BACKPAC: Breaking ARM's Pointer Authentication Sidechannel Mitigation in Apple M2

Yongjun Han
Seoul National University
Seoul, Republic of Korea
yongjuni2000@snu.ac.kr

Dae R. Jeong
Seoul National University
Seoul, Republic of Korea
dae.r.jeong@snu.ac.kr

Abstract

ARM Pointer Authentication Code (PAC) is an architectural security mechanism designed to enforce pointer integrity. PACMAN [28] showed that speculative execution can turn pointer-authentication outcomes into a microarchitectural oracle, allowing attackers to break the promise of PAC. In response, ARM proposed a combined hardware–software mitigation [6]. The hardware feature `FEAT_FPAC` raises a fault on failed authentication immediately, and a software sequence uses `XPAC` to clear PAC metadata, so that successful and failed authentications produce indistinguishable speculative memory accesses.

In this paper, we show that this mitigation can still be defeated, and that the responsibility lies with its microarchitectural implementation. On processors implementing `FEAT_FPAC`, a new authentication-dependent transient behavior arises that the `XPAC`-based software sequence fails to eliminate. As a result, an attacker can still distinguish successful from failed authentications through a previously unexplored side-channel signal, despite the combined mitigation being correctly applied. We characterize this leakage on Apple M2 processors that implement `FEAT_FPAC`, and discuss how it could be exploited to mount a new PAC bypass attack. Our results highlight that architectural security specifications are only as sound as their microarchitectural implementation.

CCS Concepts

• **Security and privacy → Side-channel analysis and countermeasures**; *Hardware security implementation.*

APSys '26, Bangkok, Thailand

ACM ISBN 979-8-4007-2912-6/2026/09
https://doi.org/10.1145/3838177.3841734

Keywords

Security, memory corruption attacks, micro-architectural side channels, pointer authentication, mitigation

ACM Reference Format:
Yongjun Han and Dae R. Jeong. 2026. BACKPAC: Breaking ARM's Pointer Authentication Sidechannel Mitigation in Apple M2 . In *17th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '26), September 17–18, 2026, Bangkok, Thailand.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3838177.3841734

1 Introduction

Memory safety vulnerabilities remain a persistent threat to modern computing systems. Exploits such as buffer overflows [23] and use-after-free [22] bugs allow attackers to corrupt memory and hijack the control flow of an victim program. Advanced code-reuse attacks, including Return-Oriented Programming (ROP) [30] and Jump-Oriented Programming (JOP) [8], have rendered traditional defenses like non-executable memory (NX) insufficient. To address these challenges, hardware vendors have introduced architectural extensions designed to enforce pointer and control-flow integrity at the hardware level [4, 5, 11, 27].

ARM Pointer Authentication (PAC) is a prominent hardware security feature introduced in the ARMv8.3-A architecture [27] and deployed in modern processors, including Apple's M-series chips [3]. PAC aims to protect pointer integrity by embedding a cryptographic signature into the unused upper bits of a 64-bit pointer. Pointers are signed with a `PAC` instruction (e.g., `PACIA`) and verified with a corresponding authentication instruction (e.g., `AUTIA`) before dereference. If authentication fails, the processor triggers a fault, preventing the use of tampered pointers and mitigating a broad class of control-flow hijacking attacks.

While PAC provides a robust defense against architectural attacks, the rise of microarchitectural side-channel attacks has introduced a new dimension of vulnerability. Attacks like Spectre [19] and Meltdown [20] demonstrated that speculative execution—a fundamental performance optimization in modern processors—can leak

sensitive information across security boundaries. Furthermore, the intersection of memory corruption vulnerabilities and microarchitectural side-channels was explored in the PACMAN attack [28], which demonstrated that speculative execution could be leveraged to bypass PAC authentication. By constructing a "PAC Oracle" via side-channels, an attacker can brute-force the cryptographic signature without triggering a crash, effectively breaking the security guarantees of PAC.

In response to PACMAN, both the academic community and industry have proposed various mitigations [2, 6, 9, 25, 26]. In particular, ARM introduced `FEAT_FPAC`[1], a hardware extension that alters the exception handling mechanism of authentication failures [5, 6]. ARM also suggests pairing these extensions with an `XPAC`-based software pattern, which strips the embedded signature from a pointer before subsequent use, ensuring that speculatively executed code operates on a canonical pointer even when authentication fails. The underlying assumption is that by applying the `XPAC`-based software pattern alongside modified fault behavior, the threat of speculative PAC bypasses can be contained.

This paper demonstrates that ARM's suggested mitigation, while effective against fault-based oracles, inadvertently introduces a new form of PAC leakage. We show that authentication failures still influence the processor's *speculative execution window* even after the fault-based oracle is removed, and present BACKPAC, a new primitive that exploits this effect to recover PAC signatures without triggering any faults.

In summary, the paper makes the following contributions:

- We analyze and characterize existing PACMAN mitigations proposed by various vendors.
- We show that on Apple M2, authentication failures shorten the speculative execution window, leaving an authentication-dependent microarchitectural signal even when ARM's recommended mitigation is applied.
- We construct BACKPAC, an oracle that distinguishes successful from failed PAC authentication without triggering any architectural fault.

Responsible Disclosure. We disclosed our findings to the Apple Security Team on December 4, 2025. Apple successfully reproduced our PoC but decided not to fix it, on the grounds that ARM's PACMAN mitigation guidance applies specifically to ARM-produced CPU cores.

[1] `FEAT_FPACC_SPEC` was introduced later as an extension that further hardens speculative authentication behavior. For hardware without `FEAT_FPACC_SPEC`, ARM officially recommends pairing `FEAT_FPAC` with the `XPAC`-based software pattern [6].

```
1 pacia lr, sp
2 sub   sp, sp, #0x40
3 str   lr, [sp, #0x30]
4 ...
```

(a) Function prologue.

```
1 ldr   lr, [sp, #0x30]
2 add   sp, sp, #0x40
3 autia lr, sp
4 ret
```

(b) Function epilogue.

Figure 1: PAC signing and authentication of the return address.

2 Background

This section provides the necessary background on ARM Pointer Authentication, speculative execution attacks, and the PACMAN attack.

2.1 ARM Pointer Authentication

ARM Pointer Authentication (PAC) [5, 27], introduced in ARMv8.3-A, mitigates code-reuse attacks such as ROP [30] and JOP [8] by embedding a cryptographic signature—derived from the pointer value, a 64-bit context, and a secret 128-bit key—in the unused upper bits of a 64-bit virtual address. `PAC*` instructions sign a pointer and `AUT*` instructions verify it.

If authentication fails, the hardware does not raise an exception immediately; instead, it mangles the pointer so that a subsequent dereference triggers a translation fault. This deferred error handling preserves speculative execution of code paths that may be discarded—but it is precisely the design choice that PACMAN later exploits.

2.2 Speculative Execution Attacks

Modern processors employ speculative execution to improve performance by executing instructions before their control dependencies are resolved. On a mis-prediction, the architectural state is rolled back, but microarchitectural state—such as caches, TLBs, and branch predictors—retains side effects of the transient execution. Speculative execution attacks [14–17, 19, 21, 28, 29] exploit this asymmetry: when transient execution touches secret-dependent data, an attacker observes the resulting state changes through microarchitectural side channels (e.g., cache timing via `Prime+Probe` [24] or `Flush+Reload` [31]) to infer the secret. Recent work has shown that faults raised during speculative execution can shorten the speculation window itself [17], creating a new class of side-channel distinct from cache-based oracles.

2.3 The PACMAN Attack

PACMAN [28] is a novel attack demonstrated by Ravichandran et al. that combines the aforementioned side channels with speculative execution to bypass ARM PAC. It utilizes speculative execution to create a side-channel

```
1 ldr X0, [X1]
2 aut X0
3 xpac X0
4 ldr X2, [X0]
5 ...
```

(a) ver1

```
1 ldr X0, [X1]
2 mov X3, X0
3 xpac X0
4 aut X3
5 ldr X2, [X0]
```

(b) ver2

Figure 2: Mitigation flow using `XPAC` on systems implementing `FEAT_FPAC`. By clearing the PAC field after authentication, both successful and failed authentication paths transiently execute loads with the same effective address, preventing the original PACMAN oracle [6].

oracle, allowing an attacker to distinguish between a correct and an incorrect PAC without triggering a system crash.

The core of the PACMAN attack is a "PACMAN gadget," a code sequence that speculatively authenticates a pointer and then uses it in a way that leaves a microarchitectural trace (e.g., a cache modification) dependent on the validity of the pointer. An attacker can brute-force the PAC by guessing a value, injecting it, and triggering the gadget. Since the authentication failure mangles the pointer rather than faulting immediately, the processor may speculatively execute subsequent instructions using the mangled pointer. This speculative execution creates a measurable difference in the cache state compared to the execution with a correct pointer, which the attacker can detect. By repeating this process, an attacker can recover the correct PAC for a given pointer and context.

3 Mitigations against PACMAN

Following the disclosure of PACMAN, several mitigations were discussed. However, deploying effective defenses has proven challenging due to performance and compatibility constraints.

Software Mitigations. The primary software mitigation involves manually inserting speculation barriers (e.g., `SB` or `DSB` instructions) in critical code paths. These barriers prevent the processor from speculating past the authentication instruction, thereby eliminating the sidechannel leakage. However, due to the high performance overhead of these instructions, they are applied very selectively (e.g., in the Linux kernel context switching code) and do not provide comprehensive protection against all potential PACMAN gadgets in user-space applications.

Combined Hardware and Software Mitigations. ARM introduced `FEAT_FPAC` in the Armv8.3-A architecture extension [7], a hardware extension that modifies the architectural handling of pointer authentication failures: when authentication fails, the `AUT*` instruction directly raises a PAC Fail exception, rather than producing a non-canonical VA that faults only on a subsequent dereference.

On systems implementing `FEAT_FPAC`, ARM recommends mitigating PACMAN by explicitly clearing the PAC field with `XPAC` after authentication, and provides two software sequences for doing so [6], illustrated in Figure 2. The first sequence (ver1) applies `XPAC` directly to the authenticated register so that subsequent loads use the canonicalized pointer regardless of the authentication outcome. This sequence has two limitations: it relies on an implementation-specific guarantee that the faulting `AUT` does not stall younger instructions (which holds in ARM-designed cores), and it is incompatible with combined authenticate-and-use instructions such as `RETAA` or `LDRAA`. The second sequence (ver2) avoids this limitation by decoupling authentication from the load: the pointer is copied to a scratch register on which `AUT` is performed solely to trigger a fault on failure, while `XPAC` is applied to the original register, which is then used for subsequent loads.

Both sequences share the same underlying idea: ensure that the authentication metadata is removed before any speculative dereference, so that successful and failed authentication paths execute structurally similar memory operations. As a result, both paths transiently produce similar microarchitectural state, and the original PACMAN oracle is eliminated.

Hardware-only Mitigation. ARM's `FEAT_FPACC_SPEC` and recent industry proposals aim to make speculative microarchitectural state independent of the authentication outcome. Apple [2] makes speculation proceed independently of the authentication result, so that the speculative path is the same regardless of the outcome, while Qualcomm [26] modifies the signature of a failing pointer so that it appears valid to subsequent speculative uses. On Apple M2, however, the deployed mitigation is the combined `FEAT_FPAC`+`XPAC` approach described above. As we show in §4.1, this combined mitigation is still vulnerable to a new class of PAC leakage arising from speculation shrinkage.

4 Speculation Shrinkage under `FEAT_FPAC`

In this section, we examine how pointer-authentication outcomes affect transient execution under PAC mitigations. While prior work has reported speculation shrinkage caused by MTE faults on ARM CPUs [17], we focus on authentication-induced changes to the speculation window on Apple Silicon M2.

```
1 if (cond) {
2     autda   x0, x3              // authenticate pointer
3     xpacd   x0                  // strip PAC bits
4     ldr     x2, [x0]
5     orr     x2, x2, x2          // RAW dependency chain:
6     orr     x2, x2, x2          //   controllable delay
7     ...
8     ldr     x2, [x1, x2, lsl #6]  // encode into cache
9 }
```

Figure 3: Speculation-measurement gadget. A read-after-write `orr` chain after PAC stripping extends the speculation window before the final encoding load.

4.1 Experimental Setup

We conduct our experiments on both the performance (P) and efficiency (E) cores of Apple M2. Since Apple M3 shares the same relevant architectural features as these cores, we expect similar behavior, indicating potential susceptibility.

Our experiments are performed on macOS 26.2 running on Apple M2. For precise reverse engineering, we require high-resolution timing sources and reliable cache flush instructions. On Apple M2, by leveraging *PacmanPatcher [13]*, we were able to access undocumented, platform-specific performance counter registers as well as the `dc civac` instruction from user space, enabling higher-resolution measurements. Specifically, we use the performance counter exposed through the system register `S3_2_c15_c0_0` as our timing source. We adopt Google Project Zero's MTETest [10] for our measurements, modifying it for PAC.

4.2 Speculation Shrinkage Measurement

To observe speculation shrinkage, we use the measurement gadget shown in Listing 3. The gadget authenticates a pointer with `autda` (line 2), models ARM's recommended mitigation by stripping its PAC bits with `xpacd` (line 3), and dereferences the resulting pointer (line 4). The loaded value is then passed through a chain of dependent `orr` instructions before being encoded into the cache via the final indexed load (line 8). The `orr` chain forms a read-after-write dependency, preventing out-of-order collapse and acting as a controllable delay between the authenticated load and the final encoding load.

We trigger transient execution by mispredicting the branch at line 1. If transient execution reaches the final load, the corresponding cache line is brought into the

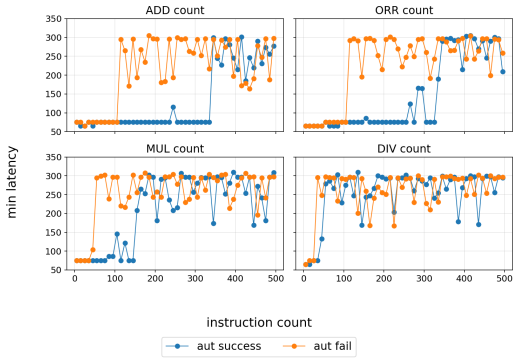

Figure 4: Speculation window (in executed instructions) by instruction latency. Orange: failed authentications; blue: successful. Lower-latency ops (`add`, `orr`) fit more instructions before shrinkage than higher-latency ops (`mul`, `udiv`).

Table 1: Instruction latencies

Execution Class	Instruction Type	Latency (cycles)
ALU	`orr`	1–2
ALU	`and`	1–2
MUL	`mul`	3
DIV	`udiv` (64-bit)	7–9

cache, which the attacker detects by probing and measuring access latency. By varying the length of the `orr` chain and checking whether the cache footprint remains observable, we estimate how far transient execution progresses after authentication.

Despite the `XPAC`-based mitigation, we observe that authentication failures shorten the speculation window on Apple M2.

These results show that although the `XPAC`-based mitigation removes the original PACMAN-style load oracle, it does not eliminate all authentication-dependent transient behavior. Authentication failures under `FEAT_FPAC` induce speculation shrinkage, allowing the attacker to distinguish authentication success from failure through the presence or absence of the final microarchitectural footprint.

To verify that this effect is driven by elapsed cycles rather than by the specific choice of delay instruction, we repeat the measurement with the `orr` chain replaced by other instruction types. Figure 4 shows that the speculation window length, expressed in number of executed instructions, scales inversely with per-instruction latency: low-latency operations (`add`, `orr`) fit more instructions into the window, with shrinkage on failed authentication

beginning at about 115 instructions for both, whereas higher-latency operations cause it to appear earlier, at about 55 instructions for `mul` and about 35 for 64-bit `udiv`. Per-instruction latencies, taken from Apple's CPU optimization guide [1], are summarized in Table 1.

All measurements reported in this paper are taken on the performance (P) core, while we confirmed that the same speculation shrinkage also occurs on the efficiency (E) core.

AI-use disclosure. We used OpenAI Codex solely to assist in writing the Python plotting script used to generate Figure 4. We independently collected the experimental data, reviewed and validated the script and its output, and performed the analysis and interpretation. The AI tool did not generate, alter, or interpret the experimental data or results.

5 Security Implications

In this section, we analyze the security implications of speculation shrinkage and the conditions under which the shrinkage becomes an exploitable PAC leakage primitive which we call BACKPAC.

In the remainder of this section, we first describe our threat model (§5.1), then explain how the leakage manifests as a multi-step attack (§5.2), and finally identify code patterns that are vulnerable to this leakage (§5.3).

5.1 Threat Model

We adopt the PACMAN threat model [28], where the attacker's goal is to recover a valid PAC for an attacker-chosen pointer on a PAC-protected system, ultimately enabling control-flow hijacking via ROP/JOP-style attacks. The victim implements `FEAT_FPAC` but not `FEAT_FPACC_SPEC`, which is the case on Apple M2 and M3, employs `XPAC`-based mitigations against PACMAN-style attacks, and contains a known memory corruption vulnerability as well as a BACKPAC gadget (§5.3). The attacker runs unprivileged code on the same core as the victim, shares no memory with it, but can observe microarchitectural side channels such as the cache and TLB. Prior work has already reverse-engineered Apple M2's cache and TLB structure in sufficient detail to make these side channels practically exploitable [12, 18].

5.2 BACKPAC Overview

BACKPAC exploits the fact that authentication failures shorten the speculation window: a successful authentication lets transient execution reach subsequent memory or branch operations and touch additional microarchitectural state, while a failed one truncates speculation

```
1 if (cond) {
2   autda  ptr
3   xpac ptr
4   ... (enough instructions)
5   load(ptr2)
6 }
```

(a) Data gadget.

```
1 if (cond) {
2   autia  ptr
3   xpac ptr
4   ... (enough instructions)
5   br(ptr2)
6 }
```

(b) Instruction gadget.

Figure 5: Pseudo-code gadgets used in our attack. The data gadget dereferences a PAC-stripped pointer, while the instruction gadget uses it as an indirect branch target.

before those effects occur. Although both paths are architecturally squashed, their microarchitectural footprints remain distinguishable.

At a high level, the attack proceeds in the following steps:

(1) **Mistraining the branch predictor.** The attacker mistrains the branch predictor to induce speculative execution along a chosen control-flow path containing a BACKPAC gadget.
(2) **Triggering speculative execution.** The attacker provides inputs that cause the processor to speculatively execute the gadget, which performs pointer authentication on an attacker-influenced pointer.
(3) **Send phase.** The authentication outcome is implicitly encoded into microarchitectural state: a successful authentication allows transient execution to reach a subsequent memory or branch operation, while a failed authentication truncates transient execution before that operation is issued.
(4) **Receive phase.** The attacker recovers the encoded information via side-channel measurements (e.g., cache or TLB timing). The presence or absence of a measurable signal reveals whether the pointer authentication succeeded or failed.
(5) **Brute-force phase.** The attacker iteratively tests candidate PAC values by repeating steps (1)–(4) until the correct PAC is identified.

5.3 Vulnerable Code Patterns

Since pointer authentication can be applied to both data flow and control flow, BACKPAC gadgets take two corresponding forms, illustrated in Figure 5. A *data gadget* (Figure 5(a)) authenticates a pointer with `autda`, strips its PAC bits with `xpac`, and is followed by a transient instruction window ending in a memory access through an independent pointer. An *instruction gadget* (Figure 5(b)) uses the same structure but ends in an indirect branch through an independent register, with `autia` as the authenticating instruction. In both cases,

successful authentication allows transient execution to reach the final operation and leave a microarchitectural footprint (in the cache, TLB, or branch predictor), while a failed authentication truncates speculation before that operation is issued.

6 Countermeasures

The most principled mitigation lies in hardware. ARM's `FEAT_FPACC_SPEC` extension is specifically designed to ensure that authentication outcomes do not affect speculative microarchitectural state [6], directly eliminating the primitive that BACKPAC exploits. Apple has adopted this mitigation starting from the M4 generation, where we no longer observe the speculation shrinkage on which BACKPAC relies. However, M2 and M3, which implement `FEAT_FPAC` without `FEAT_FPACC_SPEC`, remain susceptible.

On affected processors, software can insert speculation barriers (`SB`, `DSB`) between authentication and subsequent memory or control-flow operations, but their performance overhead limits practical use. Compilers and security-critical libraries should therefore avoid emitting BACKPAC gadgets (§5.3).

7 Conclusion

We presented BACKPAC, a microarchitectural leakage primitive arising from the interaction between ARM's `FEAT_FPAC` hardware mitigation and its recommended `XPAC`-based software sequences. While this combination eliminates the original PACMAN oracle, it introduces a new authentication-dependent transient behavior in which failed pointer authentication shortens the speculation window. Our findings show that pairing fault-based hardware mitigation with software pointer stripping is insufficient to eliminate authentication-dependent transient behavior.

References

[1] Apple Inc. 2025. Apple Silicon CPU Optimization Guide. https://developer.apple.com/documentation/apple-silicon/cpu-optimization-guide. Accessed: 2026-05-14.

[2] Apple Inc. 2025. Consistent Speculation of Pointer Authentication. U.S. Patent Application 20250094567, published March 20, 2025. https://patents.justia.com/patent/20250094567.

[3] Apple Inc. 2026. Apple Platform Security: Operating System Integrity. https://support.apple.com/guide/security/operating-system-integrity-sec8b776536b/web. Accessed: 2026-05-21.

[4] Arm. 2018. Arm A-Profile Architecture Developments 2018: Armv8.5-A. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-a-profile-architecture-2018-developments-armv85a. Accessed: 2026-05-17.

[5] Arm Limited. 2023. *Arm Architecture Reference Manual for A-profile architecture*. DDI 0487J.a.

[6] Arm Ltd. 2025. Arm CPU Security Update: Pointer Authentication. https://developer.arm.com/documentation/110389/latest/.

[7] Arm Ltd. 2025. Feature Descriptions: The Armv8.3 Architecture Extension. https://developer.arm.com/documentation/109697/2025_12/Feature-descriptions/The-Armv8-3-architecture-extension. Accessed: 2026-05-14.

[8] Tyler Bletsch, Xuxian Jiang, Vincent W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. 30–40.

[9] Neophytos Christou, Alexander J. Gaidis, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2024. Eclipse: Preventing Speculative Memory-error Abuse with Artificial Data Dependencies. *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security* (2024). doi:10.1145/3658644.3690201

[10] Google Project Zero. [n. d.]. p0tools: MTETest. https://github.com/googleprojectzero/p0tools/tree/master/MTETest. Accessed: 2026-08-19.

[11] Intel. 2020. A Technical Look at Intel Control-Flow Enforcement Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html. Accessed: 2026-05-17.

[12] Hyerean Jang, Taehun Kim, and Youngjoo Shin. 2024. SysBumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 64–78. doi:10.1145/3658644.3690189

[13] jprx. 2025. PacmanPatcher: Patch XNU to enable userspace access to Apple performance counters and cache maintenance instructions. https://github.com/jprx/PacmanPatcher. Accessed: 2026-05-09.

[14] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. 2023. The Gates of Time: Improving Cache Attacks with Transient Execution. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1955–1972. https://www.usenix.org/conference/usenixsecurity23/presentation/katzman

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

[16] Jason Kim, Daniel Genkin, and Yuval Yarom. 2025. SLAP: Data Speculation Attacks via Load Address Prediction on Apple Silicon. In *2025 IEEE Symposium on Security and Privacy (SP)*. 3549–3566. doi:10.1109/SP61157.2025.00098

[17] Juhee Kim, Jinbum Park, Sihyeon Roh, Jaeyoung Chung, Youngjoo Lee, Taesoo Kim, and Byoungyoung Lee. 2025. Tiktag: Breaking ARM's Memory Tagging Extension with Speculative Execution . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 4063–4081. doi:10.1109/SP61157.2025.00039

[18] Jason Kim, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. 2023. iLeakage: Browser-based Timerless Speculative Execution Attacks on Apple Devices. In *ACM CCS 2023*.

[19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[20] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. 973–990.

[21] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2109–2122. doi:10.1145/3243734.3243761

[22] MITRE. 2006. *CWE-416: Use After Free*. Common Weakness Enumeration. https://cwe.mitre.org/data/definitions/416.html Accessed: 2026-05-19.

[23] Aleph One. 1996. Smashing the stack for fun and profit. *Phrack magazine* 7, 49 (1996), 14–16.

[24] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*. Springer, 1–20.

[25] Ciyan Ouyang, Peinan Li, Yubiao Huang, Dan Meng, and Rui Hou. 2026. Janus: Compiler-Based Defense Against Transient Execution Attacks Using ARM Hardware Primitives. arXiv:2605.10049. https://arxiv.org/abs/2605.10049.

[26] Qualcomm Inc. 2024. Mitigating Pointer Authentication Code (PAC) Attacks in Processor-Based Devices. WIPO Patent Application WO2024059407A1, published March 21, 2024. https://patents.google.com/patent/WO2024059407A1/en.

[27] Qualcomm Technologies, Inc. 2017. *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions*. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[28] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. 2022. PACMAN: Attacking ARM Pointer Authentication with Speculative Execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*.

[29] Sandro Rüegge, Johannes Wikner, and Kaveh Razavi. 2025. Branch Privilege Injection: Compromising Spectre v2 Hardware Mitigations by Exploiting Branch Predictor Race Conditions. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association, Seattle, WA, 2615–2631. https://www.usenix.org/conference/usenixsecurity25/presentation/ruegge

[30] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*. 552–561.

[31] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. 719–732.