TEEM: A CPU Emulator for Teaching Transient Execution Attacks

Ben Swierzy, Melina Hoffmann, Felix Boes, Felix Betke, Lennart Hein, Maxim Shevchishin, Jan-Niklas Sohn, Michael Meier

Abstract: Side channel attacks have been an active field of attacker research for decades. The Spectre, Meltdown and Load Value Injection publications established a new type of attacks, known as transient execution attacks, which utilize that architectural rollbacks leave traces in microarchitectural caches and buffers. These can serve as covert channels, resulting in practically relevant but hard to prevent attack scenarios. The associated weaknesses are complex, which makes it hard for security researchers to detect them and even harder for developers to prevent them. To achieve advancements in this field it is important to teach students about the underlying concepts. However, the documentation of modern CPUs is neither complete nor correct, which increases difficulties in obtaining practical experience. As a result, there is a need for a CPU emulator that facilitates practical learning with options for looking inside the box. We contribute TEEM, a Transient Execution EMulator of a RISC-V CPU supporting several microarchitectural features relevant for teaching transient execution attacks. Our empirical teaching experiences clearly indicate an improvement in the student's understanding of Meltdown and Spectre.

Keywords: Side Channels Attacks; Emulation; Teaching

1 Introduction

Over the past three decades, side channel attacks have been on a constant rise and threaten a variety of systems. In many cases, the associated weaknesses are sophisticated, which makes them hard for security researchers to detect and even harder for developers to prevent. There exist numerous variants of side channel attacks. Some variants, like power analysis attacks, measurements of electromagnetic radiation and invasive methods, require physical

1 University of Bonn, Institute for Computer Science IV, Friedrich-Hirzebruch-Allee 8, 53115 Bonn, Germany, swierzy@cs.uni-bonn.de
2 University of Bonn, Institute for Computer Science IV, Friedrich-Hirzebruch-Allee 8, 53115 Bonn, Germany, hoffmann@cs.uni-bonn.de
3 University of Bonn, Institute for Computer Science IV, Friedrich-Hirzebruch-Allee 8, 53115 Bonn, Germany, boes@cs.uni-bonn.de
4 University of Bonn, Institute for Computer Science IV, Friedrich-Hirzebruch-Allee 8, 53115 Bonn, Germany
5 University of Bonn, Institute for Computer Science IV, Friedrich-Hirzebruch-Allee 8, 53115 Bonn, Germany
6 University of Bonn, Institute for Computer Science IV, Friedrich-Hirzebruch-Allee 8, 53115 Bonn, Germany
7 University of Bonn, Institute for Computer Science IV, Friedrich-Hirzebruch-Allee 8, 53115 Bonn, Germany
8 University of Bonn, Institute for Computer Science IV, Friedrich-Hirzebruch-Allee 8, 53115 Bonn, Germany & Fraunhofer FKIE, 53115 Bonn, Germany, mm@cs.uni-bonn.de, michael.meier@fkie.fraunhofer.de

doi: 10.18420/sicherheit2024_013
access to the attacked device. In contrast, many timing side channels or padding oracles can be exploited remotely through software or network interactions. Simple examples range from variable-time string comparisons, allowing to reconstruct passwords, to the Lucky Thirteen attack on TLS [AP13], which facilitates the decryption of messages through a timing-induced padding oracle.

The publications of Meltdown [Li18], Spectre [Ko19] and Load Value Injection (LVI) [Va20] laid the foundations for a new side channel attack type called transient execution attacks. These attacks exploit a side effect of modern optimization techniques, namely, that the instruction-level execution of a program does not necessarily match the microarchitectural execution inside the CPU. They exploit the transient execution of instructions, which are rolled back on an instruction set level but modify the microarchitectural caches and buffers. These can be utilized as a covert channel by measuring the access time on memory locations. The resulting attack scenarios are practically relevant and difficult to prevent. For example, successful attacks are regularly found for trusted execution environments undermining their security guarantees. As the vulnerability is in the hardware, it is difficult to patch and mitigations can be costly, e.g., causing a overhead factor of 2 to 19 for preventing LVI [Va20].

The increasing complexity of microarchitectural and other side channel attacks in recent years necessitates teaching students about the underlying concepts in detail, to promote advancements in this field among the upcoming security researchers. However, CPU manufacturers try to mitigate transient execution attacks as soon as possible and the documentation for modern processors is neither complete nor correct [Va21]. Thus, real-world exploits and mitigations need to be based on assumptions which need to be constantly re-evaluated and later can turn out to be incorrect [Sc21]. This greatly increases the difficulty for students to obtain valuable practical experience. In this scenario, an emulator can be highly beneficial as it facilitates practical learning while providing options for taking a look inside the box at the same time. Prior work either focuses on simplicity and disregards out-of-order executions and mitigations [Mi22], or stays very close to realism disregarding any educational aspects [SF21].

Our work contributes the educationally-focused Transient Execution EMulator (TEEM) resembling a modern RISC-V CPU with out-of-order execution, delayed fault verification, speculative execution, a data cache, a branch target buffer (BTB), and a return stack buffer (RSB). Furthermore, we share our experience on the advantages and further potential this tool offers for teaching.

2 Background

In this section, we briefly explain basic terminology and concepts relevant for transient execution attacks. Due to space restrictions and the complexity of this topic, this explanation
is far from complete. For a more comprehensive introduction, we refer to the overview chapter in the habilitation of Daniel Gruss [Gr20].

Instructions executed by modern CPUs have different latencies. If an instruction needs to fetch data from memory, it takes far more cycles than simple arithmetic operations. As manufacturers are constantly endeavouring to increase the instruction throughput of their processors, they try to prevent the CPU from being stalled. For this, superscalar architectures are deployed, which decode and issue multiple instructions per cycle and use multiple execution units for instruction-level parallelism. Furthermore, out-of-order execution is able to reorder instructions and assign them to free execution units while honoring data dependencies. This introduces a first gap between the architectural and the microarchitectural state in the CPU.

This gap increases when transient execution techniques are in use. With delayed fault verification, the CPU assumes that instructions, which might cause a fault, will likely execute without causing a problem. Thus, the following instructions can already be executed and increase the throughput, if the assumption is correct. In case the assumption is incorrect, the transiently executed instructions are rolled back and neither modify the architectural state nor cause a relevant performance penalty. However, they modify the cache, which is exploited by Meltdown to extract transiently-obtained protected information. In addition, conditional and indirect branch instructions are commonly causing stalls as the CPU needs to wait for all data dependencies to be resolved before knowing where the execution continues. For this, a branch prediction unit (BPU) predicts whether a branch is taken or not, depending on past information. Furthermore, a BTB stores past jump targets and an RSB stores past call locations to predict return addresses. The predicted control flow is speculatively executed until it is either confirmed or needs to be rolled back. Similar to before, the speculatively executed instructions affect the microarchitectural state and are utilized by Spectre to leak secret information.

Since these vulnerabilities open severe attack scenarios, e.g., dumping the complete physical memory [Li18] and stealing information across isolation levels like virtual machines [Br19], there have been several mitigations deployed to prevent these attacks. An undocumented hardware mitigation from Intel causes illegal reads to always return zero preventing some types of Meltdown [Ca20]. Additionally, there exist software mitigations such as serialization through fence instructions which force stalls until memory operations are finished. Some mitigations such as flushing all caches and buffers after context switches are possible but not practical due to very large performance penalties.

3 CPU Emulator

TEEM consists of two interacting parts: a backend that implements the actual emulator and a user interface (UI). The UI offers easily understandable visualizations of the emulated CPU components and interaction with the running emulator via the command line. Sect. 3.1 gives
an overview of the central planning and design decisions. Sect. 3.2 and 3.3 highlight aspects of the implementations of the emulator backend and the UI respectively. The emulator is implemented in Python and publicly available\(^9\) under the GPLv2+ license.

### 3.1 Methodology

The development of TEEM follows the subsequent overarching design goals: The main goal is to emulate the components and processes inside an actual CPU as detailed and realistic as needed to demonstrate transient execution attacks. Simultaneously, the emulator should be kept as simple as possible so it is still easily understandable both for students using it and future developers extending it. Additionally, it should provide convenient and low-threshold user interactions, so students can quickly focus on learning about transient execution attacks. This should include an easily understandable UI that offers interaction with the running program and detailed visualizations of the inner workings of the emulated CPU.

To this end, TEEM’s backend is modular and based on information about modern real-world CPUs. However, there are some simplifications to ease its usage and understanding. In general, it only emulates a CPU and has no notions of further abstraction levels like virtual memory or an operating system, since these are not needed for basic transient execution attacks. Furthermore, the emulator does not differentiate between macro- and microinstructions but instead wholly operates on textual RISC-V assembler instructions. The differences between the macro- and microarchitectural state, that are needed for transient execution attacks, are still modeled by the difference between the expected behavior from the program sequence and the actual steps performed in the execution unit during out-of-order and speculative execution with their side effects on internal components. Additionally, the emulator features only one execution engine with a unified reservation station that contains several multi-purpose instruction slots, which eases the visualization and understanding of out-of-order execution without detracting from the demonstration of the desired attacks. In particular, it simplifies the implementation of the emulator’s execution engine component, which contributes to the goal of keeping the emulator comprehensible for students and future developers. Likewise, there are further real life CPU components that have been simplified or omitted in general, since they are not vital for the considered attacks. TEEM also features a gdb-like user interface to interact with the program running in the emulator and to visualize the emulated CPU components and their current values.

### 3.2 Backend

TEEM’s emulator backend has a modular structure, as shown in Fig. 1. It is centered around the CPU component that initializes the other components and interfaces between them. It

\(^9\) https://github.com/teem-cpu/teem
provides a tick function that mimics the clock cycle of a real processor and triggers the other components. But, specific to TEEM, it also provides the interface for the UI to obtain the current state of the emulator backend and stores snapshots of the emulated CPU state to support time travel debugging.

TEEM implements the RISC-V RV32IM architecture [WA19] with partial support for the Zifencei [WA19], Zicsr [WA19], Zibom [RI21a] and XTHeadCmo [T-23] extensions, which introduce cache and timing instructions. Input programs can be written directly in RISC-V assembly. To increase the similarity to real proof-of-concept attacks, TEEM supports C code that is compiled to RISC-V assembly with clang in version 17+. Besides interpreting the instructions, the emulator’s parser interprets several directives, including switching between text and data sections and emitting data words.

TEEM’s frontend component supplies the execution engine with instructions ready to be issued. Since TEEM does not distinguish between macro- and microinstructions, the decoding and optimization steps of real-world CPU frontends are omitted. The frontend holds an instruction list and uses it to maintain and provide a queue of the upcoming instructions in program order. After branch instructions, it fills the queue according to the emulator’s BPU, which contains a pattern history table (PHT), a BTB and a RSB. The size of the transient execution window is implicitly configured by the execution time of the instructions.

The memory subsystem component manages both the main memory and the cache. Compared to a modern real-world CPU, the memory hierarchy is very condensed and simplified, consisting only of a main memory and a single cache that caches loads and stores performed on this main memory. TEEM implements a von Neumann architecture with the memory holding only dummy bytes for instructions, since their semantics are handled separately through the parser and frontend. The upper half of the address space is declared inaccessible, so memory operations on these addresses lead to a fault and subsequent rollback. This mimics
real-world page faults when unprivileged users try to read from kernel space, which are utilized in Meltdown-US-L1. To enable transient execution attacks, the memory subsystem also implements delayed fault verification by caching the results of load operations before the permission check is completed. The number of sets, ways and entries per cache line as well as the replacement strategy of the cache is fully configurable.

The execution engine executes instructions and manages the registers. It receives the instructions in-order from the queue and then executes them using speculative execution on branch instructions and out-of-order execution following a simple version of Tomasulo's algorithm \cite{To67}. As part of the speculative execution, the instruction queue is already filled according to BPU predictions. During execution, the actual branch target is evaluated and the BPU notified. Depending on whether or not the instruction faults, e.g., due to a misprediction, the execution engine either resumes the execution or performs a rollback.

A rollback restores the execution engine's state back to before the faulting instruction was issued, i.e., the register state is restored from a snapshot and all instructions that are still in execution are cleared from the execution engine. The central CPU component is also notified of a fault and coordinates the rest of the fault handling, e.g., flushing and repopulating the instruction queue. Note that the state of the cache and of the BPU are not restored during a rollback. This models the fault handling in real-world CPUs where the architectural state is rolled back and restored while side effects on microarchitectural components remain.

To enable out-of-order execution, TEEM implements a simple version of Tomasulo's algorithm \cite{To67}. In this version, the execution engine contains one unified reservation station with multiple instruction slots, where each type of instruction can be issued to any of these slots. The execution engine does not model individual execution units as separate components. The instructions are issued into the instruction slots in program order as given by the instruction queue. If an instruction has a result register, the slot id is put into the register as a placeholder until the result value is determined. Upon issuing, register operands are filled with the current value from the corresponding register. This can be either a concrete value or a slot id for the instruction that will provide the register value when it has been executed. Since instructions are issued in-order, the state of the register file represents the architectural state at that point in time with yet-unknown values present as slot references. An instruction can be executed as soon as conflicting behavior is impossible and all of the placeholder operands are resolved. In particular, they do not have to be executed in program order. In each cycle, at most one instruction gets executed. The registers and other instruction slots are notified of the slot id and result of the executed instruction, so placeholders are updated with the concrete result values. This models the common data bus that broadcasts at most one instruction result in each clock cycle of a real-world CPU.

After an instruction has been executed, it retires in a subsequent tick cycle in-order. During retirement, it is checked whether the instruction causes a fault or not. If it does not itself cause a fault, it can retire immediately and simply be removed from the execution engine. If it does fault, it has to wait for preceding potentially faulting instructions to retire to
ensure that faults are handled in program order. It also needs to wait for the architectural
register state to emerge, so it can be restored correctly. If the instruction still causes a fault,
a rollback is performed and the main CPU component is notified. In case of a faulting and
reverted load instruction, the execution is resumed at the next instruction, so users do not
have to implement fault handlers. This does not hinder the demonstration of attacks that
exploit transient execution. As described above, the memory subsystem performs delayed
fault verification, but it ensures that only non-faulting instructions can write to the actual
main memory. In particular, checking for faults only after the instruction has already been
executed and the result broadcasted leaves a time window to transiently execute instructions
that come later in the program order. This allows for transient execution windows to emerge,
which are exploited in Meltdown type attacks. Furthermore, together with the speculative
scheduling and execution as described above, this enables Spectre type attacks.

Additionally, some instructions have associated wait times to simulate the latencies and
differing execution times in real-life CPUs. Each jump and arithmetic instruction has a preset
minimum execution time that is determined by the concrete instruction type. Instructions
have to remain in the execution engine slot for at least this preset amount of tick cycles
before they can be executed and their result broadcasted. Per default, it is set to one cycle to
model their quick execution in real-life CPUs. This is not necessary for memory instructions,
since the memory subsystem already provides configurable wait times for different kinds
of accesses. Their default values are chosen so the minimum wait times are staggered, so
cache hits have the shortest, cache misses and succeeding writes to memory have the same
medium and faulting memory operations have the longest minimum wait time. These timing
differences are necessary for transient execution windows to emerge and for differentiating
between cache hits and misses with typical cache-based side channel attacks.

Modern CPUs use a memory disambiguator to prevent memory hazards during out-of-order
execution, which is exploitable [Ho18]. In the current version of TEEM, no memory
disambiguator is implemented. Instead, the execution engine checks before the execution of
each instruction that interacts with the memory, whether there are any preceding instructions
that access the same memory section. If this is the case or if this cannot be determined, e.g.,
in case the memory address of a preceding instruction is not yet calculated, the instruction
is not executed and has to wait for the preceding instructions.

3.3 User Interface

TEEM’s UI resembles the popular gdb interface. Programs can be run stepping one cycle at
a time. For convenience, the interface also includes commands to take multiple steps at once
and to set breakpoints to fast forward through the execution. Time travel debugging allows
the user to conveniently step back, allowing repeated study of a critical part of the execution.
Note that this feature restores the complete emulator state and is therefore different from
the architectural rollback during fault handling. It also offers convenience features like tab
completion and the repetition of the last command with the enter key.
The UI offers a visualization of the current state of the emulated CPU components and the memory. A compact rearrangement of the dashboard is depicted in Fig. 2. The dashboard includes the upcoming instructions, the queue, the instruction slots, the registers and part of the memory. In particular, their interaction showcases the stepwise execution of Meltdown and Spectre type attacks. For the other components, the current state can be shown with respective commands. For example, the contents of the cache can be shown repeatedly to follow the steps of a cache based side channel attack. The UI also allows for the user to change the values of registers, memory locations and BPU entries, and to flush the cache while the program is running.

To enable interaction between the UI and the emulator backend, there are two special instructions: `ebreak` causes the program execution to halt for a breakpoint when executed while `ecall` is used to invoke a small set of emulator specific system calls that are used to read from and write to the text console and to exit the program. Both `ebreak` and `ecall` serialize the instruction stream and execution order similarly to a `fence` instruction, so register and memory values are consistent for both the user and the system call handler.
4 Evaluation of Capabilities and Limitations

It is necessary for TEEM to implement several RISC-V extensions for special functionality like cache flushing, serialization and high precision timers. An additional C header file includes startup code and wraps the special functionality in functions and macros, so that inline assembly in C is not necessary. However, due to lack of support for RISC-V machine code representation, it is not possible to use ELF binaries, dynamically linked objects or multiple input files. This also implies that the standard C library is not available. Instead, a basic IO feature based on system calls is available with convenient wrappers in the header file. Furthermore, TEEM does not implement the concept of virtual memory, so programs directly access the physical memory locations. This simplifies the implementation of transient execution attacks, since it allows ignoring technical details. For instance, in a Linux environment, `flush(libFunc)` flushes the address of the associated PLT entry but does not remove the library function code from the data and instruction caches. On the other hand, it becomes more difficult to convert attacks from the emulator to the real world.

The emulator allows to perform different kinds of transient execution attacks. For exploiting speculative execution, the variants Spectre-PHT, Spectre-BTB and Spectre-RSB [Ca19] can be performed. Spectre-STL [Ho18] is not yet possible as no memory disambiguator is implemented. For exploiting delayed fault verification, Meltdown-US-L1 can be performed as the userspace (US) flag of a memory page is indirectly emulated by the inaccessibility of the upper half of the address space. Other Meltdown type attacks [Ca19] cannot be performed in the emulation due to the lack of virtual memory. To demonstrate these capabilities, the emulator is bundled with a set of example programs. All examples for attacks and mitigations are written directly in RISC-V assembly with the exception of Spectre-BTB, whose example is written in C to illustrate usage of the C interface.

Furthermore, the effect of different mitigations on the attacks can be studied. For this, it is possible to disable out-of-order execution, to utilize control flow serialization and to deploy the return zero on illegal read mitigation described in Sect. 2. Additionally, microprograms can be used to patch the emulators behavior on faults depending on the faulting instruction type, e.g., flushing the cache after a faulting load instruction.

The core design objective for TEEM is improving the students’ understanding for transient execution attacks. This is achieved by a transparent implementation of the out-of-order execution through visualization of the reservation stations, highlighting of branch prediction results and the possibility to inspect the caches and buffers. It allows students to follow an explorative discovery process by observing direct microarchitectural effects. Thus, TEEM fulfills its major objective. However, the differences between C code written for the emulator and real-world exploits do not ease the students’ understanding when studying proof-of-concept code. Furthermore, the emulator manages to perform around 100 instructions per second on an average desktop computer, which is orders of magnitude below the efficiency of existing CPU emulators like QEMU [Be05]. Though, this does not harm the use case, since transient execution attack programs are rather short.
5 Teaching Experience

In the following section, we report on our empirical teaching experience with TEEM. It must be noted that this is not a significant scientific evaluation and the few quantitative results are merely a small indicator. The required methodological setup for a rigorous user study would need to be deployed at least a year before the introduction of TEEM which did not fit into the short cycles of our teaching improvement process. Nonetheless, we think that our teaching experience complements the technical descriptions of TEEM.

In the Master programmes Computer Science and Cyber Security at the University of Bonn, the lecture *Side Channel Attacks* is taught since 2021 and has been held three times. In each lecture series, 20 to 35 active students participated in the problem sessions and the exams. Each lecture series consists of 15 lectures and 14 problems sessions in total. Basic concepts, padding oracle attacks, timing attacks, power analysis attacks and faulting attacks are covered in the first six lectures and are practically explored by the students in the problems sessions. In the remaining nine lectures, CPU architectures, cache side channels and transient execution attacks are taught in detail.

In the first two iterations of the lecture, the students worked with a rudimentary Meltdown simulator to practically explore cache side channels and the idea of delayed fault verification. During the discussion of the problem sheets and in the exam, it was observed, that the students grasped practically-taught concepts better than in a purely theoretical way. For the third edition, a group of four students, that passed the exam with great success, developed the majority of TEEM in a lab course. In the problem sessions and in the exam, the students were able to discuss the technical details covered by TEEM more comprehensively. Especially, the timing and sequence of the different steps in a Meltdown-type attack were understood with a noticeable higher quality than before. The benefits have also been observed in the exam results, which improved from an average German grade of 2.6 (open book exam) and 2.4 (oral exam) to 2.0 (closed book exam) when TEEM was introduced.

6 Conclusion & Future Work

Transient execution attacks are a rather young but complex class of side channel attacks that threaten modern computer systems. We presented TEEM, a transient execution emulator, that helps students to enter this research field by experimenting with Meltdown and Spectre attacks. By implementing many modern CPU paradigms and RISC-V as instruction set, the emulator stays close to modern processors without being challenging to work with. In our lecture series, the corresponding problem sessions and exams, TEEM clearly helped students to understand and communicate the concepts of transient execution attacks.

In the future, it is planned to further evaluate TEEM with the help of a student study. We expect it to scientifically support our empirical observations. Additionally, the emulator can be extended to support a wider range of attacks and mitigations. For example, some
of the RISC-V vector extensions [RI21b] could be implemented to be able to demonstrate the Downfall attack [Mo23] or the RSB stuffing mitigation [Tu18] could be introduced. Furthermore, the concept of memory disambiguation, virtual memory, multiprocessing and context switches could be adopted, so that the attack scenarios are closer to the real world attacks between isolated processes or kernel and userspace. Then, it becomes reasonable to extend the emulator to attacks such as LVI.

References


