Practical Mitigation Guidance for Hardware Vulnerabilities

ANTONIO GÓMEZ-IGLESIAS, EVAN PEREGRINE, AGATA GRUZA, NEELIMA KRISHNAN, AND PAWAN KUMAR GUPTA

Antonio is a software engineer at Intel where he focuses on security software mitigations. He holds a PhD in computer science and has worked in different roles in the areas of performance, computer architecture, parallel programming, and security for the last 15 years.
antonio.gomez.iglesias@intel.com

Evan Peregrine is a software ecosystem engineer at Intel, specializing in long form technical documentation. He contributed to the ACPI specification, Clear Linux, 01.org, and Zephyr Project before joining Intel's software security communications team in 2018.
evan.c.peregrine@intel.com

Agata Gruza has been at Intel for over five years working on performance optimizations of big-data frameworks like Cassandra, Spark, and Hadoop for Intel Architecture. Currently, she is a lead performance engineer and focuses on Linux kernel software mitigation. Agata is a Google (Android Developer) and Facebook AI (Secure and Private AI) scholarship recipient. She holds double MS in computer science and mathematics from Montana State University and The John Paul II Catholic University of Lublin, Poland, respectively. She is an open source contributor and a founder of Women in Big Data NorthWest Chapter. In her free time Agata enjoys hiking and outdoor activities.
agata.gruza@intel.com

Transient execution attack methods and their mitigations have been subject to much scrutiny in recent years. While new hardware platform designs are built to mitigate these methods, existing systems may need to implement microcode or software mitigations. But due to the complexity and variety of these methods, system administrators may wonder what, when, and how to mitigate their systems. We examine common mitigation approaches for the Microarchitectural Data Sampling (MDS) and Transactional Asynchronous Abort (TAA) methods, how these mitigations help prevent attackers from leaking data, how they work to prevent attackers from leaking data, and how sysadmins can configure the mitigations depending on the needs of their environment.

Hardware Vulnerabilities and Transient Execution Methods

In recent years, researchers have demonstrated a novel set of methods known as *transient execution attacks* (TEA, formerly termed *speculative execution side channel*), which target some of the hardware designs introduced in many modern processors, in particular speculative execution. The leading researchers have detailed several variants of this new class of methods that target different hardware components and instructions that execute transiently under various conditions. The hardware industry has responded by issuing microcode updates for affected platforms, developing software techniques to mitigate affected instructions, and changing the designs of new processors. These efforts help ensure that by the time new variants are disclosed, users can protect their systems against potential implementations of these methods. This is a common process that the industry has used to mitigate other hardware issues and errata in the past [1].

Demystifying Microcode

Hardware manufacturers have been using microcode (μcode) since the mid-1990s, among other things to fix bugs found on existing processors. μcode is a way to modify the behavior of hardware without changing the silicon itself by changing how the CPU translates instructions into micro-operations (μops). For example, when a CPU executes x86 instructions, parts of the CPU decode each instruction into a sequence of machine-readable μops that defines what the instruction does. Microcode updates allow hardware manufacturers to modify how particular instructions translate into μops, thereby changing the instruction's behavior.

Software Stack

As seen in Figure 1, there are many different elements in the software stack. Depending on the issue, different components of this stack might change to accommodate new optimization, hardware functionality or to complement μcode changes with additional features. For example:

Neelima Krishnan is a software engineer at Intel. She leads the validation of the security mitigations in the Linux kernel with a special focus on hardware vulnerabilities. neelima.krishnan@intel.com

Pawan Gupta is a software engineer at Intel working on software mitigations for hardware vulnerabilities. He is the author of the TSX Asynchronous Abort mitigation in the Linux kernel. His areas of interest are embedded systems, kernel programming, device drivers, micro-controllers, and real-time operating systems. pawan.kumar.gupta@intel.com

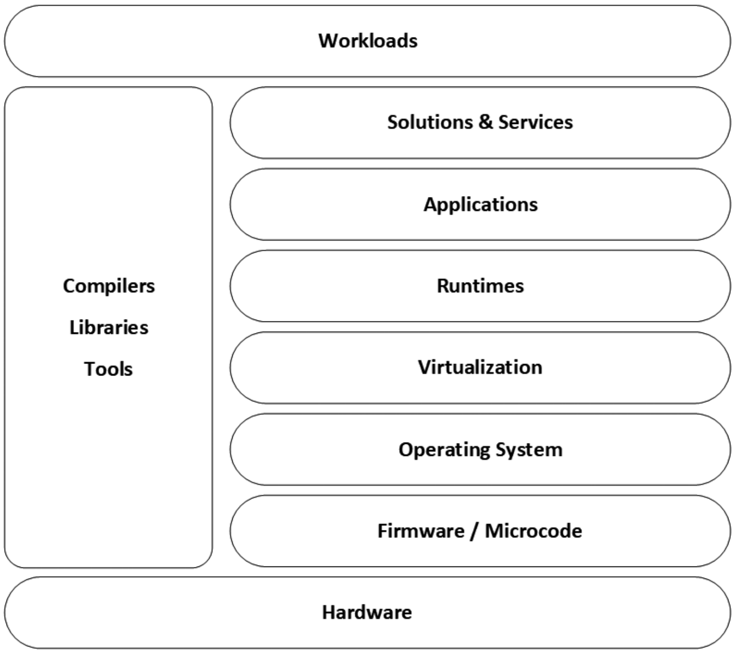

Figure 1: Modern software stack

- The operating system (OS) can include methods that make it more difficult for potential malicious actors to target other processes, other users, or the OS itself.
- Over the years, we have seen how some of these methods target some popular libraries, particularly cryptographic libraries. Popular and well-maintained cryptographic libraries are regularly updated to include programming techniques that make these attacks more difficult. For example, constant-time implementations of crypto algorithms increase their protection against timing methods.
- In certain cases, compilers have introduced changes so that the code generated includes constructs to increase the protection against potential malicious actors. For example, we saw how compilers like gcc included options to protect code against certain Spectre attacks [2].

But the list of software mitigations for these methods does not end here. Software developers regularly update virtual machine managers, web browsers, libraries, tools, and middleware to help mitigate issues originating in hardware [3].

Characteristics of Transient Execution Methods

We focus here on the recently disclosed Microarchitectural Data Sampling (MDS) [4] and Transactional Asynchronous Abort (TAA) [5] methods. In these transient execution attacks (TEA), both the victim (process, kernel, etc.) and the malicious actor must share some physical computing resources. This means that these methods have several inherent restrictions:

- Remote attacks are difficult or not possible. A malicious actor will typically require having local access to a system.
- Any data is accessed in read-only mode. Malicious actors cannot change or roll back a system's data.
- There is no direct privilege escalation. A malicious process cannot give itself root access.
- In some methods, attackers have little or no control over what data they can access. Sophisticated analysis techniques are required to parse secret data out of system noise.
- Both victim and attacker must run on the same physical core.

In addition to these limitations, most TEA share the following procedure:

1. Access target data
2. Send data through a covert channel
3. Receive data from the covert channel
4. Analyze the data for secrets

To demonstrate, consider the following typical scenario: a malicious actor wishes to extract data from a public cloud system where multiple users can access the same machine and run any type of code. In this type of system, an orchestrator or another piece of software will assign a user to a machine according to the user's specified requirements, and the user has little to no control over which machine they are assigned to. The assigned system will typically also be running other users or processes that have been allocated in the same manner, which means a malicious actor has little to no control over which users or processes they can attack. Because these other users and processes can run arbitrary code, a malicious actor needs to work really hard to find a way to force a victim to run a workload that may be of interest to the attacker, and the attacker must also devise a way to infer what code the victim is running. Finally, if the attacker wishes to implement a data sampling method, the malicious process must share those key computational resources for an extended period of time with the victim process to establish certain data access patterns that the attacker can analyze to infer the data that the victim process was using.

Design and Implementation of Mitigations

While there is not a single recipe to follow when mitigating these issues, this section describes the general process used to mitigate MDS and TAA. The mitigations for both issues require changes at the μcode level and the software level and, therefore, are good case studies of the mitigation process for TEA.

Step 1: New Microcode

Let's review an example of how μcode defines how instructions translate into μops executed by the CPU. The MDS and TAA methods try to leak stale data from small microarchitectural buffers inside the CPU, and the mitigations for these methods consist of clearing the affected buffers before their contents can be sent through a covert channel. This raises the question of when and how those buffers are flushed. We cannot clear the buffers in a disorganized fashion, since that could have undesirable effects, such as cross-thread attacks, stalls, or performance implications. One option we do have is to provide a mechanism so software elements higher up in the stack (such as the OS or applications) can decide when to clear the buffers. For that reason, Intel redefined an existing instruction (VERW, Verify Segment for Writing) that was deprecated and not in use. On affected systems, after the μcode update, VERW can be used to flush and clear the content of the buffers affected by MDS/TAA.

Step 2: How to Invoke the New Functionality Provided by the Microcode (if Required)

Now we have a tool (VERW) that software can invoke to clear those buffers. An example of a C function that calls this instruction in the Linux kernel is shown below:

```
static inline void mds_clear_cpu_buffers(void) {
        static const u16 ds = __KERNEL_DS;
        asm volatile("verw %[ds]" : : [ds] "m" (ds) : "cc");
}
```

Listing 1: Linux kernel function to invoke VERW

We mentioned the Linux kernel since the OS invokes this functionality. The OS is the only component of the software stack that can protect different users from user-to-user attacks, as well as protect the kernel itself from potential attacks originating in userspace.

Step 3: Implementing the New Functionality

Now that we have a function like mds_clear_cpu_buffers(), the next step is to identify the right places to clear the buffers. The most appropriate location to flush the buffers is during a ring transition. Ring transitions occur when the system changes the privilege level at which code can execute. For example, if a user application performs a system call to the kernel, a ring transition from ring 3 (userspace) to ring 0 (kernel space) occurs. To mitigate these issues, the VERW instruction is invoked before the system call returns from kernel space back to userspace.

As another example, VERW should be invoked if there is a context switch between different processes, regardless of the owner of those processes. This prevents attacks on systems that disable simultaneous multithreading (SMT), since only one process can run on a physical core at any given time, and the buffers are cleared before another process runs in the same core.

Step 4: Options to Configure the Mitigation and Report Mitigation Status

The last step when it comes to reducing the severity of these security issues is to provide mitigation options so that those with the right privileges can configure them at boot time, as well as a mechanism to detect the status of those mitigations. Sysadmins can control mitigations from the kernel command line. A full list of available options based on the hardware vulnerability is available at kernel.org [6]. In most cases, because transient execution attacks require a malicious actor to be able to execute locally on a system, machines that run only known, controlled, and trusted software may be very difficult or even impossible to target with these methods. Also, due to the nature of the code and users they support, certain systems might not be the target of transient execution attacks and may not need the mitigations. For example, if after a detailed risk analysis where the usage of the system and the characteristics of TEA are considered, the sysadmin decides that the risk of TEA is very low, they may choose to disable the mitigations.

In cases where all the userspace applications are trusted and don't execute untrusted code, then mitigations can be disabled. System administrators may want to disable the mitigations on such systems, as different mitigation options can have performance implications. In other cases, when programs have secret data that needs to be protected (for example, crypto keys), the kernel should provide a full mitigation for the same issue. Also, other components of the software stack, like compilers, might also put in place options to enable or disable the mitigations. System administrators should evaluate their environment and workloads and make an educated decision whether security mitigations are needed.

Sysadmins can use simple tools to check if a given system is mitigated against certain CPU vulnerabilities. In Linux, hardware security issues are associated with a report log, which resides in `sysfs`. The output of `sysfs` indicates if a system is affected by a specific method, and whether the system is mitigated or presently vulnerable.

To reflect recently disclosed vulnerabilities the OS needs to be up-to-date, either by updating the existing kernel or upgrading to the most current one. Updating the OS might seem like a daunting task that takes a significant amount of time. To ease the burden, security researchers created system vulnerability checkers, such as a tool for Linux and BSD available at GitHub [7] to detect whether a given machine is affected by TEA methods. Those tools provide detailed information about hardware support for mitigation techniques if a system is vulnerable to TEA, whether vulnerable systems can be mitigated with a µcode or OS update, or if software changes are required.

Other Techniques for Preventing Attacks

We have seen how these methods take advantage of hardware resources that are shared among different processes. It is possible to limit this resource sharing and thereby reduce the attack vector, with some caveats.

The first challenge here is system load. Some systems are configured to run many more processes at any given time than currently available physical cores on the system. In these cases, resource sharing is unavoidable. However, on systems where the total number of user processes doesn't exceed the total number of physical cores, it is possible to schedule processes to always run on the same physical core and never share that core with other processes, thereby reducing the chances for a malicious actor to implement one of these methods. Linux tools like `numactl` and `taskset` can be used to set the affinity of processes and implement this type of process scheduling. Also, `cgroups` can be an alternative to create process isolation.

The open source community is working on a Linux kernel scheduling technique to implement a similar solution. This technique, called *core scheduler*, allows system administrators to tag specific processes. Processes sharing a tag can run simultaneously on the same physical core (when SMT is enabled), while processes with different tags are prevented from running concurrently on the same physical core. When one process stops running, either because the process is finished or because the OS schedules a different process, the hardware resources (such as buffers and branch predictors) are cleared before another process can use them. Other operating systems and virtualization tools might also implement similar techniques.

The second caveat is interruptions. By default, interrupts can run on any core of the system as decided by the OS. So, if that is the case, then interrupts might be a target of a potential malicious actor. Particularly in systems with SMT on, a malicious actor may be able to target the data accessed by the interrupt while this interrupt is executing on the same core. However, system administrators can now choose to specify cores in the system to handle all system interrupts, preventing any user processes from running on those cores [8]. System administrators should carefully consider the implications of this approach (how it affects the overall throughput of the system, the number of system calls that are normally handled, etc.).

Conclusion

While transient execution methods have affected many modern CPUs, the industry has collaborated to ensure that mitigations for these methods are available by the public disclosure date. This requires understanding the implications of these methods and the optimum solution for mitigation. Since new hardware includes mitigations against these methods, an approach might be to update the hardware. However, because changing hardware takes time, is costly and challenging, other alternatives (like updating microcode and making changes to the software stack) are needed to mitigate existing vulnerabilities. It's crucial for system administrators to understand that the technical mitigations are just one component of the security process. Enabling sysadmins to choose what mitigation approach works best for their environment and workloads is key. Providing alternatives, explaining how the different mitigation methods work, and outlining the factors to be considered for each mitigation approach, all help enable system administrators to choose the most appropriate actions.

References

[1] Microsoft, "Host Microcode Update for Intel Processors to Improve the Reliability of Windows Server": https://support.microsoft.com/en-us/help/2970215/host-microcode-update-for-intel-processors-to-improve-the-reliability.

[2] GCC 7.3 release notes: https://lwn.net/Articles/745385/.

[3] Intel, "Deep Dive: Managed Runtime Speculative Execution Side Channel Mitigations": https://software.intel.com/security-software-guidance/insights/deep-dive-managed-runtime-speculative-execution-side-channel-mitigations.

[4] Intel, "Microarchitectural Data Sampling": https://software.intel.com/security-software-guidance/software-guidance/microarchitectural-data-sampling.

[5] Intel, "Intel Transactional Synchronization Extensions Asynchronous Abort": https://software.intel.com/security-software-guidance/software-guidance/intel-transactional-synchronization-extensions-intel-tsx-asynchronous-abort.

[6] Linux Kernel, "Hardware Vulnerabilities": https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/.

[7] Spectre & Meltdown checker: https://github.com/speed47/spectre-meltdown-checker.

[8] Linux Kernel, "SMP IRQ Affinity": https://www.kernel.org/doc/Documentation/IRQ-affinity.txt.