6: Stack frame in the code example below

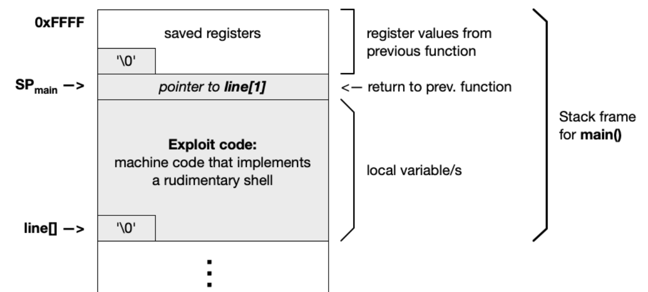


Figure 7: Stack frame in the code example, after the data block (highlighted in grey) has been read in

function returns, the program then jumps to this new, attacker-controlled address instead of the one it was supposed to use.

Consider the following code:

```
void
getline(char *buf, int max)
{
    int c;
    while ((c = getchar()) != '\n') {
        *buf++ = c;
    }
    *buf = '\0';
}

void
main()
{
    char line[128];
    while (1) {
        getline(line, 128);
        if (process(line) == DONE) {
            return;
        }
    }
}
```

This is a simple program that takes in user-provided data one line at a time, using the **getline()** function. The **main()** function processes each received input line via a non-specific **process()** function. The program ends when the **process()** function returns DONE, at which point **main()** returns.

The program includes a buffer-overflow vulnerability in it. Although **getline()** receives a max value representing the maximum length of the input buffer, the function never uses it. If too much input is received, the buffer given to the function overflows.

The stack frame looks something like what is shown in Figure 6. Since the buffer **line[128]** is a local variable of the **main()** function, it is in the function's stack frame immediately below the return address. A buffer overflow would therefore overwrite the return address in **main()**'s stack frame. Under normal circumstances, **main()** would return to **_start**, which is the routine that initialized the program, set up the stack, and then called **main()**. But with the return address overwritten, the program will instead jump to an attacker-specified location — such as within the input buffer itself.

Let us suppose that the **process()** function will return DONE if the first item in the input array is a NULL character. We can thus imagine providing as input to this application a carefully crafted data block that looks like the following:

```
0:       '\ 0'
1:       # machine code for a rudimentary shell
         #
         # ...
         #
127:     # last (valid) entry in the input buffer
128-131: 4-byte pointer to line[1]
132:     '\ n'
```

The newline character at location 132 in the input block will cause the **getline()** function to stop collecting input and return to **main()**. When **getline()** returns, the stack will look like the illustration in Figure 7. Note the overwritten return address at SP_{main}.

At this point, **main()** then calls **process()**. The leading '\0' in the buffer causes **process()** to return DONE, and **main()** exits via an explicit **return**. However, the return address on **main()**'s stack has been overwritten to point to **line[1]**, where the attacker's shellcode resides. Thus, instead of returning to the **_start** routine, **main()** jumps into attacker-controlled code that we are told implements a rudimentary shell. The application has now effectively been hijacked to become a shell process — a classic buffer-overflow exploit.

Important question: This example explains how an attacker can hijack a specific application, but why does it result in system-wide compromise?

The answer lies in the earlier design decision to let some applications run with kernel-level privileges. ***These*** are the applications that are targeted by attackers — because if an attacker can take over an application that runs with the same privilege level as the kernel, then they've effectively taken over the entire machine.

Just like Meltdown and Spectre, this class of vulnerabilities exists because we opened the door for it. The choice to blur boundaries between user and kernel privileges created an attack surface that should never have existed.

3.3 Return-Oriented Programming (ROP)

How can we address the issue above? Several approaches exist:

- **Make the stack non-executable**. The exploit relies on the microprocessor executing code placed on the stack, which

is meant for automatic variables, not code. Many operating systems historically permitted executable stacks, likely for practical reasons, but prioritizing system security suggests this should be disallowed.

- **Enforce array bounds checking**. Certain programming languages inherently perform bounds checks on array accesses, ensuring indices stay within defined limits, thus preventing buffer overruns.
- **Use canary guards**. Cowan proposed a software canary guard in 1998 [1]. The idea was to probe if stack memory had been overwritten by checking if an extra value stored on the stack was still there.
- **Use a shadow stack**. A shadow stack is a secondary stack that "shadows" the main stack (all push/pop operations are applied to both) and therefore allows for verifying the main stack's integrity [17].

The first approach — making the stack non-executable — is known as W^X ("write-xor-ex") and requires the operating system to ensure memory regions are either *writable* or *executable*, but not both. While this does block the specific stack-smashing exploit described above, it falls short in two significant ways.

One prominent way to bypass W^X is through *return-oriented programming (ROP)* [14]. In ROP, instead of placing executable code on the stack, attackers overwrite the stack with addresses. These addresses point to existing code snippets, called *gadgets*, that lie within the program. The code snippets were discovered by the attacker beforehand, by searching for and cataloging blocks of instructions in the executable file that end in **ret** instructions.

Each gadget executes a small task and ends with a **ret** (function-return) instruction. Intel x86 and Arm architectures use a **ret** instruction that pops the top stack value, increments the stack pointer, and jumps to the popped address. This allows one to "chain" multiple invocations of different gadgets, thereby enabling essentially arbitrary software operation.

Imagine that the following shows the contents of a small region of the stack:

```
.
.
0xCDEF
0x90AB
0x5678
0x1234 <- sp
.
.
```

Here, higher addresses are upward; the stack pointer moves downward when pushing and upward when popping. On Intel x86 or Arm, executing a **ret** instruction pops 0x1234, increments the stack pointer to point to 0x5678, and jumps to address 0x1234. If that address contains a gadget ending in another **ret**, the next value (0x5678) is popped, and execution jumps to address 0x5678, continuing the chain. Attackers set this up in the same way as our previous buffer-overflow attack: by overwriting the buffer with a sequence of gadget addresses and altering the function's return address to initiate the chain.

Thus, W^X fails to fully resolve the issue, due to ROP. Additionally, the root of the problem isn't really buffer overflows but the broader issue of user-level applications running with kernel-level privileges. As long as this opportunity exists, attackers will find ways to bypass any protective security measures.

The second approach — array bounds checking — effectively prevents buffer-overflow attacks by blocking writes beyond buffer limits. However, despite its security benefits, the (perceived) performance overhead has discouraged widespread adoption.

The third approach — the use of canary guards — has been integrated into the Gnu C compiler. GCC adopted the stack canary concept in 1998, and it matured into a standard feature by 2003. In general, the check occurs when the function returns; if the canary value has been altered, it may be due to stack smashing. The idea was also a move toward proactive security enhancements for software engineering, differing from the earlier model of patching issues as they are discovered in binaries.

However, as canaries are both heuristic and probabilistic, researchers have looked for more precise and deterministic mechanisms. One such is the fourth approach listed above: the shadow stack, proposed by Szekeres [17]. This is a separate stack that is operated in sync with the real stack and checked for consistency. Linux support for hardware-enforced shadow stacks was merged into the kernel in version 6.6 in October 2023.

Like automatic array-bounds checking, canary guards and shadow stacks both require extra overhead; shadow stacks require extra memory overhead as well as CPU overhead.

4 Anything that Can Be Modified Is a *de facto* Memory Device

Proposals have been made to solve the side-channel problem by preventing timing probes of the cache and/or having a separate kernel cache. These fail, however. Given the lens of security, we can now view all of the components in our system as potential memory devices. Anything that can be modified and then later activated, directly or indirectly, is capable of storing information, however fleetingly. The entire computer system is thus a memory device when viewed through this lens. The following are a few examples.

4.1 Branch Predictors, Prefetch Buffers, etc., Can All Be Modified and Read

As other papers have shown, one can fool branch predictors by training them to predict a specific path. Therefore, writing data consists of training the predictor on a particular branch instruction, and reading the data is done through a simple *if-then-else* statement based on that branch.

This is easily extended to include prefetch buffers, data predictors, etc.

Note that this type of adversarial behavior — i.e., repeated reliable miss training of the branch predictor, or training of the data prefetcher, etc. — connects to "adversarial machine learning," a topic in data-science curricula [6].

4.2 Memory Locations and their Virtual-Memory Mappings

Virtual memory's *translation look aside buffer (TLB)* can also be used like the branch predictor above to store data. As an example,

one could encode 1024 bits in the state held by the OS's virtual memory system using the following approach:

```
char *ptr = malloc( 1024 * PAGE_SIZE );
for (i=0; i<1024; i++) {
    data = get next bit of data to store
    if (data == 1) {
        ptr [ i*PAGE_SIZE ] = random();
    }
}
```

In most operating systems, when one creates large regions of memory through calls to **malloc()**, those regions are not immediately allocated and mapped by the kernel; allocation only happens on demand, when a given page is actually *used.* In the mean time, those pages exist in name only. Therefore, at the start of the **for()** loop in the code above, all of the **malloc()**'ed pages will be unallocated and unmapped. This means that any access to any of the pages will take a long time: each initial reference to one of these pages-in-name-only will generate a PAGE FAULT, causing the operating system to allocate a physical page and map it into the process's page table.

Once the code snippet above is finished, one can simply go through the allocated array to see which pages have a fast access time because they have been allocated and mapped (and therefore represent a value of '1'), versus those that have a slow access time and represent a value of '0'. Note that this is independent of the cache; it will work whether one has a cache or not, as it depends on the operating system's virtual memory system. Also, for obvious reasons, only **malloc()** would work here; a **calloc()** would defeat the process, as it initializes the entire array before returning.

4.3 Flash Cells Wear Out Quickly

Flash cells can only be erased and rewritten about 1000 times before they no longer reliably hold a value. Thus, if wants to encode a ROM message in a particular flash page, one can intentionally burn out individual cells and leave others alone and thereby indicate '0' and '1' bit values.

When a logical page is rewritten, the corresponding physical flash page gets moved onto the free list and therefore can no longer be read directly by normal flash drivers. This effectively hides the page from normal hardware.

Custom hardware or something like Open Channel SSD (see, for example, lightnvm.io) would allow one to read and write the flash devices in this way.

5 Conclusion #1: We Need To Teach Hardware and Security Together

Understanding the relationship between computer security and computer hardware (the memory system, and the field of computer architecture in general) is critical.

- We were told that buffer-overflow attacks were all about the memory system. They are not. At their root, they are enabled by allowing user-level applications to run as superuser.
- We were told that Spectre and Meltdown were all about speculative execution. They are not. At its root, Meltdown is enabled by an intentional modification (and the resulting **un**intentional weakening) of the virtual memory system. Spectre exploits the fact that hardware cannot distinguish between the memory addresses generated by different threads operating in the same address space.
- Proposals to limit the interaction of the kernel and user code through the cache system and/or memory system are shortsighted, as nearly every component in a modern CPU can be used as a memory device. Solving the *cache-as-a-side-channel* problem doesn't solve the general *side-channel* problem.

These cognitive disconnects happened because researchers in the two domains, hardware and security, do not communicate, their domains are siloed, and students are not taught both topics together. Thus, the intersection between the two domains represents a significant amount of cross-disciplinary material that is not well understood by many people, yet represents an extremely significant potential for system vulnerabilities.

This situation is far from ideal.

The obvious solution is to teach these two topics together, to ensure that the intersection between the two domains is well understood by both hardware designers and security experts. In other words, hardware students should be taught security, and security students should be taught hardware ... and the two topics should be taught together instead of separately. Doing this will eliminate vulnerabilities that slip through the cracks between the two disciplines. As mentioned, these types of vulnerabilities are the most significant ones we know of today, so addressing this simply, through education, would be easy to do and would also represent a tremendous improvement to the status quo.

6 Conclusion #2: We Need a New Definition for 'Architectural State'

Teaching the topics of hardware and security together will solve many future problems. In the mean time, however, there are existing problems that need to be addressed. As described earlier, side-channel exploits are very powerful and can turn any structure in the system that holds state into a *de facto* memory device that provides indirect communications between speculative code and non-speculative code. We believe that the simplest solution to this problem is to include these stateful structures in the definition of Architectural State, so that they are included in both correctness proofs and security models.

6.1 Redefine Architectural State to Include All Shared Structures

The security issues we face stem from the divergence between microarchitecture and architectural state, a problem highlighted by researchers when Spectre and Meltdown first emerged [4]. Over decades, computer architects have developed increasingly sophisticated microarchitectures to boost performance, incorporating features like caches, pipelines, superscalar and out-of-order execution, branch prediction, speculative execution, data prefetching, multi-core designs, and multi-threaded execution. The components involved are not directly manipulated by instructions and are thus excluded from the traditional notion of what constitutes Architectural State. For instance, instructions interact directly with the program counter, register file, and memory system — these structures constitute Architectural State. Instructions do **not** interact

directly with branch predictors or prefetch buffers, as these are designed to be transparent performance enhancers. Thus, these latter structures are **not** included in Architectural State.

The divergence creates an ever-widening gap between what a microarchitecture does and what security analyses *assume* it does, with vulnerabilities like Spectre and Meltdown exploiting the space between the two. Traditionally, Architectural State includes only elements directly manipulated by the instruction set. Architects have assumed that new mechanisms are safe to include if they do not alter Architectural State, a practice that ensures backwards compatibility with legacy software. However, this assumption fails for security analysis, a fact that can be demonstrated by considering caches. Caches are considered "safe" to add to a design, because they do not impact Architectural State. Nonetheless, as we have seen, caches create exploitable side channels, which means they are not actually "safe" at all.

Part of the disconnect is that the two fields of hardware and security have completely different perspectives and objectives:

> Regarding the analysis of potential structural changes one might make to a system, computer architects define "safe" (as in "safe to add to the system") in terms of backwards compatibility; security analysts define "safe" in terms of vulnerability.

This is problematic. Members of the two disciplines are not working toward the same goal, and as we have shown, they each have made system-design decisions that contradict the other's fundamental assumptions, thereby undermining system security. The misalignment of these two definitions needs to be rectified if future systems are to satisfy *both* definitions, which is the ultimate goal.

Our existing definition of Architectural State renders current security analyses flawed. It is thus to be considered outdated; it needs updating, so that it supports security analysis.

We propose redefining Architectural State to encompass all shared microarchitectural components, such as caches, branch predictors, write buffers, and translation lookaside buffers. This new definition should enhance security in modern systems by preventing unintended information leaks. Any structure that can hold state across different instructions — unlike individual pipeline registers or reorder-buffer entries, which only hold state for a single instruction at a time — must be included.

6.2 Apply the Notion of Atomic Operation to this New Architectural State

Our research group at the U.S. Naval Academy has developed a formal model for analyzing microarchitectural security, with a brief overview, including an analysis of Spectre/Meltdown on an out-of-order processor, provided in [19]. The model shows that, as one might expect, current out-of-order microarchitectures are vulnerable to Spectre/Meltdown if their speculative execution window is large enough to accommodate the exploit code. This model also reveals a key insight:

- Ensuring that speculative effects remain confined to the speculation window prevents all such exploits.

This is critical: by guaranteeing that speculative instructions do not leak information beyond their execution window, we can block current and future attacks exploiting speculative execution and side channels. This aligns with transactional processing, where operations are said to be *atomic* if they either complete fully or leave no trace. If we can make all access to shared structures atomic (recall that shared structures are the ones that behave as *de facto* memory devices), such that those structures are only modified by an instruction if and when it successfully commits from the reorder buffer, then the side-channel problem is solved.

Computer architects are already familiar with atomic transactions (e.g., see [3]) and already implement atomic update of the register file in high-performance processors. In modern reorder-buffer designs (a combination of [15, 16, 18]), while an instruction is in flight, its computed result is stored temporarily in the reorder-buffer entry. The result is only written to the register file if and when the instruction *successfully commits* — i.e., at some point, the instruction is determined to be non-speculative and unaffected by exceptions, interrupts, or previous branches.

This is effectively *atomic operation*, at least with respect to the register file. An atomic approach to register-file update ensures that the register file either reflects the instruction's result or appears as if the instruction never executed. Note that computer architects treat register-file update in this manner specifically because the register file is a component of the traditional Architectural State. Similar atomic-update mechanisms apply to the other members of Architectural State: the program counter and the memory system.

In contrast, other microarchitectural structures that are supposed to be transparent performance-enhancers (e.g., caches, branch-predictor tables, prefetch buffers, translation lookaside buffers, etc.) are not considered to be part of Architectural State, and therefore the microprocessor's operations on them are **not** atomic.

Our proposed redefinition extends the atomic treatment to cover *all shared structures* — caches, branch predictors, write buffers, TLBs, instruction buffers, prefetch buffers, etc. These structures must be updated by a given instruction only when that instruction commits. If the instruction is canceled, the state of these structures should remain unchanged, as if the instruction never occurred.

By adopting this approach, we can create future systems that are secure against speculative and side-channel attacks. The good news: the technology and expertise to implement this already exist.

7 Conclusion #3: Eschew Complexity Obfuscation

A famous maxim is *eschew obfuscation*, a somewhat self-explanatory way of saying "keep it simple, stupid!" Regarding computer security, complexity of the system design is problematic, as it can cloud the foreground and thereby hide underlying implementation issues from all but the most in-depth of analyses. Worse, design complexity will also hide errors of logic that are embedded within the high-level design itself, irrespective of what implementation errors may also exist. This reality underscores our message that it is bad practice to allow a gap of understanding to exist between computer hardware and computer security.

Exploits that target implementation flaws are much easier to discover than those that target design flaws. If there is a clear mismatch between a design and its final implementation, then there is an evident bug, and it could potentially be exploited. However, if

the implementation matches the design, but there is a flaw in the *design's* logic, then that is harder to recognize. Further, the more complex the design, the more difficult it is to recognize any such error. Complexity of design can hide underlying problems, and it should therefore be avoided as much as possible.

Consider as an example the C-vs-Rust debate.

C is a relatively simple language that is nonetheless extremely powerful in its expressive capabilities. It has been around for a half century and has seen relatively little change since the second edition of K&R [9]. In contrast, as we have described in the previous sections, computer-system designs changed *significantly* over the same period, and those changes led to several important and problematic security holes.

The general consensus of the language crowd has been that C is the problem and needs updating. The language *Rust* [10] was introduced as a solution to the problem: Rust is considered to be a "safe" programming language (note: a term without definition) and therefore a viable replacement for C.

However safe (or "safe") the language may be, Rust introduces tremendous complexity to the system, through its hardware abstractions and sometimes bizarre syntax & semantics. We argue that this complexity is unnecessary, and furthermore, because of it, Rust is likely to introduce a new breed of security concerns that (a) would not exist in a simpler language and (b) will be obfuscated by the complexity of both the language and its implementation and will therefore be difficult to detect.

The traditional theoretical models of microprocessor operation and operating system design were simple ... and, as it turns out, they were also very robust in a security sense. Over time, both hardware and software implementations diverged from the original theoretical models and became significantly more complex. This complexity helped to hide the various design flaws that had crept in with the design evolutions.

We argue for a return to the simpler, more robust, models.

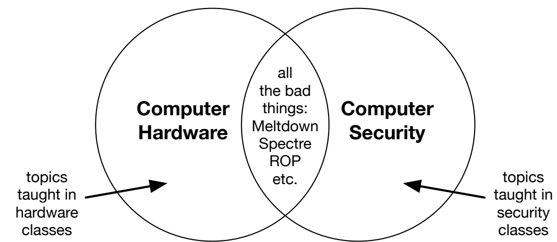
8 Reiteration

Regarding the overlapping region between the two fields of *computer security* and *computer hardware*, there are several important details that warrant our attention and consideration:

(1) Today's most devastating (e.g., problematic and difficult to solve) security holes lie in this region.
(2) There are relatively few subject-matter experts to be found in this region.
(3) As the two fields mature, both microarchitectures and their security issues will become more complex; therefore, the region is likely to become larger, more complex, and increasingly important over time.

In sum, the intersection between the two fields is problematic, is poorly understood, and is growing over time. Such a scenario constitutes a present and growing danger, one that must be addressed if we are to have secure systems in the future.

Explicitly teaching hardware students about security, and security students about hardware, and treating the two topics together instead of separately, will help to mitigate the problem. See [19] for an example treatment of the material.



Acknowledgments

This work was supported by the Department of Defense and the Office of Naval Research.

References

[1] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7* (San Antonio, Texas) *(SSYM'98)*. USENIX Association, 5.
[2] John Fotheringham. 1961. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Commun. ACM* 4, 10 (Oct. 1961), 435–436.
[3] Tim Harris, James Larus, and Ravi Rajwar. 2010. *Transactional Memory, 2nd Edition* (2nd ed.). Morgan and Claypool Publishers.
[4] Mark D. Hill, Jon Masters, Parthasarathy Ranganathan, Paul Turner, and John L. Hennessy. 2019. On the Spectre and Meltdown Processor Security Vulnerabilities. *IEEE Micro* 39, 2 (2019), 9–19.
[5] Jann Horn. 2018. Reading Privileged Memory with a Side-Channel. *Project Zero* 39 (2018).
[6] Ling Huang, Anthony D. Joseph, Blaine Nelson, Benjamin I.P. Rubinstein, and J. D. Tygar. 2011. *Adversarial Machine Learning*. Association for Computing Machinery.
[7] Bruce Jacob and Trevor Mudge. 1998. Virtual memory: issues of implementation. *Computer* 31, 6 (1998), 33–43.
[8] Bruce Jacob, Spencer Ng, and David Wang. 2007. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann.
[9] B. Kernighan and D. Ritchie. 1988. *The C Programming Language, Second Edition*. Prentice Hall.
[10] Steve Klabnik and Carol Nichols. 2017. *The Rust Programming Language*.
[11] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19.
[12] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990.
[13] Aleph One. 1996. Smashing the Stack for Fun and Profit. *Phrack* 7 (1996).
[14] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. Association for Computing Machinery, 552–561.
[15] James E. Smith and Andrew R. Pleszkun. 1985. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture* (Boston, Massachusetts, USA) *(ISCA '85)*. IEEE Computer Society Press, Washington, DC, USA, 36–44.
[16] Gurindar S. Sohi and Sriram Vajapeyam. 1987. Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors. In *Proc. 14th Annual International Symposium on Computer Architecture (ISCA '87')*.
[17] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*. 48–62. https://doi.org/10.1109/SP.2013.13
[18] Robert M. Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development* 11, 1 (Jan. 1967), 25–33.
[19] Bruce Jacob with contributions from Will Casey and Anthony Melaragno. 2026. *Cybersecurity Architecture: The Design and Security of Modern Computing Systems*. Springer Nature. Available early 2026.