One Flew over the Stack Engine's Nest: Practical Microarchitectural Attacks on the Stack Engine

Silvan Niederer
nsilvan@ethz.ch
ETH Zurich
Zurich, Switzerland

Sandro Rüegge
rueegges@ethz.ch
ETH Zurich
Zurich, Switzerland

Ali Hajiabadi
ahajiabadi@ethz.ch
ETH Zurich
Zurich, Switzerland

Kaveh Razavi
kaveh@ethz.ch
ETH Zurich
Zurich, Switzerland

Abstract

Security research on modern CPUs has raised numerous concerns in recent years. These security issues stem from classic microarchitectural optimizations designed decades ago, without consideration for security. Stack pointer tracking, also known as the stack engine in recent CPUs, is one such optimization. To investigate the security implications of the stack engine, we reverse engineer its operational details on a number of recent Intel and AMD CPUs for the first time. Our results show the particular microarchitecture-dependent behaviors of the stack engine, such as the conditions under which it needs to synchronize the stack pointer values with the backend. Using these results, we build three primitives called `Direct Underflow`, `Sync+Reload` and `Prime+Sync+Probe` that enable information leakage through the stack engine under different conditions. We use these primitives in the construction of various covert and side-channel attacks, leaking sensitive patient records from a widely-used JSON library as an example. Our mitigation efforts reveal that recent AMD Zen 4 and Zen 5 CPUs include undocumented chicken bits which allow enabling or disabling the stack engine. Using these bits to disable the stack engine, we measure 3.98% and 3.94% slowdown using SPEC CPU2017 on Zen 4 and Zen 5, respectively, prompting the need to consider more secure designs for the stack engine in future CPUs which we also discuss.

CCS Concepts

• **Security and privacy → Side-channel analysis and countermeasures**.

Keywords

Stack Engine, Side Channel, Reverse Engineering

ACM Reference Format:
Silvan Niederer, Sandro Rüegge, Ali Hajiabadi, and Kaveh Razavi. 2025. One Flew over the Stack Engine's Nest: Practical Microarchitectural Attacks on the Stack Engine. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3725843.3756058

MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/2025/10
https://doi.org/10.1145/3725843.3756058

1 Introduction

Security research on microarchitectural optimizations of commodity CPUs has revealed an overwhelming number of security concerns in recent years [7, 12, 13, 15, 16, 24, 28, 29, 31, 32, 34, 35, 37, 40, 41, 43, 44, 51, 53–56, 60, 61]. This body of work has shown that many optimizations, originally designed decades ago, have unintended security implications that lead to sensitive information leaks. In this work, we analyze a classic microarchitectural optimization: *stack pointer tracking* [8] which is also known as the *stack engine* in recent CPUs. The insights from our reverse engineering on a wide range of x86 CPUs from Intel and AMD enable us to build novel primitives to leak sensitive information from the stack engine in a variety of scenarios.

Stack engine. Many architectures rely on the stack for calling conventions. In x86, stack-specific instructions, such as `push`, `pop`, `call`, and `ret`, enable compilers to pass arguments and manage subroutines. The stack also serves as a scratch space for register spills. The `RSP` register always holds the current top of the stack. Any `push`/`pop` translates to two implicit operations: (1) `RSP` update (increment in case of `pop` and decrement in case of `push`), and (2) a store/load operation at the address in the `RSP`. This implementation creates a chain of data dependencies among stack operations: all stack operations must update and use the latest `RSP` value. Stack pointer tracking is a widely used frontend optimization that resolves these dependencies after instruction decode [8]. This optimization tracks the `RSP` offset (referred to as Δ`RSP`) that is used to (1) perform the `RSP` updates in the frontend so that fewer operations are dispatched to the backend, and (2) update the store/load operations with the tracked Δ`RSP`. This allows stack operations to execute independently in the backend and consequently increases the instruction-level parallelism. Modern CPUs from AMD and Intel implement similar mechanisms, known as the stack engine [2, 27].

Reverse engineering the stack engine. Given that many microarchitectural optimizations have been shown to leak sensitive information [31, 37], an important question arises: *can an attacker exploit the stack engine to leak information in security-sensitive setups?* To address this, we conduct a detailed reverse engineering of the stack engine to uncover its mechanics and internal states in the latest AMD and Intel CPUs. To the best of our knowledge, this is the first reverse engineering of the stack engine. Our findings confirm that all recent AMD and Intel CPUs, including AMD Zen 5 and Intel Alder Lake, implement a stack engine. We observe that the stack engine dispatches an additional ALU operation under specific conditions. First, different CPUs support different maximum ΔRSP for their tracking. When this depth is exceeded (ΔRSP overflow), the stack engine triggers a synchronization that behaves like an ALU operation. Second, operations not supported by the stack engine (e.g., an `or` on the RSP) also cause a synchronization, but only if ΔRSP is nonzero. In other words, unsupported stack operations do not trigger a synchronization when ΔRSP is zero.

Leakage primitives from the stack engine. Once we know how the stack engine works in different CPUs, we aim to build reliable primitives to trigger and observe the stack engine's additional operations. First, we develop three primitives that trigger additional operations caused by the stack engine. (1) *Direct Underflow* primitive exploits the cases where a deep call stack (based on each CPU's maximum supported depth) triggers an additional operation upon overflow. (2) *Sync+Reload* exploits the fact that unsupported stack operations dispatch an additional operation only if the ΔRSP updates are unbalanced (i.e., ΔRSP is nonzero). While the last two primitives work on most CPUs, we need a more general primitive for modern CPU architectures like AMD Zen 5. These CPUs track a large ΔRSP where direct overflows occur rarely and, in addition, they unconditionally dispatch additional operations upon unsupported stack operations. (3) *Prime+Sync+Probe* primes ΔRSP to a specific value and selectively increases it to test whether an overflow occurs. This later enables us to observe a victim's execution that has variable stack depths depending on a secret value in the probe phase.

Stack engine exploitation. Using our stack engine primitives, we first build two covert channels under same-thread and cross-thread leakage models. In a same-thread leakage model, where attacker and victim run within the same process, additional operations can be observed using performance counters. Our channel achieves a leakage rate of 3 kB/s with negligible error, and 48 kB/s with an error rate of 25%. However, observing these additional operations across Simultaneous Multithreading (SMT) cores is more challenging. Prior port contention techniques [3, 23] could, in theory, detect such operations. However, these methods rely on dependent instructions issued to a single execution port with multi-cycle latencies. In contrast, the additional operations dispatched by the stack engine are independent, complete in a single cycle, and can be issued across multiple ports, like a simple `add`. As a result, prior techniques produce weak and indistinguishable signals. We devise a new port contention technique that enables precise observability of the stack engine operations using a new dispatch alignment technique. Our cross-thread covert channel achieves 12.5 kbit/s with an error rate of 2%. We further demonstrate a highly reliable side channel attack against cJSON [19], a widely-used JSON parsing library. Our attack leaks the structure of the parsed data and ultimately extracts sensitive information, distinguishing patients based on their medical records with an accuracy of 100%.

Our mitigation efforts reveal an interesting finding: AMD Zen 4 and Zen 5 CPUs include undocumented chicken bits that enable or disable the stack engine. We confirm this using our reverse engineering microbenchmarks. Disabling the stack engine introduces a performance overhead of 3.98% on Zen 4 and 3.94% on Zen 5, measured using the SPEC CPU2017 [48] benchmark. This prompts the design of secure stack engine architectures in the future which we also discuss.

Contributions. The main contributions of this work are as follows:

- We provide a detailed reverse engineering of the stack engine for recent AMD and Intel CPUs, uncovering its mechanics and internal operations across different microarchitectures;
- We identify a set of stack engine primitives that allow us to observe stack operations across various security boundaries;
- We present a set of covert channel and side channel attacks for same-thread and cross-thread leakage models. We demonstrate a side channel attack on a widely used JSON parsing library, leaking sensitive information about parsed data records such as patients' data;
- We uncover undocumented chicken bits in recent AMD CPUs to disable the stack engine. This allows us to both analyze the performance benefits of the stack engine and to mitigate our attacks.

Responsible disclosure. We disclosed our findings and the resulting security implications of the stack engine to both AMD and Intel. Neither vendor requested an embargo. AMD has released a security brief in response to our disclosure, noting that this potential vulnerability is mitigated by using constant-time code when handling secret data [5]. Intel similarly recommends following their Data Operand Independent Timing guidance when writing constant-time code [26]. You can find additional information, including the source code for our experiments and attacks, at the following link: https://comsec.ethz.ch/stackengine

2 Background

We provide the necessary background on high-level CPU architecture (Section 2.1), stack management in software (Section 2.2), and microarchitectural optimizations for stack management (Section 2.3).

2.1 CPU architecture

In this work, we focus on the x86 CPU architecture from dispatch to integer execution units as shown in Figure 1. The frontend uses (x86)-decoders to convert opcode sequences into operations that are easier to handle than the variable-length x86 sequences. The resulting operations are referred to as *Macro Op* (MOP). Once decoded, the opcache holds MOPs until they are evicted. Optimally, instructions are only decoded once and all subsequent executions of the same code require only opcache lookups. A dispatch stage selects operations and *dispatches* them to scheduler(s), breaking them into smaller operations referred to as *micro ops* (uops).

A scheduler provides out-of-order functionality. It tracks dependencies and *issues*[1] the oldest uops that are ready to the execution ports, each of which connects to one or more execution units. In workloads with many execution dependencies, the scheduler fills up faster (from dispatch) than it can drain by issuing operations. As a result, dispatch must stall, reducing performance. It is therefore desirable to minimize execution dependencies on out-of-order schedulers.

[1] Our definition is based on the AMD terminology [2]. Intel uses the reverse terminology for dispatch and issue [27].

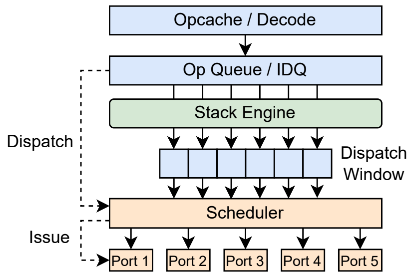


Figure 1: A generic architecture overview. A frontend (blue) *dispatches* operations from an Op Queue to the backend (orange) where a scheduler *issues* operations to execution ports.

2.2 Stack management

The benefits of out-of-order execution dwindle when the operations carry data dependencies. Therefore, compilers optimize code to utilize instruction-level parallelism whenever possible. However, some elements of the x86 architecture are notoriously hard to parallelize. In particular, many computer architectures, including x86, rely on a stack to pass arguments or as a scratch space when registers are needed for other calculations (i.e., registers are spilled to the stack). On x86, the stack additionally stores the return address of a function call. `RSP` or the Stack Pointer is a special register in x86 that holds the current top of the stack. The stack acts as a *Last In First Out* (LIFO) structure that is at the same time mapped into the memory address space and therefore also enables direct indexing. This leads to two distinct types of stack usage.

(1) Push/Pop. Traditionally, x86 uses `push` and `pop` instructions to insert into and remove from the stack. This requires dispatching two operations. For `push`, the stack pointer is decremented, followed by a write at `RSP`. For `pop`, the value at `RSP` is read, followed by an increment on the `RSP`. This method enables idiomatic variably-sized stacks at the cost of 2 uops[2]. `call` and `ret` instructions have similar semantics as `push` and `pop` instructions but they include an additional jump uop.

(2) Stack frame. Instead of changing `RSP` for every value stored and retrieved, a compiler may reserve a larger region of memory to directly index. This is used for stack-local variables, in particular, when the working set of a function does not fit into the registers. To provide uniform access to parameters and locally defined variables, the compiler first subtracts the size of the required storage space from `RSP` and then writes to addresses relative to `RSP` with a constant offset for any entry. To unwind, `RSP` is either incremented or overwritten by a value previously stored in the `RBP` register (i.e., the Base Pointer of the current stack frame). This reduces the number of operations to one uop per value saved, plus one setup instruction.

In out-of-order CPUs, stack frames can provide notable performance improvements compared to `push` and `pop` instructions. `push` and `pop` read and modify the `RSP`, resulting in an execution dependency on every instruction. Using stack frames reduces the dependency to just one operation, enabling parallel execution of multiple store and load operations. Despite the apparent drawbacks of `push` and `pop`, they are still used by default due to two main reasons. First, `push` and `pop` encode into fewer bytes than a `mov` with a memory reference, reducing code size and instruction cache use. Second, and perhaps more importantly, their drawbacks made them an early target of optimizations which we discuss next.

[2]Depending on the implementation, the address calculation and store are 2 distinct operations for a total of 3 uops.

2.3 Stack pointer tracking

Given the prevalence of `push` and `pop` instructions and their dependency-bound execution time, it is beneficial to resolve and eliminate their dependencies in the frontend before dispatching the uops to the backend for execution [8].

In a typical application binary interface (ABI), applying an offset from a register to the stack pointer is generally avoided [39]. `push` and `pop` satisfy this as they perform subtraction/addition operations on the `RSP` with a fixed immediate. These operations are also typically *balanced*: when a scope or call frame ends, the `RSP` points to the same location as when it entered. Memory accesses to stack locations typically are constant-value relative to the stack pointer, e.g. `mov rax, [rsp + 0x8]`, or the frame base pointer (`RBP`). In x86, this relative offset (`0x8`) is called *displacement*.

Given that both the modifications and the reads of `RSP` depend only on an immediate and the `RSP` itself, Bekerman et al. [8] propose a stack pointer tracking technique that, instead of emitting expensive backend updates, tracks an immediate offset value in the frontend which is used to add to or subtract from the displacements of subsequent operations before the dispatch. For operations that require backend resources (e.g., adding a register value to `RSP`), or complex operations like 3-operand `lea` that are expensive to implement in the frontend, the tracked Δ`RSP` needs to be *synced* back to the backend. The sync operation applies the current tracked offset to the backend `RSP` value, ensuring that the next operation operates on the architectural value of `RSP`. Because the sync operations are relatively rare in commodity code, frequent updates (and, in turn, execution dependencies) of the `RSP` could be avoided. This optimization was first implemented in the Pentium M microprocessor [21], and both Intel and AMD mention a *stack engine* in their software optimization manuals [2, 27]. Agner Fog's optimization manual [18] discusses the original implementation in additional detail. We provide a concrete example of the stack pointer tracking mechanism and its benefits, based on the `strtok_r` function in glibc 2.35 [20], shown in Listing 1.

Traditional `push`/`pop` tracking. Most modern microarchitectures support stack pointer tracking for `push` and `pop`. The operations of `push` (usually, 1 ALU operation for subtraction and 1 store) get optimized into performing the ALU operation in the frontend, leaving only a single, independent store for the backend. Therefore, all `push` operations in this example are independent and performance is bound only by the dispatch width and store ports. However, the `sub rsp, imm` is not supported in most architectures, and would hence require performing a sync operation that applies the offset tracked in the frontend (`dRSP` in Listing 1) to the backend. The same applies at the end of the stack frame, where the `add` introduces a dependency, but the `pop` operations are optimized again.

Extended immediate-based tracking. Our analysis over a wide range of architectures, discussed in Section 5, shows that immediate-based updates of the `RSP` are only supported in rather recent architectures, such as AMD Zen 5 and Intel Golden Cove. This feature

```
push r13      ; dRSP -= 8; mov [Brsp + dRSP], r13
mov r13, rdx  ;
push r12      ; dRSP -= 8; mov [Brsp + dRSP], r12
mov r12, rsi  ;
push rbp      ; dRSP -= 8; mov [Brsp + dRSP], rbp
push rbx      ; dRSP -= 8; mov [Brsp + dRSP], rbx
mov rbx, rdi  ;
sub rsp, 8    ; dRSP -= 8;
//...
add rsp, 8    ; dRSP += 8;
mov rax, rbp  ;
pop rbx       ; mov rbx, [Brsp + dRSP]; dRSP += 8
pop rbp       ; mov rbp, [Brsp + dRSP]; dRSP += 8
pop r12       ; mov r12, [Brsp + dRSP]; dRSP += 8
pop r13       ; mov r13, [Brsp + dRSP]; dRSP += 8
ret           ; mov rip, [Brsp + dRSP]; dRSP += 8
```

Listing 1: `strtok_r` preamble and epilogue code (glibc 2.35). Stack pointer tracking offloads the changes on `RSP` to a frontend register, enabling ILP on the memory operations.

fills the gap of stack pointer tracking in the traditional implementations by supporting immediate-based tracking on `add rsp, imm` and `sub rsp, imm`, which generalizes the special case of an 8-byte change from `push` and `pop`. This enables the stack pointer tracker to handle an entire stack frame without execution dependency, saving on both operations dispatched and scheduler slots. As a result, tracking continues through most function call chains, making the ΔRSP a potentially long-living value.

3 Motivation and Overview

As we described in Section 2.3, the stack engine must hold a state to keep track of the current ΔRSP offset. Any stateful hardware component bears a potential security risk, either through leaking secrets from a prior victim execution or through carrying attacker-controlled state into a victim execution. To better understand the security implications of the stack pointer tracking, we first have to find out how the stack engine operates internally in existing CPUs:

Challenge 1: Reverse engineering

How does the stack engine work and which signals does it emit?

⟶ *Section 5*

The stack engine is not described in detail in the published architecture manuals. Reverse engineering this particular microarchitectural component is a challenging task due to its nimble nature: by design, the stack engine is only affected by the instruction stream and itself. As a result, we cannot use typical runtime-dependent values to analyze the stack engine, but must produce x86 code on the fly. By carefully crafting particular instructions, we find the depth and conditions under which the stack engine operates and synchronizes with the backend on various Intel and AMD microarchitectures. Our next challenge is building attack primitives using these findings:

Challenge 2: Primitives

Which primitives can we craft using the signals created by the stack engine to transmit information?

⟶ *Section 6*

Using the insights from our reverse engineering, we design three primitives that enable information to be leaked from the stack engine under different microarchitectural conditions. In particular, our Sync+Reload primitive allows us to find balanced sequences of `push` and `pop` instructions in different security domains, and our Prime+Sync+Probe primitive enables us to observe the state of the stack engine even in microarchitectures that feature a very large ΔRSP. Our next challenge revolves around showing exploitation using these primitives:

Challenge 3: Exploitation

How can we use the primitives to leak information in different scenarios?

⟶ *Sections 7 and 8*

As a first step, we show how to transmit information between two functions that architecturally cannot communicate (i.e., a *covert channel*), and in a second step, we aim to leak information from a non-cooperating function (i.e., a *side channel*). We demonstrate that a side channel leaking stack depth enables attackers to observe results based on a current *or previous* operation that depends on the stack depth, *without relying on cache attacks*. Our stack engine attack on a JSON parsing library and a protobuf implementation leaks the structure of the contained data, compromising sensitive information like distinguishing patients based on the structure of their medical records.

4 Threat Model

We assume a system with programs that are free of software vulnerabilities. The aim of the attacker is to exploit the stack engine to enable unauthorized communication (i.e., the covert-channel scenario) or leak sensitive information from the target programs (i.e., the side-channel scenario). While covert channels enable characterizing the leakage rate of a new channel, the more interesting scenario is the side channel. We assume an unprivileged attacker aiming to leak sensitive data from a victim process. In our example, the victim uses in-process isolation (e.g., protection keys [52]) and performs secret-dependent stack operations, which we show can leak via the stack engine. In-process isolation is well-suited for Function as a Service (FaaS), where customers run code (i.e., functions) without managing servers [47]. The cloud provider handles hosting, serving, and scaling. Current FaaS systems rely on process or VM isolation, incurring significant overhead. Recent proposals reduce this overhead by using in-process isolation between tenants instead [33, 36, 62].

5 Reverse Engineering the Stack Engine

We provide a detailed reverse engineering of the stack engine in modern x86 CPUs from Intel and AMD. Table 1 shows the details of the CPUs that we analyze. We evaluate all AMD Zen generation CPUs (from Zen 1 to the most recent release of Zen 5), and four Intel CPUs including Sandy Bridge, Kaby Lake, and Alder Lake P- and E-Core CPUs. Given the available documentation in software optimization guides [2, 27], we know that the stack engine supports (tracks) implicit `RSP` modifications (`push`, `pop`, `call`, `ret`) in all the aforementioned CPUs, and these `RSP` modifications are used when calculating displacements on stack uops. Additionally, unsupported

Table 1: CPUs investigated from Intel and AMD.

Microarch.	Name	Model	Year
SNB	Sandy Bridge	E5-2609	2011
KBL	Kaby Lake	i7-7700	2016
ADL GLC	Alder Lake (P)	i7-12700K	2021
ADL GRT	Alder Lake (E)	i7-12700K	2021
Zen 1	Summit Ridge	1600X	2017
Zen 2	Matisse	3600X	2019
Zen 3	Milan	7413	2021
Zen 4	Phoenix	7840U	2023
Zen 5	Granite Ridge	9900X	2024

uops trigger a sync when ΔRSP is nonzero. Interestingly, we validate that the recent AMD Zen 5 CPU supports explicit RSP modifications like add and sub uops on the RSP, and find that Intel Alder Lake and later also support a similar scheme.

Briefly, we aim to answer the following questions in this section:

(1) Do push and pop instructions create an observable frontend bottleneck due to stack engine's uops? (Section 5.1)
(2) How do different microarchitectures handle unsupported stack uops? (Section 5.2)
(3) How do different microarchitectures implement the sync uops? (Section 5.3)
(4) What is the stack depth that different microarchitectures support? (Section 5.4)
(5) How do different microarchitectures differ in the stack operations that they support? (Section 5.5)
(6) Where in the pipeline does the stack engine modify the instruction stream? (Section 5.6)
(7) How does the stack pointer tracking interact with speculation? (Section 5.7)

Answering these questions allows us to create a model of the stack engine for different CPUs (Section 5.8). We use this model and our reverse engineering results to study the security implications of stack engine in the subsequent sections. For the experiments in this section, we rely on performance monitor counters (PMCs) to gain insights about the implementation details of the stack engine. Using these insights, we later use the unprivileged `rdpru` instruction for our leak primitives, as we discuss in Section 6.

5.1 Stack engine uops in the frontend

We want to see if the stack engine provides a similar performance for push and pop instructions as stack frames (as described in Section 2) or if there still remains a bottleneck. Specifically, since the frontend is now performing the stack pointer offset calculations, are there any observable contentions that we can exploit?

We test the throughput of stack engine-enabled instructions by comparing the throughput of push to the throughput of a store to the same memory address, but relative to a different register (r10). If there was a bottleneck on the frontend calculations, the version using push would perform worse than the implementation using stack frames (cf. Section 2). We show the results in Figure 2, highlighting no significant change in execution time on any of the tested microarchitectures.

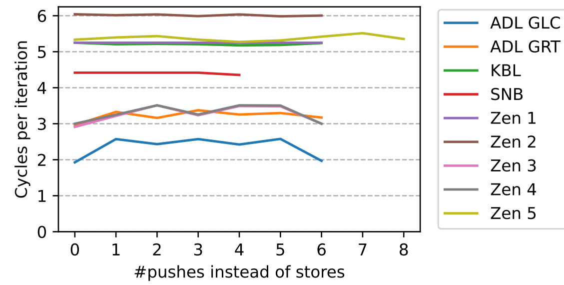


Figure 2: Cycles per iteration when filling the dispatch window with varying fractions of stores and stack pushes in different microarchitectures. push does not slow execution compared to a relative store on all tested microarchitectures.

```
for r in 0..8
mov r8, rsp; save the RSP
iterate
 push rax; Decrease the RSP
 .rep r:
  or rsp, rsp
 mov rsp, r8 ; Restore the RSP
```

Listing 2: Testing the number of sync uops: the first or must sync: x86's or sets the parity flag. However, subsequent ors may not dispatch if a ΔRSP of zero avoids additional uops.

> **Observation 1: push/pop throughput**
> The stack engine rewrite capacity matches or exceeds the backend memory uop capacity.
> ⟶ *Section 5.1*

This observation shows that we cannot leak information from the stack engine in the frontend directly.

5.2 Handling unsupported uops

Not all uops are supported by the stack engine. In particular, the frontend cannot support all possible uops applied to RSP such as stack alignment (`and`) or stack switching (e.g., due to a context switch) on all tested microarchitectures. In uops that are not supported but read the RSP, the ΔRSP must be applied (added) to RSP to ensure correctness before the uop. An optimization opportunity is that once ΔRSP is zero, no further syncing is required [2, 27].

We test these cases using the code in Listing 2. Note that the choice of uop does not matter as long as it is an uop that is not supported by the stack engine and references the RSP either as source or destination register.

The results in Figure 3 indicate that microarchitectures across both Intel and AMD dispatch an additional uop on the first modification, but subsequent uops do not incur a penalty. This is visible as a bend after handling 1 unsupported uop, where all architectures except Zen 5 dispatch one additional uop (the `or`) per iteration. However, the first uop needs to apply the offset first. Zen 5 is a notable exception to this rule: each of the `or` instructions are dispatched as 2 uops. This indicates that, contrary to the documentation [2], the stack pointer tracking does not turn off if ΔRSP is 0, but continues dispatching a sync uop (now simply an addition by zero). While this is a surprising change compared to Zen 4, it aligns with other

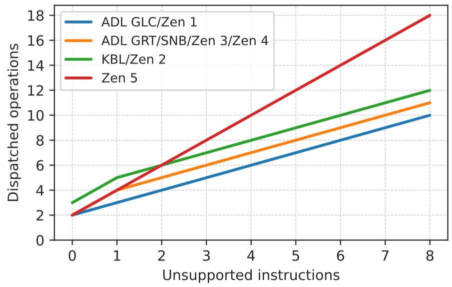


Figure 3: Reset uops across different microarchitectures. Each additional modifying instruction increases the number of uops by one. However, going from no modifying to one uop adds an additional uop. ADL GLC and Zen 1 miss this extra step and Zen 5 unconditionally dispatches 2 uops per step.

design choices of Zen 5 that remove special-case optimizations and rely on the wide dispatch to handle these cases instead [42]. Additionally, this indiscriminate dispatch implies that we cannot detect if a sequence left ΔRSP with an offset on Zen 5. Golden Cove and Zen 1 do not even exhibit a single additional uop. We believe this to be a performance counter bug as the uop is counted on Zen 2 and Gracemont.

Observation 2: Unsupported uops

Unsupported uops on ΔRSP trigger a sync uop dispatch before dispatching the uop itself. All architectures except Zen 5 elide dispatching an additional uop if ΔRSP is 0.

⟶ *Section 5.2*

As a result, we can observe if the current state of the stack engine is active on most architectures, excluding Golden Cove and Zen 1 due to performance counter inaccuracies, and Zen 5 due to the unconditional sync uop.

5.3 The execution of the sync uop

To see the additional sync uop, discussed in Section 5.2, from another thread, we must know which specific resource this uop executes on. Based on existing documentation and the original paper described in Section 2, the logical conclusion would be for the sync uop to add the current ΔRSP to the backend RSP register, i.e., an `add RSP, dRSP`. We test this by observing port-based performance counters on architectures that support them. For the Zen architectures and Intel efficiency cores, however, the port-based counters are not supported. We handle these cases as follows using the code in Listing 3, summarized in Table 2:

Zen 1 to Zen 4. These microarchitectures have split schedulers per execution port (or groups of two execution ports), where each scheduler can only accept a single uop per dispatch. When an uop does not fit into a scheduler, dispatch aborts and continues dispatching in the next cycle. This increases execution time that is observable with precise counters. Due to the execution dependency between the push and the `or` instruction, the code in Listing 3 executes in at least 2 cycles. When inserting additional ALU uops between the push and `or` (`fill(NUM, ALU_OP);` in Listing 3), we see that we can fit at most 2 independent ALU uops before an additional delay in the execution time due to overcommitting the dispatch window.

```
push rax            ; decrease the RSP
fill(NUM, ALU_OP)   ; fill ALU ports
or r9, rsp          ; trigger RSP sync
```

Listing 3: Assembly to test the type of sync uop. We expect 1 (or) + NUM (fill) ALU uops if the sync does not issue to the ALUs, and an additional uop otherwise.

Table 2: Observing sync uops in different microarchitectures using the code in Listing 3.

Microarch.	Method	Result
SNB / KBL	PMC	2 uops on ports P015(6)
ADL GLC	PMC	2 uops on ports P0156,10*
ADL GRT	EX cap	Same execution time up to NUM=3
Zen 1	EX cap	Same execution time up to NUM=2
Zen 2	EX cap	Same execution time up to NUM=2
Zen 3	EX cap	Same execution time up to NUM=2
Zen 4	EX cap	Same execution time up to NUM=2
Zen 5	EX cap	Same effect as an ALU uop

*: Only if not in the same dispatch window

Zen 5. Zen 5 has a unified scheduler and does not stall even when dispatching 8 uops in the same window. Therefore, we simply compare the backend stalls induced by overcommitting ALU uops to the stalls induced by the stack engine. The results show that the stack engine stalls behave identically to ALU uops and therefore execute on the same ports. We can confirm this by observing a matching number of ALU-token and backend-token stalls when either dispatching an `or r9, rsp` or two independent ALU uops.

Observation 3: Sync uop type

Sync uops dispatch as ALU uops indistinguishable from an `add` to RSP and issue to an ALU port.

⟶ *Section 5.3*

Because the uop executes as an additional uop on any suitable ALU port, they become observable by an SMT thread contending the same resources.

5.4 The depth of the stack engine

We want to understand when ΔRSP over- or underflows in the stack engine since this must trigger a sync uop, similar to when RSP is used as either a source or destination register. That is, when ΔRSP is continuously increased (overflow) or decreased (underflow), the ΔRSP must not simply wrap, but apply the offset to the backend. We therefore test the number of stack pointer increasing or decreasing uops dispatched before the offset is applied in the backend. We run the code shown in Listing 4 and compare the theoretically executed number of instructions with the observed number of dispatched uops according to performance counters in Figure 4. If the number of dispatched ops increases over the theoretically calculated value, we have found a synchronization point where the stack engine inserts a flushing sync uop into the dispatch window to avoid overflow. By only looking at the unexpected uops, we will see a staircase pattern where each step indicates the number of sync uops required for the given number of dispatched ΔRSP-modifying

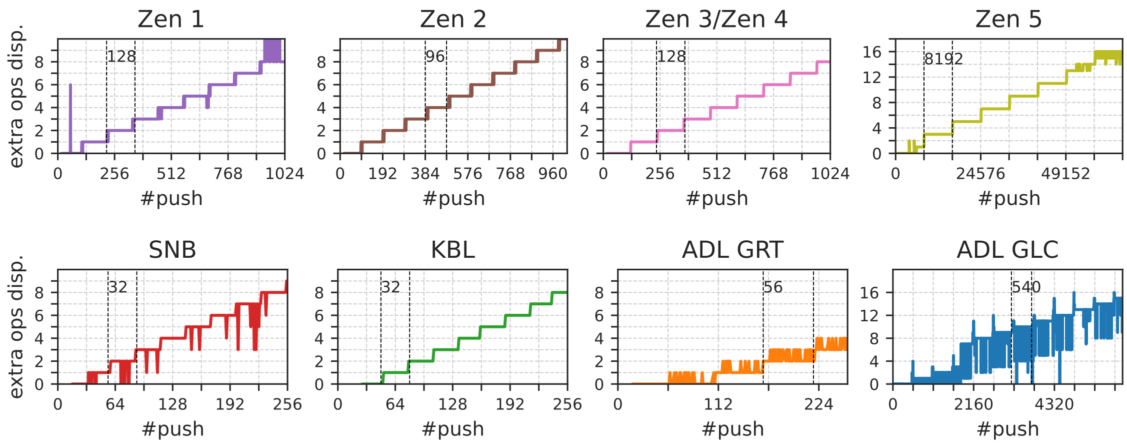


Figure 4: Testing the depth of the stack engine in different microarchitectures using pushes of 8 bytes. The y-axis denotes unexpected (additional) uops according to performance counters. Due to performance counter inaccuracies, Intel CPUs exhibit a jitter within a window of 6 ops. We mark the upper bound of the stack pointer tracker; some cores issue a sync earlier if resources are available (e.g. Kaby Lake syncs after 27 pushes). Zen 3 and 4 resulted in identical measurements.

```
xor rsp, 0 ; Reset the tracking
iterate
  push rax
```

Listing 4: Decrease stack pointer until ΔRSP overflows.

uops. We show that all tested microarchitectures exhibit a staircase pattern, although the step width varies across microarchitectures. The oldest architectures in our tests (Sandy Bridge and Kaby Lake) only support a tracking of 32 uops. Zen 1, Zen 3 and Zen 4 all dispatch the sync uop after around 128 pushes, with Zen 2 exhibiting an outlier behavior at just 96 uops. Zen 5 and the Alder Lake Golden Cove CPU track a relatively high 8192 and 540 pushes, respectively. In contrast to the performance core, the Alder Lake efficiency core dispatches a sync uop after every 56 uops.

Zen 5 differs from the other tested microarchitectures by issuing 2 additional uops per sync uop. We attribute the high jitter in the results of the Intel machines to the performance counter inaccuracies. Notably, the counter value swing of 5 uops on Golden Cove aligns with the dispatch- and retirement window size, poisoning the measurements.

Observation 4: ΔRSP size
The ΔRSP size varies across microarchitectures, but all microarchitectures issue additional ops on ΔRSP overflow.
⟶ Section 5.4

This indicates that all CPUs have stack-depth sensitive behavior. Knowing the stack depth which triggers the sync uop in different CPUs enables new leakage primitives as we discuss in Section 6.

5.5 Support for add/sub instructions

Any function that uses local variables in the stack will reserve a stack frame with `sub rsp, imm`. If this `sub` results in a sync uop, it would not be possible to leak information across a generic call stack. We design a simple test to check if the stack engine in different microarchitectures support RSP add/sub with an immediate by measuring if these instructions issue a sync uop. This is documented in the AMD manuals [2] for Zen 5, but Intel does not disclose whether their stack engine supports add/sub instructions. We find that add/sub is not supported on any AMD microarchitectures prior to Zen 5. On Intel, we see that the stack engine on the recent Golden Cove microarchitecture also supports add/sub instructions. Based on our results, Golden Cove has a 12-bit tracking register, supporting adds of up to 10 bit. The add/sub exhibits the same depth as we observe in Section 5.4 (a maximum of +65535/-65536) tracked with increments of at most 12 bits on Zen 5.

Observation 5: add/sub RSP
Recent CPUs (Zen 5, Golden Cove) support add/sub immediate operations on the RSP, eliding sync uops for common stack frames.
⟶ Section 5.5

Supporting add/sub reduces the syncing rate, allowing the stack engine to track a longer window and further reduce the number of executing uops. However, this also keeps offsets contained in the ΔRSP alive for longer, carrying potentially secret values across deeper call stacks.

5.6 Location of stack engine

The stack engine likely acts during dispatch, inserting or filtering sync uops into or from the uop stream. We believe that the performance counters count uops at dispatch based on the source selection for the current cycle which does not accurately reflect whether a uop is actually contained in the uop cache. This is supported by the statement in the optimization manual that uop cache and decoders are mutually exclusive per cycle [2]. We verified this by observing that our side channel also works when only considering the ops supplied by the uop cache, as well as by disabling the opcache, which did not prevent the attack from working. Additionally, we do not observe opcache invalidation when calling with different

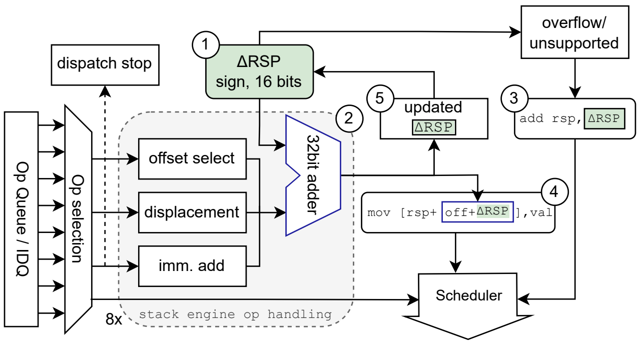


Figure 5: Overview of the observed stack engine mechanisms. A ΔRSP register tracks the current offset ①. Supported uops on RSP calculate a new offset ②. On overflow, an additional operation dispatches to sync the architectural RSP ③. If necessary, ΔRSP is added to the displacement field of dispatched uops ④. After each cycle, the ΔRSP register is updated ⑤.

ΔRSP offsets. Thus, we confirm that the stack engine operations are inserted after Decode and before Execute on all investigated architectures.

Observation 6: Location

The stack engine is located at Dispatch. Its behavior is independent of opcache and it does not invalidate opcache entries.

⟶ *Section 5.6*

5.7 Stack engine under speculation

The stack engine is not predictive in nature and as such does not open up new transient execution windows. However, it might still leak information under speculation. We have analyzed two main scenarios. First, we checked whether the stack engine offset is reset when the CPU corrects a branch misprediction. We find that the offset is reset to zero on Zen 3-4 while Zen 5 appears to retain an offset. We were not able to conclusively determine the effect on the other architectures due to excessive noise introduced by the misspeculation. Second, we aimed to detect stack engine sync operations that occur only on the speculative path that are later squashed. Using PMCs, we confirm that sync operations are indeed also observable under transient execution on Zen 3-5. An attacker might theoretically combine this behavior with a classic indirect branch target injection to build a call-depth disclosure gadget in a cross-thread attack. However, we note that such an attack would only slightly expand the capabilities of a cross-thread attacker.

Observation 7: Speculation

The stack engine does not introduce new speculation primitives. Sync operations can be dispatched speculatively and are squashed when mispredicted.

⟶ *Section 5.7*

Table 3: Experimentally observed properties of the stack engine in different architectures

Architecture	Capacity	Sync with ΔRSP = 0	Sync on add
SNB/KBL	32	○	●
ADL e	56	○	●
ADL p	540	○	◐
Zen 1	128	○	●
Zen 2	96	○	●
Zen 3/4	128	○	●
Zen 5	8192	●	◐

○ : No, ● : Yes, ◐ : partially (>10bit in ADL p and >12bit in Zen5)

5.8 Summary of the experiments

We show a schematic overview of the stack engine on Zen 5 in Figure 5. Other microarchitectures (except Golden Cove) do not support the immediate add component, and dispatch width, ΔRSP size and other details may differ. All microarchitectures feature a ΔRSP register ① and a stack engine logic ② in the frontend. Whenever ΔRSP overflows or is close to overflowing, or when an unsupported uop accesses RSP, it syncs by applying the ΔRSP into an `add rsp, imm` uop before dispatching the current uop ③. When a supported instruction referencing RSP is seen, the current ΔRSP is embedded into the displacement field of the instruction ④. Only the last instruction in each dispatch window updates the architectural ΔRSP ⑤.

We list the details of the stack engine in each architecture in Table 3. Even though a stack engine exists on all tested microarchitectures, the implementations vary, sometimes even in the same family of CPUs. Older implementations are particularly tuned for 32-bit ABIs, where `push` and `pop` are used for argument passing. Evolutionary CPUs increase the tracked depth, but the prevalence of `add` and `sub` on the `RSP` make larger tracking sizes feasible only when also supporting complete stack frames. While this optimization reduces execution dependencies, it also encodes information about the control flow in a stateful, hidden register in the stack engine. We discuss how we can build leakage primitives from the information that we uncovered in Section 6.

5.9 Stack engine in other architectures

RISC architectures with fixed-size instructions avoid single-value stack pointer updates in their ISA and thus lack a stack engine.

ARMv7 (32-bit) uses special register-saving routines aliasing to push (Store Multiple, `STM`), which take a register bitfield and preserve a fixed order. This reduces inter-instruction dependencies, eliminating the need for stack pointer tracking.

ARMv8 (64-bit) uses load/store pair instructions (`ldp`/`stp`) for stack operations. Typically, a pre-decrement store (e.g., `stp x29, x30, [sp, -48]!`) adjusts `sp` before writing to it. Subsequent stores use `sp`-relative offsets, removing the need for a stack engine.

RISC-V follows the stack frame convention: `sp` is decremented at frame start, and accesses are made relative to `sp`. We examined *BOOM* [65] and *XiangShan* [59], two popular OoO RISC-V cores—neither employs a stack engine.

```
void fn(int input) {
  if (input) {
    rbp = rsp;
    // ...
    rsp = rbp;
  } else {}
}
```

Listing 5: Unsupported uop causing a sync uop conditional on the value of `input`. This syncs the stack engine, causing a positive tracked offset when returning to the caller.

6 Stack Engine Leakage Primitives

As we demonstrated, the stack engine causes an additional uop in case of an unsupported uop (Section 5.2) and an additional uop on overflow (Section 5.4) on all tested architectures. Because this overflow happens within an offset of a few bytes around the capacities listed in Table 3, this provides a way of storing and retrieving non-architectural information. To achieve practical covert and side channels using the stack engine, we aim to build ΔRSP read and write primitives.

As discussed in Section 5, the stack engine operates on two inputs. The first input is the instruction stream that determines if the stack engine becomes active and how the RSP is read and modified. The second input is ΔRSP, which determines if there is an overflow or if a sync (Section 5.3) on unsupported uops (if ΔRSP $\neq$ 0) occurs.

Primitive #1: Direct Underflow. Given stack engine overflows, we can trigger a sync op by generating a deep call stack which will underflow ΔRSP. The inputs to this trigger are the stack depth of a called function and the pre-existing ΔRSP. Recursive functions are an example of such stack engine underflow triggers. We can determine the call depth of a recursive function as follows:

(1) The caller resets ΔRSP with the unsupported `xor rsp, 0`.
(2) The callee recurses n times, requiring 32 bytes per iteration.
(3) When the ΔRSP underflows (sufficiently decremented after $\Delta RSP_{max}/32$ recursions), an additional uop dispatches.
(4) On the return path, the ΔRSP might overflow (because we now add more than it can track), dispatching another uop. However, this depends on the specific unwinding and used offsets.

While two dispatched uops seem promising, this primitive has an important shortcoming: the uops dispatch during the execution of the callee. This means that the additional uop can only be observed if the caller already can observe the function call. Tools that could observe the sync uop would also be able to observe the control flow itself. Therefore, Direct Underflow is a weak primitive. For a more powerful primitive, we next attempt to return information to the caller.

Primitive #2: Sync+Reload. Due to the additional uops not dispatching when the ΔRSP is zero (Section 5.2), we can find balanced sequences of push and pop. Consider a case of a function conditionally resetting the stack pointer based on an input as shown in Listing 5. Unlike the normal underflow, this carries state back to the caller. When the caller knows the offset `off` of the stack pointer of this conditional call, it can push or sub for `offset` bytes, followed by an unsupported uop. If the ΔRSP was reset (i.e., `input` was `true`), the ΔRSP would be zero, and no additional uop would dispatch. Otherwise, an additional uop dispatches.

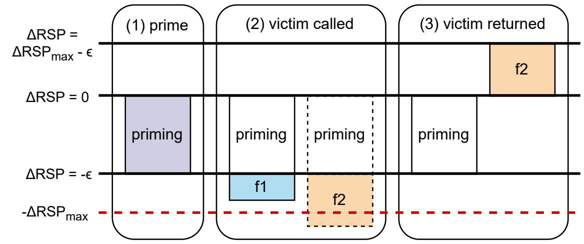


Figure 6: Prime+Sync+Probe primitive for a victim whose execution has variable stack depth (f1 vs. f2) depending on a secret. By priming ΔRSP with a predetermined value, an attacker can affect distinct ΔRSP states depending on the secret.

```
nop(64);         ; decouple from previous code
rdpru            ; Time and store result (omitted)
lfence
increase_drsp()  ; probing
                 ; 1-cycle dependency introduces here on overflow
decrease_drsp()  ; probing de-set
xor rax, rsp     ; syncs SE with backend
lfence
rdpru
```

Listing 6: `rdpru` probing code. The fenced sync operation before the final measurement is key to ensure that the stack engine syncs before the `lfence`.

Sync+Reload works on all microarchitectures from Table 1 but Zen 5. Zen 5 unconditionally dispatches a uop on unsupported uops (cf. Table 3). Additionally, Sync+Reload requires a function to conditionally operate on the RSP with an unsupported uop. We combine the best of the *Direct Underflow* and *Sync+Reload* in our last primitive.

Primitive #3: Prime+Sync+Probe. The previous two primitives rely on the instruction stream to trigger a sync uop at a given point. We now use the statefulness of the ΔRSP to our advantage: instead of starting at the neutral 0-offset, we *prime* ΔRSP to an arbitrary value. This enables us to control where, if at all, the underflow occurs when running the callee. When returning to the caller, ΔRSP is either unchanged (if no underflow occurred) or contains a larger value than when we started. We show this in Figure 6. Starting from the current RSP value, which is composed of a backend register bRSP and the frontend ΔRSP, we add an offset ϵ, sync the ΔRSP using an unsupported uop, and subtract the offset ϵ again, taking care to split ϵ into step sizes supported by the stack engine. This leaves us with a highly *negative* ΔRSP (1). When a victim conditionally uses a deeper or shallower stack, the ΔRSP conditionally underflows and requires syncing (2). When unwinding the stack, the ΔRSP either returns to the highly negative (primed) value or, if an underflow had occurred, to a positive value of $f2_{depth} - (\Delta RSP_{max} - \epsilon)$. When the caller then adds ΔRSP_{max} to the RSP, the ΔRSP overflows conditionally. We use this primitive in Section 8 to leak the structure of a JSON file by testing distinct priming values ϵ and finding the deepest value that the stack pointer assumes.

Unprivileged observer. Precisely observing dispatched uops typically requires PMCs, which need OS support and are not accessible in user mode. To observe the stack engine with an unprivileged observer, we use `rdpru` instead, which is available since AMD Zen 2.

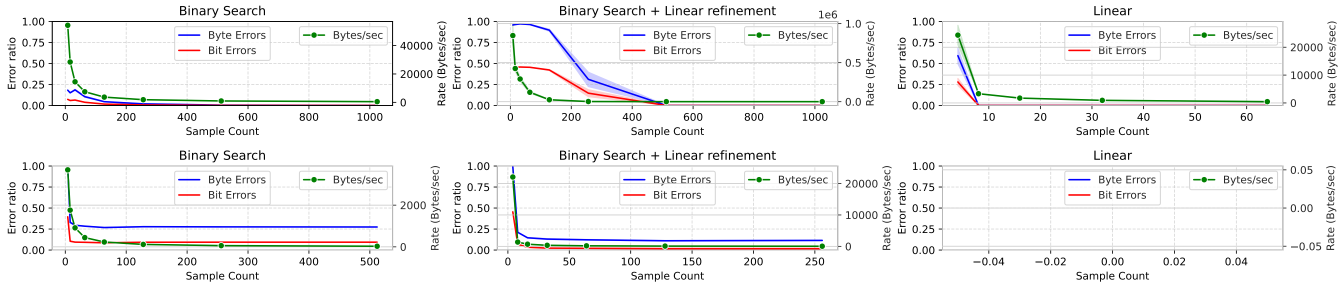


Figure 7: Single-thread covert channel results for encoding 8 (top row) or 16 bit (bottom row) values into ΔRSP. A larger encoding requires a longer probing phase, which makes it more susceptible to interruptions, resulting in linear scans not obtaining usable results. Encoding 8 bits achieves a higher rate, and the linear scan minimizes the error rate already after few samples.

Listing 6 shows the measuring sequence. Triggering an extra sync op before the last `lfence` is required because the stack engine operations otherwise do not stall the backend, resulting in the `lfence` not waiting for completion of stack engine operations. We use this primitive for our attack in Section 8.

7 Covert Channels Using the Stack Engine

Before we look into potential attacks, we craft a signal-passing covert channel, where sender and receiver are cooperating.

7.1 Same-thread leakage

We start by demonstrating a same-thread covert channel. Process isolation can provide security guarantees in a same-thread leakage threat model. However, process isolation incurs high performance overhead, motivating secure in-process isolation proposals, like CHERI [58] and Intel MPK [27], which reduce performance impact compared to process isolation. However, as we will demonstrate, non-architectural hardware optimizations like the stack engine produce potential security risks.

MPK. Zen 5 introduced Memory Protection Keys for Userspace (MPK, PKU) to the AMD domain. Unlike standard x86 memory protection, MPKs can be switched with a fast special-purpose register write without changing the current privilege level, providing tools for high performance isolation. Previous work like ERIM [52] propose frameworks built on PKUs to provide trusted and untrusted domains by scanning untrusted code for security-compromising instruction sequences that write to the PKU register.

Same-thread covert channel. We attempt to leak information from a malicious sender in the trusted domain to a malicious receiver in the untrusted domain, breaking the in-process isolation. We weaken the MPK threat model for the covert channel attack by placing a crafted transmission function in the trusted domain. However, we retain the rule that the untrusted domain is checked for any illegal accesses, including attempts to measure the execution of the trusted domain using performance counters. At the same time, we do allow measuring performance counters during untrusted code execution.

We use the Prime+Sync+Probe primitive to construct a channel with high transmission rate. In order to achieve a high rate, we encode more than one bit in the ΔRSP passed by the victim. In the attacker, we then probe for the encoded value by adding increasing offsets to the RSP and looking for the overflow sync operation.

Same-thread leakage results. In Figure 7, we show the single-thread covert channel effectiveness for Zen 5 when encoding a secret value in the ΔRSP and probing it in the calling function after a protection key switch. We report byte errors (how many bytes were incorrect) and bit errors to show the error distribution. The sample count determines the number of measurements per probed depth. When encoding a depth of 8 bits into the ΔRSP, linear search generally outperforms binary search, requiring just 8 samples and reaching a throughput of 3kB/s with negligible error. If higher error rates are tolerable, binary search brings off-by-one measurements, but reaches up to 48kB/s of throughput. Encoding more values into the ΔRSP is possible on Zen 5. However, the larger search space causes both a performance hit and less reliable results, which can be attributed to mispredictions polluting the measurements. With linear search the execution time would prove too long, resulting in frequent interruptions and broken measurements. By combining a rough binary approximation, followed by a linear refinement on a more precise region, the error rate stabilizes to 1/8 bytes affected by off-by-one errors and a leakage rate of 300B/s.

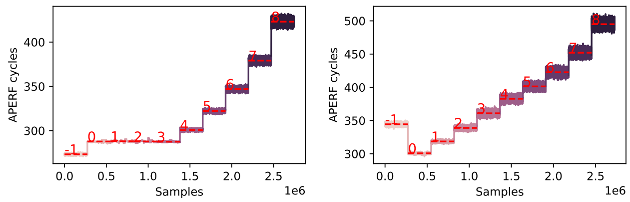
7.2 Cross-thread leakage

Next, we build a covert channel to observe the stack engine uops across threads (i.e., the cross-thread leakage mode). Prior work [3, 23, 25] has shown that port contention can be used to extract information from SMT cores. However, we show limitations of traditional port contention when observing stack engine uops, and we devise a new approach tuned for *execution-independent*, *single-cycle*, *high-throughput* uops, as required for the stack engine.

Port contention for the stack engine. Compared to typical port contention, observing the stack engine's sync uop on the execution pipes is challenging, mainly because:

(1) the sync uop may issue to one of many ports;
(2) the sync uop runs in a single cycle;
(3) the sync uop cannot build dependencies.

Prior port contention approaches [3, 23, 25] are more precise when observing instructions that go to a single port (e.g. `mul` or `div`). These instructions also have a higher latency. Attempting to apply existing port contention approaches to detect a single add yielded negligibly weak signals. To demonstrate our refined port contention, we use Zen 5 as an example, but similar effects were observed on Intel Kaby Lake. In addition, in the SMT mode on Zen 5 with another thread running, we observe stalls only if the sum of



(a) Naive **(b) Tuned**

Figure 8: (a) Naive port contention is not sufficient to observe stack engine sync uops. (b) Delaying the execution of 6 ALU uops using a dependency reveals the missing uops. -1 indicates a pausing SMT thread. The y-axis shows actual core frequency cycles provided by `rdpru`.

uops reaches 12, double the ALU ports. We deduce that SMT threads dispatch in alternating fashion, with a full dispatch window being available for a thread on every 2nd cycle.

We attempted to use the code from Portsmash [3] for ALU contention but we were unable to observe single additional dispatched uops on Zen 5 (we adjusted the code to reflect the 6 ALU ports in Zen 5). Figure 8a shows the results for Portsmash (naive) where the observer/receiver thread always dispatches 8 ALU uops in its own dispatch window. Stalls are only visible if the contender/sender thread dispatches at least 4 ALU uops in its dispatch window (i.e., when the sum of ALU uops reaches the double ALU capacity which is 12). However, this contention is not sufficient to observe single single-cycle ALU uops, as is the case for the stack engine uops.

We tune our port contention to have as many single-cycle uops as possible execute in the execution cycle where the contending thread dispatches its uops. This should result in increased contention for single-cycle uops. We achieve this on a per-dispatch window basis by using one uop on which the other uops in the dispatch window depend, pushing them to the next execution cycle. Our improved port contention results in a distinct pattern as shown in Figure 8b. We can now clearly observe independent single-cycle operations. Using performance counters, we further observe an increase in dispatch stalls of the measuring thread, hinting towards a fairness protocol limiting dispatch rates when one thread is overcommitting uops.

Covert channel. We build a covert channel based on single-cycle ALU uops, using `nop` (which does not execute on the ALU) to encode a zero, and a stack engine sync uop to encode a 1. The channel leaks a bitstring with a rate of 12.5 kbit/s and accuracy of 98% using this approach for an alternating bit pattern. Due to small timing differences, transmitting byte-aligned values (like strings) is less reliable and requires careful sequence alignment. However, we still achieve a rate of 111 B/s.

8 Side-channel Attack on the Stack Engine

Having found covert channels, we attempt to construct side channel attacks by removing the control over the sender. That is, we now need to find potential victim code that we can probe for a secret.

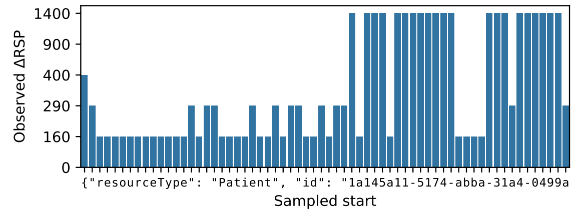


Figure 9: A trace of a JSON object retrieved from a stale ΔRSP value which reveals the start and type of JSON values.

8.1 Leaking from cJSON

cJSON [19] is a small JSON parser and writer. In our implementation, we use it to parse patient data. Internally, it uses a recursive function `parse_value` that parses the constant strings `true`, `false`, and `null`, and calls a corresponding handler for a number, a string, an object, or an array. The handlers for object and array call `parse_value` recursively to process their contained elements. Notably, errors resulting from missing termination characters like `}` and `]` are detected only after already entering the recursive pattern.

We assume a system that provides a secure computation environment for patient health data in the FHIR format [6] using in-process isolation. Patient data is provided through an interface to query the underlying JSON-formatted dataset, but a filter prevents the function from returning elements of the dataset that should not be accessible to the data processor. The data processor is allowed to run untrusted code that is scanned for secrecy-breaking accesses similar to prior work [52]. That is, accesses that change the `PKRU` register and timing of secret functions during their execution are checked for and the code is rejected in such instances. Timing instructions, however, are allowed to enable debugging and performance evaluation.

Dataset. We combine the FHIR dataset into a single object per patient, containing linked documents like current medications and conditions. Using the medium dataset, we obtain 120 patient files, ranging in minimized sizes between 312KiB to 42MiB.

Feature exfiltration. We show that we can exfiltrate information through ΔRSP without having access to the memory or concurrent timing functions. We do so using the Prime+Sync+Probe primitive combined with the `rdpru` based observer. By parsing a JSON file at different starting points, the cJSON library inadvertently encodes the structural information about the *deepest* reached depth into ΔRSP. This includes invalid starting tokens (which cause an immediate abort), numbers (which call into deep function parsing chains), and objects and lists, which deepen on every recursive encounter during the parsing. We use this information to recover a *trace* of the parsed data. Figure 9 shows a trace obtained using this approach: when encountering numbers, a deep function is called, and invalid starting points exhibit a shallower depth than valid starting characters.

Feature matching. To match an obtained trace to a known or previously observed structure, we use simple time-series matching [17]. Because we probe each individual byte, we do not suffer from time-shifts which require more complex methods; comparatively simple euclidean distances or cosine similarity suffice.

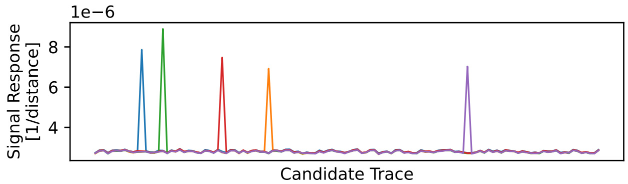


Figure 10: Signal response of an euclidean distance filter on 5 traces collected using the Prime+Sync+Probe primitive across isolated in-process domains against 120 candidates.

Results. Using the Prime+Sync+Probe primitive, we are able to profile 450 starting bytes for a scanned region of 16 KiB per second, with a high variation depending on the complexity of the parsed JSON structure and the resulting control flow. We achieve a 100% success rate classifying traces based on a 256 KiB prefix, as we show in Figure 10.

8.2 Leaking from protobuf

To confirm that cJSON is not the only library affected by potential attacks, we tested other libraries as well. In general, all libraries which use data-dependent recursion are suitable targets. protobuf [9, 22] uses recursion when serializing and deserializing objects to and from a bytestring. Similarly to JSON, there is no header that contains information about the structure of the data, and the structure needs to be reconstructed by consuming a stream in order. While these are good conditions for our attack to work, the default compile flags enable vector instructions which in turn access the stack relative to `RSP`, preventing attacks similar to the one shown for cJSON because the workload itself would frequently reset the stack pointer with unsupported operations (cf. Section 5.2). However, protobuf creates functions named `*__get_packed_size` to determine the required size of an object buffer to serialize a provided object. This family of functions has object-type dependent control flow and does not reset the stack pointer tracking, making it a suitable target for another attack. In this example, we calculate the required size of an (inaccessible) linked list. protobuf scans each element, and conditionally continues if the `next` member exists by calling the respective `element__get_packed_size()`. As a result, the length of the list is encoded into Δ`RSP` and can be retrieved in a way similar to the cJSON case.

Results. We test 25 randomly chosen list lengths between 0 and 100 and show the resulting observations in Figure 11. The plot shows a clear linearity of 96 bytes of stack usage per list element and a starting overhead of 32 bytes. We note that despite the relatively deep stack depth, the signal remains clearly visible. An attacker can therefore gain information about the data handled by the system.

8.3 ERIM

The described attacks also work when using ERIM [52] without stack switching (the default). Programs protected by ERIM also rely on MPK to provide in-process isolation. We verify our primitives with both PMCs and `rdpru` on protobuf [9, 22] using its `*__get_packed_size` methods, and cJSON under ERIM. Enabling stack switching mitigates the attack.

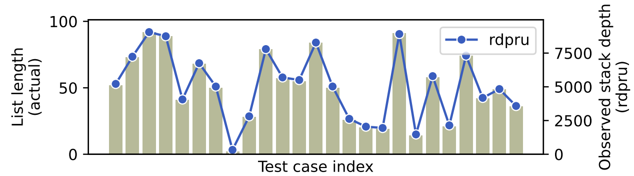


Figure 11: Observed stack depth when determining the required buffer size (`element__get_packed_size()`) using protobuf. The stack engine observation closely aligns with the actual list length at a constant factor of 96 bytes.

9 Mitigating Stack Engine Attacks

The stack engine side channel leaks secrets where there is a secret-dependent instruction stream. Hence, a comprehensive mitigation is *constant-time programming* [4]. While constant-time programming is effective, such programs incur high performance overheads [38]. *Execution balancing* is another approach that avoids the overheads of constant-time programming, and instead, it balances both sides of secret dependent executions [57]. Given our analysis, they need to consider the effects of the stack engine in order to provide security. For example, to balance secret dependent function calls, the mitigation must insert perfectly balanced unsupported operations to synchronize the behavior (e.g., `or rsp, rsp`). Also, our attack is mitigated by privilege transfers that write to `RSP`, either in hardware (e.g., SGX, VMs) or software (e.g., most OS syscall handlers when switching to the kernel stack).

While constant-time programming and (enhanced) execution balancing can protect against stack engine side channels, they require modifying the software. Section 9.1 discusses how we can mitigate our attack on existing software and hardware without any modification. Section 9.2 measures the introduced performance overhead of our proposed mitigation and Section 9.3 discusses how future CPUs can reduce this overhead. Finally, as discussed in Section 5.7, a standard constant-time program (i.e., architecturally constant time) can still leak information through the stack engine under speculative execution. In Section 9.4, we discuss the necessary considerations for Spectre defenses to preserve constant-time guarantees under speculation in the presence of stack engine transmissions.

9.1 Stack engine chicken bits

According to the AMD Software Optimization Guide [2], the functionality of tracking `add` and `sub` in the stack engine is controlled using `DECFG.StackEngineAddSubDisable`. Because the `DECFG` MSR is not documented, we scan it by toggling each individual bit and running a benchmark including the stack engine operations.

Zen 5. Running a sequence of `add rsp, 8` reveals that enabling bit 32 in the `DECFG` MSR causes 2 additional operations dispatched, compared to a run without bit 32 set. We believe that the chicken bit replaces the conditional synchronization event with an unconditional synchronization similar to what we observed in Section 5.3, issuing a sync on every dispatch rather than only on overflow or when Δ`RSP` is non-zero. In addition, when setting the bit 19, we observe 2 operations dispatched per `push`, which means they are not optimized by the stack engine.

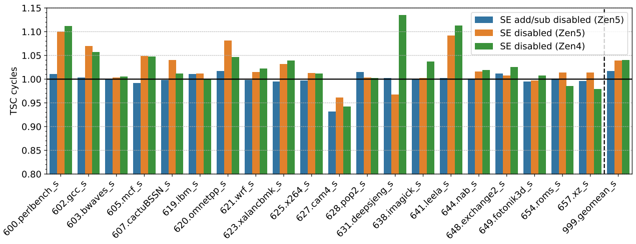


Figure 12: Normalized execution cycles of SPEC CPU2017 for three MSR configurations compared to the baseline with stack engine enabled: (1) Zen 5 with `add/sub` disabled, (2) Zen 5, stack engine disabled, and (3) Zen 4, stack engine disabled.

Zen 4. AMD Zen 4 does not officially support `add` and `sub` in the stack engine, and we observe consistent results. Setting bit 19 doubles the number of operations dispatched for `push` and `pop`.
Earlier generations. Prior generations of CPUs exhibited a bug in the stack engine where the offset would not correctly reset [1, 45]. The suggested workaround is to set bit 0 in the `DECFG` MSR.

9.2 Performance of disabling the stack engine

To evaluate the performance impacts of the stack engine, we run the SPEC CPU2017 [48] benchmark with reference inputs. We consider three configurations: (1) Zen 5 with the stack engine `add/sub` disabled, (2) Zen 5 with the stack engine completely disabled, and (3) Zen 4 with the stack engine disabled. Figure 12 shows the number of execution cycles, normalized to a stack engine-enabled baseline. Disabling the stack engine incurs a slowdown of 3.98% on Zen 4 and 3.94% on Zen 5, where disabling the `add/sub` support alone has a slowdown of 1.65%. We note that control-flow intensive programs like `perlbench` (a script interpreter) and `leela` are hit with a 10% slowdown while compute-intensive tasks like `bwaves` are virtually unaffected. This is a rather large performance penalty that prompts exploring the design of a secure stack engine architecture.

9.3 Future mitigations and stack engine designs

To mitigate the stack engine side channels in future CPUs without losing performance, the CPUs need to synchronize the stack engine more comprehensively at security boundaries. For example, a sync operation on the `wrpkru` instruction would prevent the attack at the MPK boundaries. Alternatively, a hardened design of the stack engine can randomly dispatch sync operations to prevent an attacker from observing a precise signal. Hence, the level of randomness needs to be tuned in a way that provides an acceptable security-performance trade-off. In addition, defenses that mitigate SMT contention must also carefully address stack engine leaks across SMT cores [11, 49, 50].

9.4 Spectre mitigations

As discussed earlier, stack engine operations can serve as secret transmitters under speculation. Hence, Spectre mitigations that rely on blocking potential transmitters must also account for stack engine operations. For example, prior work [64] aims to provide a comprehensive list of transmitters, and our findings show that stack engine operations should be included in such considerations, particularly for defenses like Speculative Load Hardening (SLH) [10]. These mitigations rely on unsupported stack engine operations (e.g., an `or`) to mask transient loads from the stack, which can result in variable timing dependent on the state of Δ`RSP`.

10 Related Work

Stack pointer tracking. At ISCA in 2000, Bekerman et al. [8] from Intel proposed a general technique to track register updates in the frontend (updates in the form of $reg \pm imm$) and resolve load addresses early at the frontend. The main use case that they demonstrate is the stack pointer tracking. Our reverse engineering shows that current CPUs implement a mechanism close to this work. Since we target stack engine, as a frontend optimization, we provide a summary of attacks exploiting CPU frontends.
Non-speculative frontend attacks. Similar to our work, many attacks have reverse engineered the units in the CPU frontend that can induce distinguishable behavior upon secret-dependent instruction streams [14, 16, 24, 30, 32, 34, 43, 46, 60, 63]. The main difference of the stack engine is that its secret-dependent operations are only single-cycle, independent ALU operations that do not preserve a long-term state. As a result of this, we devised an enhanced version of traditional port contention channels to observe these additional operations.
Speculative frontend attacks. Many recent attacks exploit the structures in the frontend that speculatively induce a transient window that can force decoding or executing unintended instructions (i.e., not observable architecturally) [7, 12, 28, 29, 31, 35, 37, 40, 41, 44, 51, 53–56, 60, 61]. The stack engine is not itself predictive and, as such, it does not induce transient windows.

11 Conclusion

In this work, we present the first detailed reverse engineering of the stack engine. Our analysis of recent AMD and Intel CPUs reveals that all of them implement a stack engine and that their microarchitecture-specific implementations can leak sensitive information. For instance, the stack engine dispatches an additional uop when synchronizing the tracked `RSP` offset (i.e., Δ`RSP`) in a microarchitecture-dependent manner. Building on our reverse engineering results, we develop a set of primitives that leak information about an application's (potentially secret) control flow across security boundaries. We use these primitives to construct a number of covert and side channel attacks targeting a widely used JSON parsing library, enabling the leakage of sensitive information such as distinguishing patients based on their medical records. Finally, we uncover undocumented chicken bits in recent AMD Zen 4 and Zen 5 CPUs that allow disabling the stack engine. Evaluating SPEC CPU2017 applications with the stack engine disabled incurs a performance overhead of 3.98% on AMD Zen 4 and 3.94% on AMD Zen 5, prompting designing secure stack engine architectures in the future which we also discuss.

Acknowledgments

The authors would like to thank the anonymous reviewers and Jagadish Kotra for their feedback. This work was supported by a generous gift from AMD and the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

References

[1] Advanced Micro Devices, Inc. 2012. *Revision Guide for AMD Family 10h Processors (Including Errata #721).* Revision Guide (order #41322) Rev. 3.92. Advanced Micro Devices. https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/revision-guides/41322_10h_Rev_Gd.pdf accessed 2025-08-01.

[2] Advanced Micro Devices, Inc. 2025. *AMD64 Architecture Programmer's Manual: Volume 5 - 64-Bit Media and x87 Floating-Point Instructions.* https://www.amd.com/en/search/documentation/hub.html Document Number: 58455.

[3] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *SP*. https://doi.org/10.1109/SP.2019.00066

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security*.

[5] AMD. 2025. AMD-SB-7045: Microarchitectural Attacks on the Stack Engine. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7045.html accessed 2025-08-08.

[6] Muhammad Ayaz, Muhammad F Pasha, Mohammed Y Alzahrani, Rahmat Budiarto, and Deris Stiawan. 2021. The Fast Health Interoperability Resources (FHIR) standard: systematic literature review of implementations, applications, challenges and opportunities. In *JMIR medical informatics*.

[7] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *USENIX Security*.

[8] Michael Bekerman, Adi Yoaz, Freddy Gabbay, Stephan Jourdan, Maxim Kalaev, and Ronny Ronen. 2000. Early load address resolution via register tracking. In *ISCA*. https://doi.org/10.1145/339647.339705

[9] Dave Benson and protobuf-c contributors. 2025. protobuf-c: Protocol Buffers implementation in C. https://github.com/protobuf-c/protobuf-c Version 1.0+ accessed 2025-08-01.

[10] Chandler Carruth. 2018. *Speculative load hardening.* https://llvm.org/docs/SpeculativeLoadHardening.html accessed 2025-08-06.

[11] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. 2018. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races. In *SP*. https://doi.org/10.1109/SP.2018.00024

[12] Yun Chen, Ali Hajiabadi, and Trevor E. Carlson. 2024. GadgetSpinner: A New Transient Execution Primitive using the Loop Stream Detector. In *HPCA*. https://doi.org/10.1109/HPCA57654.2024.00013

[13] Yun Chen, Ali Hajiabadi, Lingfeng Pei, and Trevor E. Carlson. 2024. PrefetchX: Cross-Core Cache-Agnostic Prefetcher-Based Side-Channel Attacks. In *HPCA*. https://doi.org/10.1109/HPCA57654.2024.00037

[14] Shuwen Deng, Bowen Huang, and Jakub Szefer. 2022. Leaky Frontends: Security Vulnerabilities in Processor Frontends . In *HPCA*. https://doi.org/10.1109/HPCA53966.2022.00013

[15] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*. https://doi.org/10.1109/MICRO.2016.7783743

[16] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*. https://doi.org/10.1145/3173162.3173204

[17] Johann Faouzi. 2024. *Time Series Classification: A Review of Algorithms and Implementations.* https://doi.org/10.5772/intechopen.1004810

[18] Agner Fog. 2024. *The microarchitecture of Intel, AMD and VIA CPUs.* https://www.agner.org/optimize/microarchitecture.pdf accessed 2025-06-10.

[19] Dave Gamble and contributors. 2025. cJSON - Ultralightweight JSON parser in C. https://github.com/DaveGamble/cJSON. accessed 2025-04-10.

[20] GNU Project. 2022. glibc 2.35 Source Code: string/strtok_r.c. https://elixir.bootlin.com/glibc/glibc-2.35/source/string/strtok_r.c#L28 accessed 2025-04-08.

[21] Simcha Gochman, Ronny Ronen, Ittai Anati, Ariel Berkovits, Tsvika Kurts, Alon Naveh, Ali Saeed, Zeev Sperber, and Robert C Valentine. 2003. The Intel® Pentium® M Processor: Microarchitecture and Performance. *Intel technology journal* (2003).

[22] Google. 2025. Protocol Buffers. https://github.com/protocolbuffers/protobuf Version 3.0+ accessed 2025-08-01.

[23] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. 2020. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *NDSS*.

[24] Ali Hajiabadi and Trevor E. Carlson. 2024. Conjuring: Leaking Control Flow via Speculative Fetch Attacks. In *DAC*. https://doi.org/10.1145/3649329.3655895

[25] Tristan Hornetz and Michael Schwarz. 2025. PortPrint: Identifying Inaccessible Code with Port Contention. In *uASC*.

[26] Intel. 2022. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html accessed 2025-09-03.

[27] Intel Corporation. 2025. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* https://cdrdv2-public.intel.com/814198/248966-046A-software-optimization-manual.pdf Document Number: 248966-046A.

[28] Jason Kim, Jalen Chuang, Daniel Genkin, and Yuval Yarom. 2025. FLOP: Breaking the Apple M3 CPU via False Load Output Predictions. In *USENIX Security*.

[29] Jason Kim, Daniel Genkin, and Yuval Yarom. 2025. SLAP: Data Speculation Attacks via Load Address Prediction on Apple Silicon. In *SP*. https://doi.org/10.1109/SP61157.2025.00098

[30] Joonsung Kim, Hamin Jang, Hunjun Lee, Seungho Lee, and Jangwoo Kim. 2021. UC-Check: Characterizing Micro-operation Caches in x86 Processors and Implications in Security and Performance. In *MICRO*. https://doi.org/10.1145/3466752.3480079

[31] 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 *SP*. https://doi.org/10.1109/SP.2019.00002

[32] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs. In *EuroS&P*. https://doi.org/10.1109/EuroSP48549.2020.00027

[33] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *USENIX ATC*.

[34] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*.

[35] Luyi Li, Hosein Yavarzadeh, and Dean Tullsen. 2024. Indirector: High-Precision Branch Target Injection Attacks Exploiting the Indirect Branch Predictor. In *USENIX Security*.

[36] Yiming Li, Laiping Zhao, Yanan Yang, and Wenyu Qu. 2023. Rethinking Deployment for Serverless Functions: A Performance-First Perspective. In *SC*. https://doi.org/10.1145/3581784.3613211

[37] 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 *USENIX Security*.

[38] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS*. https://doi.org/10.1145/2694344.2694385

[39] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell. 2025. *System V Application Binary Interface AMD64 Architecture Processor Supplement.*

[40] Andrea Mambretti, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, and Anil Kurmus. 2019. Two methods for exploiting speculative control flow hijacks. In *WOOT*.

[41] Alyssa Milburn, Ke Sun, and Henrique Kawakami. 2023. You Cannot Always Win the Race: Analyzing mitigations for branch target prediction attacks. In *EuroS&P*. https://doi.org/10.1109/EuroSP57164.2023.00046

[42] Jan Olšan. 2024. *Zen 5: AMD's most innovative core since the original Zen – analysis.* https://www.hwcooling.net/en/zen-5-amds-most-innovative-core-since-the-original-zen-analysis/ Accessed: 2025-04-12.

[43] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Čapkun. 2021. Frontal attack: Leaking Control-Flow in SGX via the CPU frontend. In *USENIX Security*.

[44] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. 2021. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *USENIX Security*.

[45] Red Hat, Inc. 2024. Potential Dragonfly Issue with some older AMD processors. https://access.redhat.com/solutions/188273 Solution Verified; last updated August 6, 2024 accessed 2025-08-01.

[46] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. 2021. I See Dead µops: Leaking Secrets via Intel/AMD Micro-Op Caches. In *ISCA*. https://doi.org/10.1109/ISCA52012.2021.00036

[47] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural implications of function-as-a-service computing. In *MICRO*. https://doi.org/10.1145/3352460.3358296

[48] SPEC2017 2017. SPEC CPU2017. https://www.spec.org/cpu2017/.

[49] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. 2022. SecSMT: Securing SMT processors against Contention-Based covert channels. In *USENIX Security*.

[50] Daniel Townley and Dmitry Ponomarev. 2019. Smt-cop: Defeating side-channel attacks on execution units in smt processors. In *PACT*.

[51] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. 2023. Inception: Exposing New Attack Surfaces with Training in Transient Execution. In *USENIX Security*.

[52] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security*.

[53] Sander Wiebing, Alvise de Faveri Tron, Herbert Bos, and Cristiano Giuffrida. 2024. InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2. In *USENIX Security*.

[54] Johannes Wikner and Kaveh Razavi. 2022. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In *USENIX Security*.
[55] Johannes Wikner and Kaveh Razavi. 2025. Breaking the Barrier: Post-Barrier Spectre Attacks . In *SP*. https://doi.org/10.1109/SP61157.2025.00089
[56] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. 2023. Phantom: Exploiting Decoder-detectable Mispredictions. In *MICRO*. https://doi.org/10.1145/3613424.3614275
[57] Hans Winderix, Marton Bognar, Lesly-Ann Daniel, and Frank Piessens. 2024. Libra: Architectural Support For Principled, Secure And Efficient Balanced Execution On High-End Processors. In *CCS*. https://doi.org/10.1145/3658644.3690319
[58] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: revisiting RISC in an age of risk. In *ISCA*. https://doi.org/10.1109/ISCA.2014.6853201
[59] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. 2022. Towards Developing High Performance RISC-V Processors Using Agile Methodology. In *MICRO*. https://doi.org/10.1109/MICRO56248.2022.00080
[60] Hosein Yavarzadeh, Archit Agarwal, Max Christman, Christina Garman, Daniel Genkin, Andrew Kwong, Daniel Moghimi, Deian Stefan, Kazem Taram, and Dean Tullsen. 2024. Pathfinder: High-Resolution Control-Flow Attacks Exploiting the Conditional Branch Predictor. In *ASPLOS*. https://doi.org/10.1145/3620666.3651382
[61] Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean Tullsen. 2023. Half&Half: Demystifying Intel's Directional Branch Predictors for Fast, Secure Partitioned Execution. In *SP*. https://doi.org/10.1109/SP46215.2023.10179415
[62] Jianing You, Kang Chen, Laiping Zhao, Yiming Li, Yichi Chen, Yuxuan Du, Yanjie Wang, Luhang Wen, Keyang Hu, and Keqiu Li. 2025. AlloyStack: A Library Operating System for Serverless Workflow Applications. In *Eurosys*. https://doi.org/10.1145/3689031.3717490
[63] Jiyong Yu, Trent Jaeger, and Christopher Wardlaw Fletcher. 2023. All Your PC Are Belong to Us: Exploiting Non-control-Transfer Instruction BTB Updates for Dynamic PC Extraction. In *ISCA*. https://doi.org/10.1145/3579371.3589100
[64] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. Ultimate SLH: Taking Speculative Load Hardening to the Next Level. In *USENIX Security*.
[65] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*.

A Artifact Appendix

A.1 Abstract

This artifact provides a container/fakeroot-based image to run the experiments shown in Sections 7 to 9. It additionally contains the code used to reverse-engineer the stack engine and raw data of our experiments for inspection (Section 5). Running them on a computer with a Zen 5 CPU enables reproduction of our leakage experiments (Figure 10) and the performance overhead of disabling the stack engine as a mitigation (Figure 12).

A.2 Artifact check-list (meta-information)

- **Compilation:** gcc, clang
- **Run-time environment:** Debian-based Linux Distribution, kernel 6.8 or greater
- **Hardware:** AMD Zen 5 CPU, optionally Zen1-4; Intel 3rd, 7th, 12/13th, 14th gen CPUs; 8GiB RAM or more
- **Metrics:** Execution time, reached stack depth
- **Output:** Raw data, plots
- **Experiments:** Provided scripts
- **How much disk space required (approximately)?:** 16 GB
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes
- **How much time is needed to complete experiments (approximately)?:** 16h for performance evaluation (SPEC2017), 6h for the covert channel, 2h for the attack and everything else
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** GPLv3
- **Data licenses (if publicly available)?:** APACHE 2.0 (FHIR dataset)
- **Workflow automation framework used?:** None, bash script
- **Archived (provide DOI)?:** 10.5281/zenodo.16733968

A.3 Description

A.3.1 How to access. We provide the tar archive via Zenodo at https://doi.org/10.5281/zenodo.16733968.

A.3.2 Hardware dependencies. We believe that any Zen 5 CPU can be used to execute the experiments. While we observe that more recent AGESA/microcode revisions disable the stack engine `add`/`sub` support by default, it can be re-enabled by writing the corresponding MSR. In case artifact evaluators do not have access to a Zen 5 node, we can provide remote access on request.

Exact match (works out-of-the-box):

- AMD Ryzen 9 9900X 12-Core Processor, microcode 0xb40401c
- Mainboard: AGESA ComboAm5PI 1.2.0.1a, ROG STRIX B650E-F GAMING WIFI, BIOS 3035

Reproduced by us (stack engine `add`/`sub` support disabled by default, but can be re-enabled for testing):

- AMD Ryzen 5 9600X 6-Core Processor, microcode 0xb404023
- Mainboard: AGESA ComboAm5PI 1.2.0.2b, PRIME B650M-R, UEFI 3067

Should work based on our understanding:

- Any Zen 5 Desktop, Server, or Mobile CPU

A.3.3 Software dependencies.

- A recent Debian-based Linux distribution (scripts use `apt` to automatically install further dependencies)
- systemd v255 or greater
- Linux kernel 6.8 or greater with the capability of loading unsigned kernel modules (workarounds with `wrmsr` are possible)
- SPEC2017 cpu2017.iso (https://www.spec.org/cpu2017/)

We recommend running as a user with passwordless sudo enabled for automatic execution without interruptions.

A.4 Installation

To install, extract the provided tar archive, switch to the newly created directory. Then, copy your cpu2017.iso from the SPEC2017 benchmark to this directory and run `bash run.sh`. All other required dependencies will be pulled and installed automatically. While most dependencies are containerized, some modifications are made to the host, in particular: A known-good version of `libpfm4-dev` is installed, and a kernel module is compiled and loaded.

We recommend enabling passwordless sudo or allow-listing `systemd-nspawn` in the sudo rules before running to avoid password prompts during the run. In standard Linux environments, you may use the following script.

The installation will take around 15 minutes and will then prompt to enter `ready`. For your convenience, the following commands

```
echo "$(whoami) ALL=(ALL) NOPASSWD:ALL" | \
sudo EDITOR='tee' visudo -f /etc/sudoers.d/aemicro58
# Once experiments are run, cleanup with
sudo rm /etc/sudoers.d/aemicro58
```

perform the steps automatically. Pass `--ready` to `run.sh` to skip the prompt if desired.

```
export URL=https://zenodo.org/records/17007332/files
wget $URL/stackengine-files.tar.gz?download=1 -O- | \
  tar -xzf -
cd systemd-container
# Required only if SPEC2017 should run
cp <your/cpu2017.iso> ./
# Install and skip long-running experiments (~2h)
bash run.sh --skip-spec --skip-covert-channel
```

A.5 Experiment workflow

We automate most steps of the workflow. First, the covert channel is run. Then, the attack and smaller experiments are run. Lastly, the system automatically collects the performance numbers for SPEC2017 under different conditions. The resulting files are collected in a `results.tar.gz` in the main directory. Note that writing to the MSR may lead to hard system lockups in 1 out of 10 writes which resolves only by long-pressing the power button and partial reruns might be necessary.

We provide scripts to plot the results together with our results. These existing values are used by default for experiments which cannot be performed on the current host. This applies in particular to the reverse engineering which needs to be performed on multiple different machines. The bundled results can be replaced individually by overwriting the files in `/app/bundled-results` in the provided container directory.

A.6 Evaluation and expected results

Without arguments, `run.sh` will run all experiments back-to-back, taking roughly 24h in total. This results from the attack and small experiments (2h), the covert-channel plot (Figure 7, 6h) and the SPEC2017 benchmark (16h). To reduce the runtime while still showing the main claims of the paper, you may selectively disable specific experiments from running. Please note that plots will be generated from bundled data in this case.

```
# Run only the attack (2h)
bash run.sh --skip-spec --skip-covert-channel
# Run the attack and SPEC2017 (18h)
bash run.sh --skip-covert-channel
# Run everything (24h)
bash run.sh
```

At the end of the execution, a `result.tar.gz` will appear in the same directory as `run.sh` and a shell within the container will open. If you do not want to further inspect the container environment, you may press CTRL-D or type `logout` to exit the container. Browse `results.tar.gz` for the resulting raw data and PDF files.

The outputs are described in the following table. Please use `run.sh initcheck` within the container to receive a listing of which plots were created from bundled data and which data was run during evaluation. You may also retrieve the raw data obtained at `/raw-results` within the container. Note that some PDFs will be located in subdirectories.

Figure	Filename output
Figure 2	stack_engine_push.pdf
Figure 3	stack_engine_reset_op.pdf
Figure 4 (top)	stack_pointer_depth_amd.pdf
Figure 4 (bottom)	stack_pointer_depth_intel.pdf
Figure 7	covert-channel.pdf
Figure 8 (a)	portcon_calibration_untuned.pdf
Figure 8 (b)	portcon_calibration_tuned.pdf
Figure 10	pku_matching_json.pdf
Figure 12	spec2017_relchange_cycles.pdf

For the attack, you should see a plot similar to the one presented in Figure 10. Note that the colors of the lines and their position (ID) may differ, but you should be able to clearly identify 5 peaks. Timing-based analysis may result in lower amplitudes and higher noise, but the line colors and peaks should align with the results for the test using performance counter values.

For the covert channel (Figure 7), some deviation may occur. Please note that the script does not reproduce all time points in order to save time. However, the starting accuracies and performance should be roughly visible.

For the SPEC2017 performance evaluation, results may depend on the system configuration, including but not limited to: RAM access times, disk speed, CPU and cooling behavior. You should see an overhead of around 5% without the stack engine for the number of dispatched operations, and an overhead of around 1% when disabling `add/sub`. While timing results might suffer from heavy noise depending on the environment and frequency changes, the number of dispatched ops should remain relatively close to 1% and 5% when disabling `add/sub` and the full stack engine, respectively.

A.7 Experiment customization

By default, we run a single iteration of the SPEC2017 benchmark. For more stable results with X repetitions, use the following options. Please note that every SPEC repetition will take around 16 hours.

```
bash run.sh --spec-repetitions <X>
```

We provide the source code and build scripts for all experiments used. To run individual experiments, use the command below and enter the container environment interactively.

Within the container, you can navigate to the `/app` directory and run `bash run.sh --help` to list all available individual experiments. The experiment files and sources are also located at `/app`. Press CTRL-D or type `logout` to leave the container environment.

```
systemd-nspawn -D container-root \
--console=interactive \
--system-call-filter=@pkey \
--system-call-filter=perf_event_open \
--capability=CAP_PERFMON \
--bind /dev/stackmsr \
/bin/bash
```

Most of the attack code is located in `stackengine-attack`. In the script `run.sh`, you may want to adjust the `--sample` parameters to sample different JSON-files and see different peaks emerging as a result.

The `stackengine-msr` directory contains the `enable` binary in `src/obj/enable`. It offers the following options:

- `--disable-stackengine` Disables the stack engine completely
- `--disable-addsub` Disables `add`/`sub` support

No arguments enable all available stack engine features.

The `reverse-eng` directory contains the experiments used for reverse engineering. If you want to run the reverse engineering experiments, locate the `run_on_all.sh` files in the `reverse-eng` directory and supply the corresponding hosts. The entire `reverse-eng` directory should be copied to the home directories on the tested hosts and the placeholder hosts should be set or removed if unused. The resulting `.csv` files should then be copied over the provided `.csv` files in the bundled directories.

A.8 Notes

The scripts may leave files in the host system, but do not reconfigure the host to automatically apply configurations on the next boot. Some of the changed settings (in particular, exposing an MSR and unlocking performance counters) may impact the security of the system. We recommend rebooting the system after the experiments are done to reduce the attack surface.