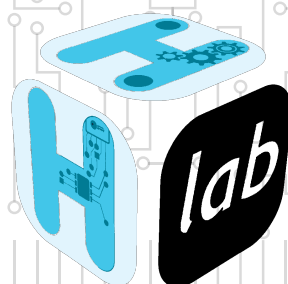

Microarchitectural attacks: concepts and analysis



H²Lab technical analysis





Information

Created in 2020, H2Lab is a non-profit organization dedicated to designing, developing, and sharing open-hardware and open-source solutions for experimentation around embedded systems. Its main areas of work are:

- Design of hardware platforms for hardware and software analysis
- Development of open alternatives to proprietary, often opaque tools
- Publication of technical guides and recommendations to promote best practices in security, privacy, and sustainability across the ecosystem of connected devices and embedded systems

Support for researchers, teachers, and students through the provision of tools, educational content, and training.

Academic partnerships to encourage sharing of resources and knowledge.

Revision history

Version	Date	Author	Changes
1.0	October 2026	H2Lab	Initial version of the guide

Contents

Information	i
1 Glossary and acronyms	1
1.1 Glossary	1
1.2 List of acronyms	1
2 Introduction and scope	4
2.1 Purpose of the guide	4
2.2 Intended audience	4
2.3 Scope and terminology	4
2.4 Executive summary	5
3 Microarchitectural fundamentals	6
3.1 Visible architecture and actual implementation	6
3.1.1 Internal structures relevant to the analysis	6
3.2 Out-of-order execution and speculation	7
3.2.1 Why in-order retirement is not enough	7
3.3 Common side channels	8
3.3.1 Two families of cache measurement	8
3.4 General chain of a transient attack	9
3.4.1 What distinguishes a leak from a full compromise	9
4 Overview of published attacks	11
4.1 Timeline	11
4.2 Spectre: controlled speculative execution	12
4.2.1 Spectre v1 – speculative bounds check bypass	12
4.2.2 Spectre v2 – branch target injection	13
4.2.3 Other Spectre variants	14
4.3 Meltdown and fault-related leaks	15
4.3.1 Meltdown (2018)	15
4.3.2 Foreshadow / L1TF and related flaws	16
4.4 Data sampling: MDS and derivatives	16
4.5 Value injection and gadget-driven speculative execution	17
4.6 Rowhammer and DRAM disturbances	18
4.6.1 Principle and first demonstrations	18
4.6.2 Bypassing or outgrowing the defences	19
4.6.3 Measures suited to embedded products	19
4.7 CLKSCREW: faults induced through power management	20
4.7.1 Why this attack is included in the guide	20



4.7.2	Background on DVFS and timing margins	20
4.7.3	How CLKSCREW works	20
4.7.4	Demonstrated impacts	21
4.7.5	Protection measures and limits	21
4.8	Timing channels outside transient execution	21
5	Impacts on an embedded system	23
5.1	Cross-reading of attack families	23
5.2	Security boundaries involved	24
5.3	Consequences and risk factors	24
5.4	Embedded-specific considerations	25
5.4.1	Asymmetric cores and heterogeneous SoCs	25
5.4.2	Real-time and resource-constrained systems	25
5.4.3	DMA, accelerators and peripherals	25
5.4.4	Boot, updates and lifetime	25
6	Countermeasures and analysis method	27
6.1	Defence in depth	27
6.1.1	Silicon and microcode	27
6.1.2	Kernel, hypervisor and firmware	28
6.1.3	Compiler and application	28
6.1.4	Memory and DRAM	29
6.2	Evolution and limits of mitigations	29
6.2.1	How to analyse an announced bypass	29
6.3	Qualification procedure for a product	30
6.4	Design review questions	30
7	Conclusion	31
	License	37



1

Glossary and acronyms

1.1 Glossary

Speculative execution Early execution of instructions before the processor has determined that they belong to the path actually taken. The architectural results of an abandoned path are discarded, but some microarchitectural effects may remain

Architectural state State visible according to the processor architecture: registers, memory, exceptions and control flow as defined by the ISA

Microarchitectural state Internal state that determines the cost of operations: caches, predictors, queues, TLB, load buffers and DRAM structures

Side channel Observable property, distinct from the intended functional output, whose variation makes it possible to infer information such as execution time, cache occupancy or power consumption

Transient execution Instructions executed before a dependency or a prediction is resolved, then removed from the architectural path when the prediction turns out to be wrong

Privileged resident Software running with elevated privileges such as the kernel, the hypervisor, the firmware or the secure monitor

Rowhammer Electrical disturbance of DRAM cells caused by repeated activation of neighbouring rows, which may flip bits in rows that were not targeted

Speculation barrier Instruction or sequence that prevents, depending on the architecture, some subsequent instructions from executing before an earlier event is resolved

1.2 List of acronyms

ABI Application Binary Interface, binary interface between compiled components

AES Advanced Encryption Standard, symmetric encryption algorithm

AMD Advanced Micro Devices, processor manufacturer



ARM Family of Arm processor architectures

BHI Branch History Injection, injection into the branch history

BIOS Basic Input/Output System, platform initialisation firmware

BTB Branch Target Buffer, branch target prediction structure

BTI Branch Target Injection, injection of branch targets

CFI Control-Flow Integrity

CLKSCREW Name of the attack published under the title *Exposing the Perils of Security-Oblivious Energy Management*

CVE Common Vulnerabilities and Exposures, public vulnerability identifier

DMA Direct Memory Access, memory access performed directly by a peripheral

DRAM Dynamic Random-Access Memory

DRAMMER Deterministic Rowhammer Attacks on Mobile Platforms, Rowhammer attack on mobile platforms

DVFS Dynamic Voltage and Frequency Scaling

ECC Error-Correcting Code

FPGA Field-Programmable Gate Array, programmable logic device

IBPB Indirect Branch Prediction Barrier

IBRS Indirect Branch Restricted Speculation

IOMMU Input-Output Memory Management Unit

ISA Instruction Set Architecture

KASLR Kernel Address Space Layout Randomization

KPTI Kernel Page-Table Isolation

L1 Level 1, cache level closest to the core

L1TF L1 Terminal Fault, family of transient faults related to the L1 cache

LTO Link-Time Optimization

LVI Load Value Injection

MDS Microarchitectural Data Sampling

MMU Memory Management Unit

OS Operating System

PCID Process-Context Identifier

RIDL Rogue In-Flight Data Load, attack sampling in-flight data



RISC Reduced Instruction Set Computer

RISC-V Open instruction set architecture of the RISC family

RSB Return Stack Buffer, return address prediction buffer

RTOS Real-Time Operating System

SGX Software Guard Extensions, Intel hardware enclave technology

SMT Simultaneous Multithreading, concurrent execution of several threads on one core

SoC System on Chip

SSBD Speculative Store Bypass Disable

STIBP Single Thread Indirect Branch Predictors, isolation of indirect predictors between threads

TEE Trusted Execution Environment

TLB Translation Lookaside Buffer, address translation cache

TRR Target Row Refresh, targeted refresh of DRAM rows

USENIX Association that organises computing conferences and publishes research papers

VM Virtual Machine

2

Introduction and scope

2.1 Purpose of the guide

This document explains microarchitectural attacks to engineers who design, integrate or secure embedded systems. It describes the physical and software mechanisms that make these attacks possible, presents the main families published since 2014, and puts the countermeasures in context: hardware, operating system, compiler and application.

The subject is not limited to a single processor vulnerability. It is a class of problems in which an optimised implementation reveals, through an observable effect, information that the architectural model was supposed to keep out of reach. The best-known examples exploit speculation, caches or DRAM, while others exploit predictors, internal buffers or power consumption.

This guide is not an exploitation procedure. The mechanisms are described so as to support risk analysis, architecture review and the preparation of a product response, without providing ready-to-use attack code.

2.2 Intended audience

This guide is intended for:

- embedded and system software development teams
- architects, work package leads and product owners
- security teams assessing SoCs, RTOSes, hypervisors or TEEs
- validation and vulnerability response teams

2.3 Scope and terminology

The attacks covered here are publicly documented attacks that have shaped the understanding of processor security. They cover general-purpose processors and embedded systems that integrate application cores, caches or external DRAM. Mentioning an attack does not mean that every

processor of a family is vulnerable: vulnerability depends on the implementation, the silicon revision, the microcode, the software and the access conditions.

The literature sometimes uses diverging numbering schemes. “Spectre v1” and “Spectre v2” refer to the two most common original classes, and later suffixes and numbers do not form a universal taxonomy. The dates given below are the earliest identifiable public release or disclosure dates, not the date of private discovery. For Rowhammer, the established term in the literature is *Rowhammer*. “RawHammer” is not a commonly recognised, distinct family of microarchitectural attacks, so this guide interprets that name as Rowhammer.

2.4 Executive summary

Three ideas summarise most of the attacks presented here:

1. the architectural cancellation of an instruction does not necessarily erase the traces it left in internal structures
2. an attacker can turn a difference in latency, cache state or data into an observation channel
3. a software fix can reduce the risk without removing the hardware behaviour that causes it

For embedded systems, the first step is therefore not to blindly apply a patch associated with an attack name. One must identify the cores actually present, their execution mechanisms, the privilege and trust boundaries, resource sharing, the update model and the mitigations supported by the vendor.

3

Microarchitectural fundamentals

3.1 Visible architecture and actual implementation

An ISA defines the expected functional behaviour of a processor. It generally does not specify exact memory access times, the detailed cache policy or the state of the predictors. Two cores executing the same instructions can therefore produce the same architectural result while exhibiting different timing behaviour.

Performance depends in particular on data and instruction caches, TLBs, branch predictors, prefetchers, load/store queues and, for systems with DRAM, on bank and row management. These structures temporarily keep a state that depends on previous accesses. As soon as an actor can measure or disturb this state, it can become a channel between domains that seemed isolated.

3.1.1 Internal structures relevant to the analysis

Pipeline and register renaming. The front-end fetches and decodes instructions, then places them in structures that allow parallel execution. Renaming maps architectural registers to physical registers in order to remove some artificial dependencies. Results usually wait in a *reorder buffer* before being retired in program order. This in-order retirement guarantees a consistent architectural view, but does not erase changes already made to structures such as the cache.

Branch predictors. The processor records the history of conditional branches, the targets of indirect branches and sometimes return addresses. It may use several tables indexed by address and history bits. The exact algorithm is specific to each microarchitecture and rarely fully documented. One must therefore distinguish the generic name of a structure from its actual implementation, in particular to determine whether two domains share its state or can influence it.



Caches and TLB. A cache is organised into lines, sets and ways. Several memory addresses map to the same set, which creates contention even when they refer to distinct data. Data caches are not the only relevant structures: instruction caches, translation caches and the TLB can also reveal accesses or create contention. The measured latency depends on the cache level, the memory controller, the frequency and concurrent activity.

Load and store queues. They track pending memory accesses and writes that have not yet been propagated. They are used to resolve dependencies and to detect whether a load can reuse an earlier store. Some attacks target their values or transient states rather than the permanent content of a cache.

DRAM and memory controller. Main memory is organised into banks, rows and columns. Opening a row can speed up subsequent accesses to that row, while closing it or opening another row results in a different latency. The controller schedules requests, applies refresh policies and may provide mechanisms to prevent disturbances. These choices affect both performance and security.

3.2 Out-of-order execution and speculation

A modern core can decode several instructions at once, execute them out of order when their operands are ready, and then retire their results in architectural order. A slow memory dependency or a branch that is hard to predict increases the benefit of speculative execution: the core works ahead instead of waiting.

If a prediction is wrong, the architectural results of the transient path are discarded. However, effects already applied to a cache, a predictor or an internal queue may persist long enough to be measured. A Spectre-type attack causes a computation that depends on protected data to be executed transiently, then encodes that data into a measurable effect. The attacker does not need the transient computation to be architecturally committed.

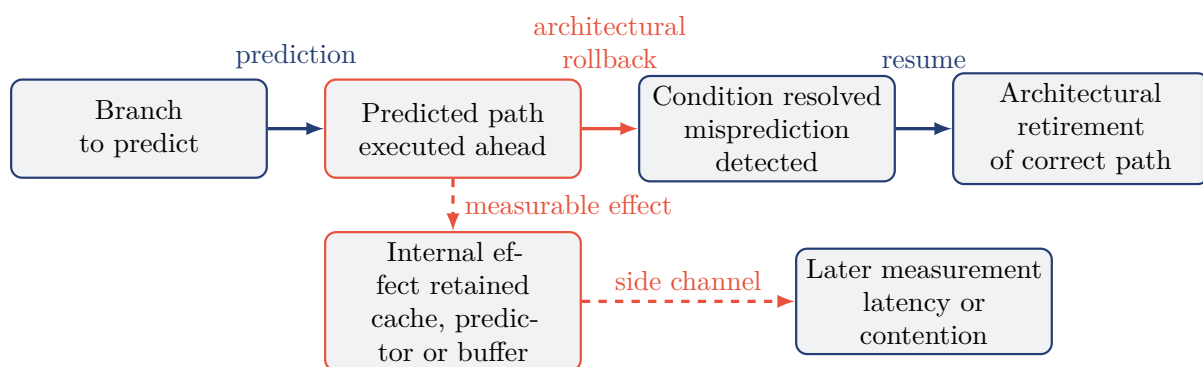


Figure 3.1: A transient instruction can disappear from the architectural state while leaving a trace in the microarchitectural state

3.2.1 Why in-order retirement is not enough

In-order retirement protects the architectural consistency of exceptions and control flow. For example, a forbidden load can raise a precise exception without its result ever being written to a

visible architectural register. This guarantee does not promise that accesses performed during out-of-order execution are invisible to every hardware resource.

Three states must be distinguished. The first is the architectural state that remains after retirement. The second is the transient state of in-flight instructions, which is cancelled on a misprediction. The third is the durable or semi-durable microarchitectural state, such as a cache line, which can survive the cancellation and influence a later measurement. Transient attacks exploit the gap between the second and the third state.

The useful lifetime of this state depends on the resolution path. A page fault, a cache miss or a chain of dependencies can delay the validation of an instruction. The attacker looks for a window long enough for an instruction that depends on protected data to reach an execution unit and then modify an observable structure. The longer the latency to wait for, the higher the risk of transient execution, but processors also impose limits on resources and speculation depth.

Not all processors speculate in the same way. A simple in-order microcontroller without a sophisticated predictor does not necessarily offer the Spectre primitives described for an application core. It can nevertheless remain exposed to other channels (timing, power consumption, bus contention, memory errors or DMA).

3.3 Common side channels

Cache access time. A line already present in the cache is usually served faster than a missing line. Flush+Reload and Prime+Probe demonstrated measurement methods that exploit shared lines and cache set occupancy respectively [74, 56].

Contention and latency. An attacker can measure competition for a cache, a memory bus, an execution port, a DRAM bank or a predictor, without directly sharing the observed address.

Power and frequency. Current, power and frequency variations can depend on the activity of execution units. In 2022, Hertzbleed showed that dynamic frequency scaling could, under certain conditions, turn data-dependent power variations into timing variations observable remotely [72].

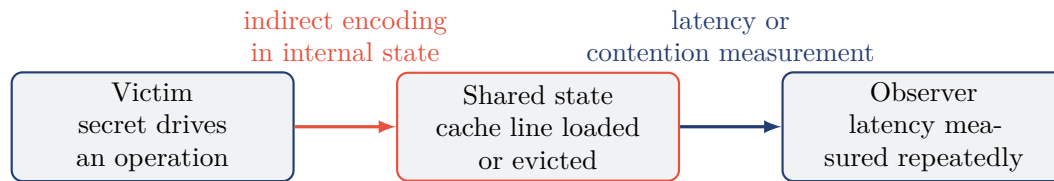
Disturbance rather than measurement. Rowhammer differs from the previous channels: the intended effect is a modification of bits in memory, not merely the inference of data through its access time.

3.3.1 Two families of cache measurement

Flush+Reload. This method applies when the same physical content is shared, for instance a shared library. The observer evicts a line from the cache, lets the victim run, then measures the time needed to reload that line. A fast access suggests that the victim brought it back into the cache. Deduplicated pages, shared libraries and enclaves using common pages can make sharing possible. The method observes a specific line with good resolution, but requires such sharing and a way to measure latencies.

Prime+Probe. This method does not require the two domains to share a memory line. The observer accesses a set of addresses that fill the ways of a cache set, lets the victim run, then measures the cost of its own repeated accesses. If the victim used the same set, some of the

observer's lines may have been evicted. The attack depends on knowing or inferring the indexing function, the replacement policy and the stability of the latencies.



The visible result is not the secret itself
but a distribution of measurements that depends on the cache state

Figure 3.2: Principle of a side channel based on a shared cache state

Noise and measurement quality. Interrupts, frequency changes, other cores, hardware prefetching and the operating system add noise. An attacker can repeat observations and aggregate them, which explains why reducing timer precision is not always enough. The risk also depends on how repeatable the secret is, how often the operation can be triggered and how many observations are allowed.

Channels without shared lines. Contention channels can exploit shared resources even when addresses are disjoint. Interference on memory banks, execution units, TLBs or internal queues can provide a coarser measurement. A threat analysis must therefore distinguish “no shared memory” from “no shared resource”.

3.4 General chain of a transient attack

A conceptual scenario has four steps:

1. **Prime** The context or the code places an internal structure such as a predictor in a chosen state
2. **Trigger** A misprediction, a fault or a slow dependency opens a transient execution window
3. **Encode** The transient path uses protected data to influence a distinctive microarchitectural state
4. **Observe** A timing or contention measurement makes it possible to distinguish states and reconstruct the information

This representation is useful for reasoning, but real attacks differ in their preconditions, their encoding primitive and their reach. A leak does not automatically amount to arbitrary code execution. It can however reveal cryptographic secrets, kernel data, information from another process or data from a virtual machine.

3.4.1 What distinguishes a leak from a full compromise

A repeated measurement may initially reveal one bit or a few bits. Recovering a complete key often requires the cryptographic operation to be reproducible, the attacker to be able to trigger

or wait for enough executions, and the noise to remain manageable. A memory leak can expose data without granting control over the execution flow. A Rowhammer corruption can modify a page, but the attacker still has to turn this modification into an exploitable effect.

Research papers demonstrate a chain under precise assumptions. To transpose the result to a product, one must check that the target gadget or path exists, the privileges available to the attacker, the resources actually shared, the timing precision and the ability to repeat the experiment. A CVE tied to a processor family therefore does not, on its own, describe the severity of each configuration.



4

Overview of published attacks

4.1 Timeline

2006 – 2014: cache channels The work of Osvik, Shamir and Tromer on caches and of Yarom and Falkner on Flush+Reload established measurement techniques reused by many attacks [56, 74]

2014 – 2016: Rowhammer The study by Kim et al. formalised in 2014 the DRAM bit flips caused by repeated activations. Subsequent work demonstrated JavaScript, mobile and system scenarios [36, 65, 70]

January 2018: Spectre and Meltdown The disclosure of Spectre and Meltdown made public the consequences of speculative execution on isolation between processes and between privilege levels [37, 45]

2018: early variants Spectre v1.1 and v1.2 explore speculative stores and read-only memory protection. Foreshadow demonstrates leaks from SGX enclaves and other L1 contexts [25, 69]

2019: memory accesses and buffers MDS, RIDL, Fallout and ZombieLoad show that data can be sampled from microarchitectural buffers, including across some security boundaries [27, 71, 62, 64]

2017: ClockScrew This attack shows that software control of power management can induce computation faults in a TrustZone secure domain [?]

2020 – 2021: new primitives LVI reverses the direction of the leak by injecting a value into a transient computation. Rowhammer research evolves activation patterns and ways to bypass defences [68, 20, 39, 34]

2022 – 2024: recent developments Retbleed, BHI, Inception and Downfall explore returns, branch history and vector gather instructions respectively. Zenbleed affects a specific AMD implementation [73, 47, 44, 52, 2]

Disclosure dates do not always match conference proceedings dates. Research teams often coordinate publication with vendors, then publish the scientific paper several months later. The dates in the timeline are those mentioned in the corresponding advisories or papers, not the general availability dates of the fixes.

4.2 Spectre: controlled speculative execution

4.2.1 Spectre v1 – speculative bounds check bypass

The first published variant of Spectre, in January 2018, typically exploits a bounds check or another conditional check. An attacker influences the predictor so that it anticipates the “allowed” path, while the actual condition will only be known later. During the transient window, the code uses an index that depends on data outside the permitted domain. A second indexed access leaves a footprint in the cache. Once the condition is resolved, the architecturally correct program has nevertheless already modified the internal state [37].

The impact is not intrinsically tied to the kernel. A website can attack a browser process in some configurations, and a process can target data accessible in the same address space or to co-resident components. Exploitation depends on the target code, the precision of the measurements and the boundaries that actually share resources.

Conditions for success. A victim code path must read or transform sensitive data before a check is definitively resolved. The attacker must be able to cause a useful prediction, influence at least one input of the speculative path and measure a resource that depends on the result. The gadget can be in a library, an interpreter, a kernel or firmware. The attacker does not need to inject instructions into the victim code, but must be able to influence the data or the context that activate the gadget.

Conceptual example. A service checks that an index belongs to a public array. If this check is mispredicted, a transient read can use the unvalidated index and obtain a value located outside the array. A subsequent operation turns this value into a choice between several memory lines. Once the branch is corrected, the observer compares latencies and deduces information about the value. The key point is that the check protects the architectural result, but does not necessarily prevent the speculative execution of the body.

Scope and gadget variants. Variant v1 is often described as a bounds check bypass, but this wording is narrower than the actual class. Privilege conditions, null values, sizes, pointers or other control-flow checks can be mispredicted. The leaked value and the encoding channel are not necessarily in the same memory object. This diversity explains why searching for known code sequences does not cover every possible gadget.

The first responses included speculation barriers placed after sensitive checks, index masking sequences whose safety is established for the ISA, and compilation techniques such as Speculative Load Hardening [46, 43]. A classic check in the C language is not automatically a microarchitectural barrier: the compiler and the processor can keep speculating. Barriers must be chosen according to the processor and inserted at the points where protected information influences an

observable access. Spectre v1.1 and v1.2, published in 2018, showed that the analysis must also consider speculative stores, store gadgets and pages marked read-only [25].

Limits of software responses. A barrier placed too early can be moved or removed if the compiler does not recognise it as a synchronisation primitive. A barrier placed too late can let a secret-dependent access through. Index masking depends on the type, the width and the exact semantics of the ISA. It must be reviewed after a change of compiler, optimisation options or microarchitecture. A functional test that only checks the return value does not detect a transient leak.

4.2.2 Spectre v2 – branch target injection

Spectre v2, also called Branch Target Injection (BTI), manipulates indirect branch predictions. A victim executes an indirect branch, and the predictor, influenced by an earlier context, proposes a target that is controlled by or useful to the attacker. If the predicted path transiently executes a “gadget” that depends on a secret, this secret can be leaked through a side channel. Depending on the implementation, the sharing of some predictors between contexts makes attacks possible across processes or privilege levels [37].

Target prediction and gadget. An indirect branch is a jump whose target is computed at runtime, for example a function call through a pointer. The processor tries to predict its destination from previous branches. The BTB can associate the branch address with a predicted target. An attacker trains the prediction state in one context, then triggers the execution of the victim branch. If the predictor reuses a shared or insufficiently separated state, the core can start executing the gadget before checking the real target.

Isolation between contexts. Exploitation depends on how the microarchitecture indexes the prediction tables, when they are flushed and on privilege transitions. The domains do not need to share the same address space if the transient prediction can target victim code reachable in the trusted domain. With virtualisation, the scheduler and the sharing of the physical core determine whether two guests can influence common structures.

The initial defences combined retpoline (transformation of indirect branches by the compiler), IBRS/IBPB/STIBP on some x86 generations, and barriers or protection sequences provided by vendors [67, 43]. These mechanisms have a variable cost in performance and power consumption. They are not interchangeable: retpoline does not cover every form of prediction and its effectiveness depends on the microarchitecture.

Why several mitigations coexist. Retpoline modifies the generated code to avoid some indirect branch predictions. IBRS controls prediction behaviour according to the privilege level on processors that support it. IBPB requests a prediction barrier on a chosen transition. STIBP limits the sharing of some predictors between SMT threads. The documentation of each processor specifies the safe combinations, the required microcode and the cost. An option enabled in the kernel does not imply that every transition path uses all these primitives.

Later attacks demonstrated that the history domain or the predictor state could remain relevant despite protections designed around direct targets. Branch History Injection (BHI), disclosed in 2022 and published in 2024, shows that branch history can influence an indirect prediction in contexts where earlier defences seemed sufficient. Retbleed (2022) shows that return instructions can, on some x86 microarchitectures, behave like vulnerable indirect branches. The

corresponding fixes required new mitigation choices, microcode and sometimes kernel changes [73, 47].

Evolution of bypasses. Recent mechanisms try to separate or flush not only a predicted target, but also the branch history that contributes to selecting the gadget. BHI highlights that controlling targets is not necessarily enough if the global history remains influenceable. Retbleed led to a review of how some processors predict returns and to the addition of specific software or microcode sequences. Inception shows that a threat model must include atypical training paths, not only the branches explicitly present in the source code. These results illustrate how research evolves, not a proof that all mitigations are ineffective.

4.2.3 Other Spectre variants

Spectre v4 / Speculative Store Bypass (2018). The processor may have a pending store in its store queue without yet knowing its address or its complete data. To avoid stalling, it may execute a younger load assuming that this load does not access the same address. If the assumption is wrong, the load has read a stale value. A computation that depends on this value can then leave a trace in the cache before the processor detects the dependency and replays the necessary instructions. The affected processors provide in particular mechanisms to disable speculative store bypass (SSBD). Enabling them selectively or globally can degrade performance [28].

Spectre v1.1 and v1.2 (2018). Variant v1.1 studies speculative stores triggered by a bounds check bypass. A transient store can temporarily modify a target or a structure later used by the victim. Variant v1.2 focuses on read-only protections that do not necessarily block a speculative store already in flight on the processors studied. These proposals extended gadget reviews beyond memory loads and highlighted that architectural permissions do not always describe the behaviour before resolution [25].

Spectre v3a / Rogue System Register Read (2018). This variant transiently reads privileged system registers on some implementations, then encodes the value into a microarchitectural state. It differs from the kernel memory read classically associated with Meltdown, even though both can rely on a transient privilege fault. The mitigation documented for one variant cannot be generalised without checking the vendor advisory and the instructions available to the attacker [?].

SpectreRSB (2018). The attack targets the Return Stack Buffer (RSB), which predicts the targets of return instructions. A call feeds this structure with the expected return address and the return instruction consumes it. If the return stack is empty, overflowing or polluted during a context switch, the processor may fall back on other predictors or use a residual state. This misprediction can lead the victim to transiently execute a gadget. Defences include filling or clearing the RSB on relevant transitions, but the behaviour and mitigation sequences are specific to each processor family [38].

NetSpectre (2019). This study shows that a Spectre-type channel can, under specific conditions, be exploited remotely through a network service. The victim must expose a suitable code path and the attacker must be able to repeat requests often enough to distinguish a signal from network noise. The reported bit rates are low compared with a local measurement, and the



stability of the service is decisive. The result shows that access to a precise local timer is not always required, but does not mean that every network service is exploitable [63].

Inception (2023). This attack against specific AMD processors combines prediction behaviours and a technique called “phantom speculation” to create unexpected transient paths, including around returns. It illustrates that a mitigation established for one family of gadgets does not necessarily cover every way of training the predictor [44].

About the numbering. The names Spectre v1 to v4 come from different publications and do not cover all prediction attacks. The names v1.1, v1.2 and v3a describe refinements or variants proposed by researchers and vendors. Retbleed, BHI and Inception are generally presented under their own name rather than as the next Spectre number. To determine a mitigation, one must therefore start from the vendor bulletin and the microarchitecture, then link this bulletin to the attack family described in the publication.

4.3 Meltdown and fault-related leaks

4.3.1 Meltdown (2018)

Meltdown was publicly disclosed in January 2018 and described in detail by Lipp et al. The vulnerability, present on some processor generations, allows a load to transiently access data whose read will later be declared forbidden. The fault is precise from an architectural point of view, but occurs too late to prevent some internal effects produced by the load. A second operation depends on the data read and modifies the cache state. The attacker then measures latencies and associates the lines present with possible values [45].

Scenario and impact. In the historical scenario studied, an unprivileged process repeatedly read kernel addresses mapped in its virtual address space but protected by permissions. The measurements showed that information could be reconstructed byte by byte under the experimental conditions. The availability of KASLR does not cancel the problem, because it may first be possible to discover data used to locate or understand the kernel space. The risk nevertheless depends on the exact processor model and on the presence of the relevant mapping.

Difference with Spectre v1. Meltdown does not necessarily require a mispredicted check gadget in the victim software. The privilege fault itself opens the exploited transient window. Spectre v1, conversely, uses a control-flow misprediction to transiently bypass a check in code that runs normally. Both families use a side channel, but they differ in the triggering condition and the location of the gadget.

The major software response on affected systems was KPTI (Kernel Page-Table Isolation), which further separates kernel and user-space mappings. Microcode updates and new processor designs also helped fix this class of behaviour [42, 43]. KPTI adds transitions and can have a cost, which varies depending on the availability of PCID and on the workload.

Limits of KPTI. KPTI removes most kernel mappings from the page table used in user mode, which reduces the surface reachable by the original attack. It does not turn all memory accesses into non-speculative operations, and is therefore not a general answer to Spectre. Switching between page tables can increase the cost of system calls and interrupts. The effect is reduced

on some platforms by context identifier management mechanisms, but must be measured on the actual embedded system.

Other Meltdown variants. Later work studied transient reads of data related to system registers, supervisor protections and user-mode accesses [45, 69]. The names given to these variants do not designate a uniform vulnerability present on every processor. They describe results obtained on specific models and under specific conditions. Firmware and microcode versions, page permissions and mode changes must be documented for each product.

4.3.2 Foreshadow / L1TF and related flaws

Foreshadow, published in 2018, shows that a transient read can reveal data present in the L1 cache in circumstances involving translation or privilege faults. The demonstrated variants concern in particular Intel SGX enclaves and virtualised environments, and their reach depends on isolation controls, core sharing and the virtualisation software [69].

Mitigations included microcode updates, flushing or managing the L1 cache on relevant transitions, and recommendations to hypervisors on scheduling and memory allocation [69, 27]. The approach is similar to Meltdown in that it exploits a transient state, but the assumptions and the targeted resource differ, and a fix for one does not prove the absence of the other.

4.4 Data sampling: MDS and derivatives

In 2019, several works published under the name Microarchitectural Data Sampling (MDS) studied transient data present in internal buffers of Intel processors: load buffers, store buffers and line fill buffers. RIDL, Fallout and ZombieLoad describe different sampling paths and scenarios. Unlike the basic Spectre model, the attacker does not always directly choose the address of the leaked data, but samples values that pass through an internal structure. Process, kernel or virtual machine information could be recovered under the reported laboratory conditions [27, 71, 62, 64].

The measures included microcode updates that add buffer-clearing instructions, their invocation at security transitions and, in some environments, disabling SMT [27]. The cost depends on the transitions and on the workload. These attacks are highly dependent on the core model, and their name must not be extrapolated to ARM or RISC-V cores without a specific vendor advisory.

Differences between subfamilies. The MDS names group together research on sampling data from internal structures. RIDL describes reads of in-flight data coming from various structures and load paths. Fallout focuses in particular on store buffers. ZombieLoad targets data passing through fill buffers and shows leak scenarios between contexts. The boundaries that can be crossed and the required conditions vary with processors and SMT configurations.

Why the content is not always chosen. In an MDS attack, sampling can return a value left by a recent access or a mix of transient values. The attacker often has to repeat the operation and filter the results. This differs from a carefully built Spectre gadget where data selected by the secret can directly influence a cache index. Leaks are therefore noisier, but can reach information from other logical contexts.



Operational response. Microcode updates introduced operations to clear some structures, but software must use them at the intended boundaries. The kernel sometimes chooses a clearing strategy on every context switch, only on sensitive transitions or when switching to a virtual machine. Disabling SMT can reduce concurrent sharing, at the cost of lower throughput and a different per-task power consumption. The measurement profile chosen must match the policy actually enabled in production.

Downfall (2023). Downfall revealed a transient leak on some generations of Intel processors through the Gather vector instruction and temporary register buffers. The results led to a microcode fix and to an impact analysis of workloads that make heavy use of these instructions, as the mitigation can have a measurable cost in some vector workloads [52].

Zenbleed (2023). Zenbleed is a vulnerability specific to some generations of AMD Zen 2 processors. A vector register renaming behaviour can allow a process to observe data from another logical context. This case shows that a microarchitectural data leak does not necessarily boil down to a classic cache channel. The recommended fix is a microcode update provided by AMD, and the precise impact depends on the processor and on the installed version [2].

4.5 Value injection and gadget-driven speculative execution

Load Value Injection (LVI), disclosed in 2020 and published in 2021, reverses the model of several earlier attacks. Instead of extracting a secret value through a side channel, the attacker tries to inject a controlled or influenced value into a transient operation of the victim, then to hijack its speculative computation. The demonstrations targeted in particular SGX environments and may require a demanding exploitation chain. The measures are platform-specific, and the defensive barriers or sequences can induce a high cost in some enclaves [68].

Conceptual chain. The attacker causes a situation where a transient load receives a value coming from an internal structure influenced by another context. The injected value modifies the speculative computation of a trusted victim. If this computation then accesses a resource depending on the operand, the result can be leaked through a secondary channel. This attack is therefore an injection of data into a computation, not a mere passive sampling.

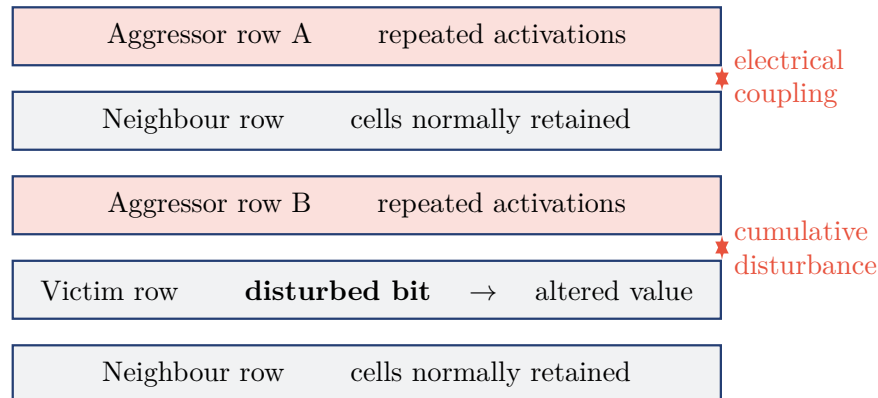
Consequences and mitigation. LVI can turn an enclave meant to protect secrets into a gadget target, under the conditions described in the publication. Mitigations may require barriers around sensitive loads and considerably increase execution time in enclave environments. The assessment must determine whether the trusted payload contains affected gadgets and whether the vendor recommends the mitigation for the processor model used. A mechanism designed against MDS leaks must not be assumed sufficient against a targeted injection.

These results highlight that a strategy based solely on “clearing caches after use” is incomplete: the attacker can act on operand values or internal structures upstream of the computation.

4.6 Rowhammer and DRAM disturbances

4.6.1 Principle and first demonstrations

In 2014, Kim et al. documented bit flips in DRAM when neighbouring rows are repeatedly activated. Since the electrical margins of the cells are limited, intense activity on so-called “aggressor” rows can disturb a neighbouring so-called “victim” row. The first demonstrations were followed by end-to-end attacks: Rowhammer.js showed a path from a browser in 2016, while DRAMMER studied exploitation on Android mobile devices of the same period [36, 65, 70].



The controller may refresh DRAM at regular intervals without necessarily counting adversarial activations of each row

Figure 4.1: Repeated activity on aggressor rows can disturb cells of a victim row

From virtual access to physical activation. A program issues reads or writes through virtual addresses. The operating system translates these addresses into physical pages, then the memory controller distributes the requests across DRAM rows and banks. The attacker must therefore obtain an access pattern that frequently reaches the targeted physical rows. Caches can prevent some requests from reaching DRAM, and prefetchers can alter the pattern. Historical work studied various ways of reducing the effect of the cache or creating sufficient memory pressure.

Practical conditions. The ability to cause a bit flip depends on the DRAM type, the internal geometry, the temperature, the refresh, the controller and the timing parameters. The mapping between physical address and row is not always public. Even if flips are possible, the attacker must place sensitive data in the victim row or take advantage of a software structure whose corruption is useful. The reliability of the flipped bits and their polarity are significant constraints when building an attack.

A bit flip is not a compromise in itself. One must obtain a useful location, predict or detect the affected bit, then make the corruption modify an exploitable object. Works such as Flip Feng Shui and RAMBleed showed impacts on systems and sensitive data. RAMBleed is particularly important because it turns DRAM disturbances into a read channel under certain assumptions [59, 39].

Impact on memory isolation. Demonstrations targeted page tables, memory management metadata, keys or cached files. Corrupting a page table entry can change the translation or the permissions if the memory layout and the flipped bit allow it. Flip Feng Shui studies the placement of data and pages to make some changes exploitable. DRAMMER demonstrated that mobile memory management constraints did not rule out all exploitation. These results depend on the OS, the allocator and the controller.

RAMBleed as a read channel. RAMBleed is not only a corruption attack. The authors exploit a statistical relationship between the values of adjacent cells and the disturbance behaviour to infer bits of a neighbouring row. The leak requires particular placement, repetition and filtering conditions. It shows however that the availability of ECC memory does not automatically turn every disturbance into a mere corrected and invisible error [39].

4.6.2 Bypassing or outgrowing the defences

The first defences relied in particular on refresh, activation count thresholds (TRR), memory placement and ECC memory. Later work showed their limits:

- TRRespass (2020) and Blacksmith (disclosed in 2021, published in 2022) produced complex, non-uniform patterns that bypass some Target Row Refresh implementations
- Half-Double (2021) showed that the effect can extend beyond the immediately adjacent rows, depending on the DRAM and the conditions
- ECCploit (2019) studied the possibility of overcoming or bypassing ECC protection in some configurations
- RowPress (2023) studied the effect of keeping a row open for a long time rather than only rapidly repeating activations

These works do not prove that all memories or all controllers are vulnerable. They do demonstrate, however, that an internal, opaque and vendor-dependent mitigation mechanism is not a universal guarantee [20, 34, 21, 16, 57].

4.6.3 Measures suited to embedded products

Possible measures include choosing qualified DRAM and controllers, keeping the recommended parameters and revisions up to date, properly sized ECC protection, enabling the available tracking/refresh features, restricting untrusted memory accesses and reducing the privileges of components that can generate heavy activity. ECC corrects some single errors, but is not an absolute defence against adversarial patterns, multiple errors or side-channel attacks. Tests must match the DRAM component and the controller actually deployed.

What to check during qualification. The hardware file must identify the manufacturer, the part number, the DRAM revision, the memory controller and the refresh parameters. Activation tracking features must be configured according to the memory profile, and their operation must be confirmed on the shipped hardware. The replacement policy and changes of supply source must trigger a reassessment, because two electrically compatible components can exhibit different disturbance behaviours.

4.7 CLKSCREW: faults induced through power management

4.7.1 Why this attack is included in the guide

CLKSCREW, published by Adrian Tang, Simha Sethumadhavan and Salvatore Stolfo at the USENIX Security 2017 symposium, is not a side-channel attack and does not rely on a transient leak comparable to Spectre. It belongs first and foremost to the family of fault injection attacks. It nevertheless deserves its place in a microarchitectural analysis in the broad sense, because it hijacks the processor’s internal frequency and voltage management mechanisms to alter the result of computations performed by a trusted domain [?].

The distinction matters for terminology. Here, the microarchitecture is not merely measured by the attacker, as in a cache channel. Its electrical and timing operating point is manipulated until some operations no longer respect their operating margins. The property under attack is the integrity of the computation, rather than confidentiality through observation.

4.7.2 Background on DVFS and timing margins

Dynamic voltage and frequency scaling (DVFS) adjusts the processor’s operating point to the workload in order to save energy or control temperature. A higher frequency reduces the time available for signals to settle within each cycle. A lower voltage slows down some logic gates. Hardware and firmware must therefore maintain validated voltage-frequency pairs, often called operating points.

The timing margin is the difference between the maximum delay of the slowest logic path and the available clock period. If the period becomes too short for the actual delay, a register can capture an unstable or incorrect value. The processor does not necessarily have a general mechanism to detect or correct this defect. The fault can then propagate into a cryptographic computation, a check or a control decision.

4.7.3 How CLKSCREW works

The systems evaluated by the authors exposed power management interfaces to normal-world software that were powerful enough to influence the processor’s operation. The attacker had a malicious kernel driver in the Android normal world. It manipulated the operating point while sensitive code was running in ARM TrustZone. By pushing the processor outside its safe operating conditions, it caused computation errors without physical access or external fault injection equipment.

The interest of the attack comes from the fact that TrustZone normally isolates secure software through execution levels and hardware controls. Yet an electrical resource or a DVFS mechanism shared across the system can be driven from a less privileged domain if the software interface does not enforce the security boundaries. The secure domain keeps its memory permissions, but it cannot guarantee the correctness of the hardware if another domain changes its operating point.



4.7.4 Demonstrated impacts

The authors demonstrated on commercial ARM/Android devices that a malicious kernel driver could extract cryptographic keys protected by TrustZone. They also showed a privilege escalation that allowed unsigned code to be loaded into the TrustZone domain. These results do not mean that every ARM processor or every TrustZone implementation is vulnerable. They depend on the access controls on power interfaces, the organisation of the regulators and the error detection mechanisms of the SoC tested [?].

The prerequisite is stronger than a mere application account: in the published scenario, the attacker controls a normal-world kernel driver. This position remains realistic in a model where the rich kernel can be compromised by a software vulnerability or a malicious update. CLKSCREW then turns a compromise of the normal world into an attack on the integrity and confidentiality of the secure domain.

4.7.5 Protection measures and limits

The first measure is to treat frequency, voltage and regulators as security resources. DVFS programming interfaces must not be reachable without mediation by an untrusted driver. The secure monitor, the firmware or a hardware controller must enforce the validated operating points, the frequency limits and the changes allowed while a trusted service is running.

Complementary measures include checking DVFS transitions while secure tasks are running, isolating or arbitrating shared regulators, detecting voltage and frequency violations, and integrity checks on sensitive computations. A cryptographic routine can sometimes detect a fault through redundancy or result verification, but these mechanisms must be designed against an attacker able to cause adaptive errors. A second identical execution is not enough if the faulting party can target both executions in a correlated way.

Software locking is only useful if it is enforced by a root of trust and cannot be modified from a lower privilege domain. Unlock paths, resume from sleep, firmware updates and performance mode changes must also be checked. The vendor advisory and the SoC documentation remain decisive, because voltage-frequency limits and regulator controls are specific to each component.

4.8 Timing channels outside transient execution

Not all microarchitectural attacks rely on a misprediction. Cache channels exploited against cryptographic implementations predate Spectre and Meltdown. An encryption routine can for example look up tables whose address depends on a key byte. If access times differ depending on whether the line is present in the cache or not, an observer can accumulate measurements and eliminate the keys that are inconsistent with the observations. The study by Osvik, Shamir and Tromer analysed in particular cache attacks and their countermeasures for AES [56].

Hertzbleed (2022). Processors adjust their frequency according to power and thermal limits. Hertzbleed shows that a data-dependent power effect can, under certain conditions, change the operating frequency and therefore the execution time observed remotely. A classic protection against local power measurements is not sufficient if a remote service allows a sensitive operation to be repeated and its duration to be measured [72].

Defensive measures. Constant-time implementations prevent control paths or memory addresses from depending directly on secrets. Dynamic frequency, the instructions used, the generated code and the possibility of repeated measurements must also be taken into account. Masked algorithms or blinding sometimes make observations less directly exploitable, without constituting a proof of constant time. Validation must be carried out on the final processor, with the product's frequencies and thermal conditions. The countermeasures and their limits are described in the work on cache attacks and Hertzbleed [56, 72].

5

Impacts on an embedded system

5.1 Cross-reading of attack families

Family	Exploited state	Intended effect	Priority defence
Spectre v1	Conditional branch prediction	Transient execution of a gadget activated by controlled data	Validated barriers or masking around sensitive accesses
Spectre v2, BHI	Target prediction and branch history	Transient hijacking of an indirect branch or a return	Microcode, compiler sequences and predictor isolation depending on the processor
Meltdown, Fore-shadow	Transient load despite a fault or a translation failure	Read of privileged data still reachable by the microarchitectural path	Silicon fix and mapping separation with KPTI if required
MDS, Downfall, Zenbleed	Temporary buffers or registers	Sampling or exposure of data from another context	Microcode and structure clearing documented by the vendor
LVI	Load value injected into a transient computation	Influence on the computation of a trusted context	Platform-specific sequences and gadget review
Rowhammer	Physical state of DRAM cells	Flip or inference of bits in main memory	Qualified DRAM, suitable refresh and ECC protection
CLKSCREW	Voltage-frequency operating point of the processor	Computation faults in a secure domain	Mediated DVFS access and protected operating points
Timing channels	Cache, power, frequency or contention	Statistical inference of sensitive data	Constant-time algorithms and limitation of exploitable sharing

Table 5.1: Families differ in their hardware primitive and require suitable mitigations

This table is meant as guidance, not as a compliance verdict. A mitigation can cover several attacks, but its effect always depends on hardware revisions, software versions and the transitions

involved. Conversely, the presence of a weakness in a processor does not necessarily imply an exploitable path in the product if no relevant secret or gadget is reachable by the attacker.

5.2 Security boundaries involved

Depending on the platform, an attacker may try to cross one or more boundaries:

- user process to co-resident process
- user to kernel, hypervisor or firmware
- guest to hypervisor or another virtual machine
- rich environment to TEE or enclave
- remote application to a local or remote service exposing a channel

Sharing a physical core, SMT, a last-level cache, a predictor or DRAM influences the risk. A TEE protects architectural access to memory, but does not, on its own, guarantee the absence of side channels in the core, the cache or the memory interfaces.

5.3 Consequences and risk factors

The consequences range from the recovery of a one-off secret to the repeated leakage of cross-process or cross-VM data, including privilege escalation when the leak is combined with another weakness. The decisive factors include exposure to untrusted code, access to a timer or to a large number of measurements, hardware sharing, the presence of secrets in memory, the use of a hypervisor or a TEE, and how long the product cannot be updated.

Confidentiality. This is the most direct impact of cache channels and transient attacks. The data involved can be keys, authentication tokens, user content or firmware secrets. Their sensitivity depends on how long they remain in memory and whether they can be renewed. An ephemeral key that stays a few milliseconds can remain exposed if an attacker can trigger the path repeatedly.

Integrity. Rowhammer can modify data or a security check if the flipped bit hits a critical structure. A corrupted value can bypass a permission, change a pointer or alter dynamically loaded code. This consequence requires a useful placement and a repeatable corruption. Transient injection attacks such as LVI influence a speculative computation without necessarily changing the committed architectural state.

Availability and functional safety. A mitigation can lengthen response time, reduce throughput, cause more page table transitions or require disabling SMT. In a real-time product, the question is not only “is the fix enabled?”, but also “are the latency bounds still guaranteed?”. A security response that breaks a critical deadline can create a safety risk. Tests must therefore measure representative worst cases and check time and energy budgets.

An “embedded” processor is not automatically immune. ARM application cores used in gateways, phones, compute boards and industrial systems are likely to integrate speculation and



complex caches. Conversely, a vulnerability known on x86 must not be attributed to a Cortex-M, a RISC-V core or a given SoC without confirming the microarchitecture and the vendor advisory. An ISA label alone is not enough.

5.4 Embedded-specific considerations

5.4.1 Asymmetric cores and heterogeneous SoCs

The same SoC can combine performance cores and energy-efficient cores, or even different core generations. The kernel can migrate a task from one core type to another depending on load or temperature. Predictors and caches can be private to a core, shared within a cluster or shared across the SoC. A measurement performed on a single core therefore does not necessarily characterise every execution path.

The inventory must go down to the level of physical cores and revisions. It is also useful to record last-level cache sharing, the presence of SMT, migration policies and the isolation mechanisms offered by the vendor. Configurable processors or FPGAs with integrated cores can have different cache and predictor versions depending on the synthesis image.

5.4.2 Real-time and resource-constrained systems

Real-time systems can limit concurrent accesses, lock cache lines or prevent task migration. These choices sometimes reduce measurement noise, but can also make execution times more predictable for an observer. Periodic interrupts and background tasks can serve as a clock or synchronisation point even when a high-resolution timer is not available.

Mitigations have a measurable cost on critical paths. A barrier can increase the latency of an interrupt handler. A cache flush can delay the return to a task. Disabling SMT can reduce parallelism and change power consumption. Validation must therefore combine leak analysis, worst-case latency measurement and verification of functional deadlines.

5.4.3 DMA, accelerators and peripherals

A DMA engine or an accelerator can access memory while the processor runs a task. Non-coherent caches, the translation buffers of an IOMMU and bus contention create additional shared states. These resources are not necessarily targeted by Spectre, but can reveal or disturb accesses. A peripheral able to generate high throughput can also affect timing measurements or DRAM refresh mechanisms.

The threat model must identify the bus masters, the memory regions accessible to each of them, the coherency rules and the IOMMU controls. Process isolation through the MMU does not automatically isolate DMA accesses. Drivers, peripheral firmware and encryption engines must be examined as components that can influence memory and access times.

5.4.4 Boot, updates and lifetime

An embedded system can remain deployed for several years without frequent maintenance. The secure boot mechanism must then authenticate kernel, firmware and microcode updates, and prevent rollback to a version that re-enables a known weakness. The update chain must be able to deploy a mitigation while preserving a signed recovery mode.

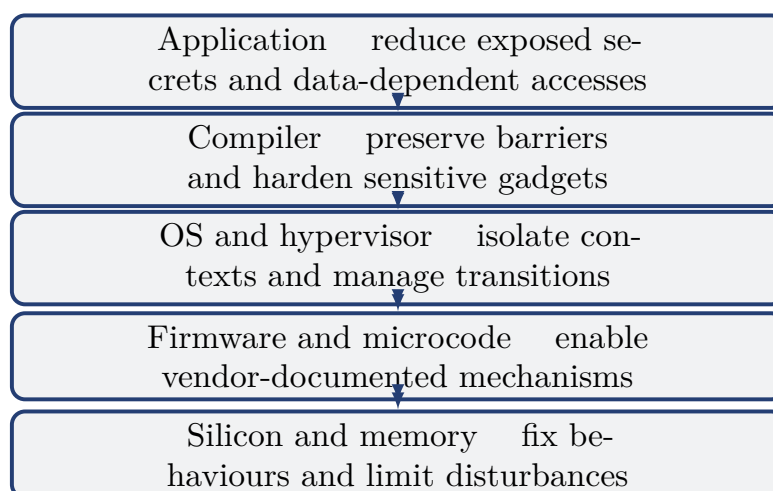
Equipment isolated from the network is not necessarily out of reach. A local attacker, third-party code, a maintenance interface, a compromised update or a guest virtual machine can be the entry point. The assessment must consider how the fix is delivered, whether it can be updated during operation and whether the equipment can verify that the new mitigation is actually active.



6

Countermeasures and analysis method

6.1 Defence in depth



Security depends on the alignment of all layers
and on the configuration actually shipped

Figure 6.1: Mitigation must remain consistent from application code down to the hardware

An effective strategy combines controls acting at different levels. The application reduces gadgets and exposed secrets. The compiler preserves protection sequences. The kernel and the hypervisor handle context transitions. The firmware enables the microcode and configures the SoC features. Finally, the silicon and the DRAM must provide the expected properties. A configuration flaw at a single layer can cancel the benefit of the others.

6.1.1 Silicon and microcode

Hardware fixes can change the faulty behaviour, isolate predictors, clear buffers or add speculation control. They provide the response closest to the cause, but do not necessarily cover every gadget

or analogous effect. Microcode can deploy mechanisms on existing hardware, provided the vendor and the platform allow it. Checking the current version is necessary: installing a system image does not guarantee that the required microcode has been loaded [43, 27].

Verification in production. Microcode can be loaded by the BIOS, by a bootloader or dynamically by the OS. The boot chain, the update policies and the image permissions therefore influence the version actually running. The product must keep evidence of the active version and avoid rolling back to a vulnerable version during a restore or a firmware update.

6.1.2 Kernel, hypervisor and firmware

Systems can apply KPTI, barriers, predictor or buffer clearing mechanisms, scheduling policies and SMT disabling options. Secure firmware must consider transitions to and from less trusted domains, exception paths, context switches and sleep/wake-up modes. Mitigation policies can depend on the processor and be disabled or restricted by configuration: this state must be documented, not merely the presence of a fix in the source code.

Memory barrier or speculation barrier. These two terms are not synonyms. A memory barrier orders the reads and writes visible to cores or peripherals. It may not prevent a dependent instruction from executing transiently. A speculation barrier or a sequence recommended by the vendor targets speculative progress and has ISA-specific semantics. On ARM, x86 and RISC-V, instruction names, guarantees and limitations differ. One must follow the documentation of the core family and not replace a speculation barrier with an instruction that only guarantees memory ordering.

Frequently forgotten transitions. The review must include task switches, privilege level changes, interrupts, synchronous exceptions, resume from sleep, guest switches and TEE entries or exits. An atypical transition path can omit the clearing applied to the normal path. Microcode update sequences must also be checked after a cold boot and after a software reboot.

6.1.3 Compiler and application

Barriers and masking sequences must be understood by the compiler and guaranteed by the ISA. A high-level language construct does not become a hardware barrier by intention. Compiler updates can introduce or remove protection sequences depending on the options. Routines that handle secrets should favour constant-time algorithms and avoid data-dependent accesses where relevant, although these measures alone do not prevent all transient attacks.

Validation of the shipped binary. Source code audits alone do not prove that the binary contains the intended barriers. One must inspect the generated assembly for sensitive functions, check the LTO and optimisation options, and maintain non-regression tests across compiler updates. Barrier attributes, intrinsics or macros have architecture-specific semantics. A memory barrier instruction intended to order accesses is not necessarily a barrier against speculation.

Secret management. Reducing key lifetimes, separating secrets between tasks and removing unnecessary copies can reduce exposure. Constant-time cryptographic algorithms generally avoid branches or memory accesses indexed by secret data, but do not guarantee the absence of leaks through frequency, power consumption or contention. The generated code must be examined on the target, taking into account cryptographic accelerators as well as their buffers and DMA.

6.1.4 Memory and DRAM

For Rowhammer, the defence must include the DRAM, the controller, the firmware and the software workloads. TRR, ECC and refresh mechanisms can vary with the manufacturer, memory density, temperatures and revision. The specification of a feature does not guarantee that its production configuration is active, nor that its properties withstand adaptive patterns.

ECC and software integrity. ECC can correct some single errors and report more complex errors, but its coverage depends on the code used, the word width and the organisation of the memory modules. Controllers can mask, correct or log errors differently. A critical product must define a reaction to the error counter, to repeated corrected errors and to uncorrectable errors. Silent correction is not a substitute for detecting abnormal activity.

6.2 Evolution and limits of mitigations

Countermeasures are sometimes bypassed because they address an instance rather than the general primitive. A protection that blocks a BTB target does not necessarily clear the global history. A barrier after a check does not cover another gadget located earlier. A refresh mechanism calibrated on a uniform pattern can miss irregular activations. Clearing a buffer on a given transition does not protect another transition.

The disclosures of Retbleed, BHI, Inception and Blacksmith are examples of this evolution. They do not make all previous mitigations useless, but require a precise check of their security assumption, of the hardware variants they cover and of the updates that complement them. The mitigation state must be tracked per processor and firmware reference, not through a mere list of attack names.

6.2.1 How to analyse an announced bypass

A new publication does not always mean that all previously fixed systems become vulnerable again. One must distinguish the scientific mechanism, the hardware actually tested and the conditions required for exploitation. A study can demonstrate a bypass on a particular microarchitecture without showing that it applies to every product using the same ISA.

1. **Identify the primitive** Determine whether the publication targets a branch target, a history, a buffer, a cache, a translation path or the DRAM
2. **Read the attacker model** Check the initial rights, the co-residency options, the number of measurements and the reachable interfaces
3. **Compare hardware revisions** Record the cores and revisions tested as well as the microcode and firmware options used
4. **Link the result to the fix** Check whether the vendor has published an advisory, an update, a recommended configuration or a statement of non-applicability
5. **Assess the product scenario** Determine whether untrusted code, exposed secrets and a measurable channel are present on the actual product
6. **Record the decision** Document the assumptions, the versions tested, the measures enabled and the residual risks

This method avoids two opposite mistakes. The first is to declare all devices vulnerable because a processor of the same family is affected. The second is to ignore an advisory because the attack name does not appear in an old compliance matrix.

6.3 Qualification procedure for a product

1. **Inventory** Record the exact references of SoCs, cores, revisions, microcode, DRAM, controllers, compilers, kernels, hypervisors and TEEs
2. **Define the boundaries** Map the secrets and the domains that share them: processes, cores, caches, predictors, buffers, buses and DRAM
3. **Consult primary advisories** Use vendor bulletins, system maintainer advisories and CVE references matching the actual hardware reference
4. **Establish the active state** Check microcode and software versions, the mitigation indicators reported by the system, build parameters, scheduling and SMT configuration
5. **Assess workload and costs** Measure the performance, latency, power and boot time penalties induced by the selected measures on representative product scenarios
6. **Test and monitor** Use vendor tests and a reproducible validation strategy, then integrate new advisories into vulnerability management and the update process

6.4 Design review questions

- Is secret data present in a domain co-resident with untrusted code?
- Are the required barriers architecture-specific and preserved after optimisation?
- Do privilege, exception, sleep or virtualisation transitions clear the necessary state?
- Does the TEE or the hypervisor share caches, predictors or buffers with adversarial domains?
- Are the mitigations enabled in the image and under production conditions?
- Can the product receive microcode, firmware and system fixes throughout its lifetime?
- Are DRAM protections verified on the memory references actually purchased?
- Are the costs of mitigation understood and accepted by the product architecture?



7

Conclusion

Microarchitectural attacks have established that architectural correctness is not enough to demonstrate confidentiality. Caches, predictors, buffers, vector units and DRAM can introduce observable or disturbable effects outside the usual functional model.

For embedded engineers, a robust response combines exact identification of the hardware, an analysis of trust boundaries, the application of the mitigations recommended by vendors, suitable software practices and a sustainable update strategy. The names Spectre, Meltdown or Rowhammer are useful historical landmarks, not binary security tests. The property to establish is more precise: on this hardware revision, in this software configuration and against this attacker model, are secrets and sensitive operations protected by mechanisms that are actually enabled and verified?

Bibliography

- [1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, 2005.
- [2] Advanced Micro Devices. Amd-sb-7008: Transient scheduler attack. Security Bulletin, 2023.
- [3] ANSI. Ansi x3.159-1989: Programming language c, 1989.
- [4] ANSSI. Ebios risk manager, 2024.
- [5] Arm Limited. Armv7-m architecture reference manual, 2023.
- [6] Arm Limited. Armv8-m architecture reference manual, 2023.
- [7] Patrick Baudin et al. Acs: Ansi/iso c specification language, version 1.22, 2024.
- [8] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.
- [9] Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security*, 2011.
- [10] BSI. It-grundschutz compendium, 2023.
- [11] Camelot-OS. Sentry kernel: verified embedded kernel primitives, 2026.
- [12] Camelot-OS. Shield: implementation of posix byte-order conversion functions, 2026.
- [13] CEA LIST. Frama-c platform for static analysis of c code, 2024.
- [14] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *USENIX Security Symposium*, 2002.
- [15] CISA. The minimum elements for a software bill of materials, 2021.
- [16] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ECC memory against rowhammer attacks. In *2019 IEEE Symposium on Security and Privacy*, 2019.
- [17] Crispin Cowan, Calton Pu, Dave Maier, Heather Walpole, Paul Bakke, Steve Beatties, Aaron Grier, Perry Wagle, and Qian Zhou. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.
- [18] Ang Cui, M. Kataria, et al. Analyzing the impact of double-fetch bugs in the linux kernel. In *USENIX Security Symposium*, 2017.



- [19] Ulrich Drepper. What every programmer should know about memory. Technical report, Red Hat, Inc., 2007.
- [20] Pietro Frigo, Enrique Meyer, Ke Xiao, Harish Chintala, Victor van der Veen, Mahyar Ghasempour, Moinuddin K. Qureshi, Khaled N. Khasawneh, and Daniel Gruss. TRRespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy*, 2020.
- [21] Google. Half-double: Next-generation rowhammer. Google Research, 2021.
- [22] H2Lab. Nouveau guide technique: protection mémoire physique, 2026.
- [23] H2Lab. Nouveau guide technique: stratégie de mise à jour flipflop, 2026.
- [24] Richard W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.
- [25] Jann Horn. Speculative execution, variant 1.1: Bounds check bypass store. Project Zero security research, 2018.
- [26] IBM. Powerpc book e architecture: A specification for enhanced embedded powerpc architecture, 2005.
- [27] Intel. Microarchitectural data sampling. Security advisory, 2019.
- [28] Intel Product Security Incident Response Team. Speculative store bypass. Security advisory, 2018.
- [29] ISO/IEC. Iso/iec 9899:1990: Programming languages — c, 1990.
- [30] ISO/IEC. Iso/iec 9899:1999: Programming languages — c, 1999.
- [31] ISO/IEC. Iso/iec 9899:2011: Programming languages — c, 2011.
- [32] ISO/IEC. Iso/iec 9899:2018: Programming languages — c, 2018.
- [33] ISO/IEC. Iso/iec 9899:2024: Programming languages — c, 2024.
- [34] Patrick Jattke, Michael Gruber, Filip Gąsienica, Fabian Manz, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *2022 IEEE Symposium on Security and Privacy*, 2022.
- [35] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, first edition, 1978.
- [36] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*, pages 361–372, 2014.
- [37] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*, pages 1–19, 2019.

- [38] Ehsan G. Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! speculation attacks using the return stack buffer. *arXiv:1807.07940*, 2018.
- [39] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy*, 2020.
- [40] Guillaume Le Guernic et al. Journey to a rte-free x.509 parser. SSTIC presentation, 2019.
- [41] Guillaume Le Guernic et al. From cves to proof: make your usb device stack great again. SSTIC presentation, 2021.
- [42] Linux Kernel Documentation. Page table isolation. Linux kernel documentation, 2024.
- [43] Linux Kernel Documentation. Spectre side channels. Linux kernel documentation, 2024.
- [44] Moritz Lipp et al. Inception: Exposing new attack surfaces with training in transient-execution. In *32nd USENIX Security Symposium*, 2023.
- [45] 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. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium*, 2018.
- [46] LLVM Project. Speculative load hardening. LLVM Compiler Infrastructure documentation, 2024.
- [47] Giorgi Maisuradze et al. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks. In *33rd USENIX Security Symposium*, 2024.
- [48] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.
- [49] MITRE. Cwe-362: Concurrent execution using shared resource with improper synchronization, 2024.
- [50] MITRE. Cwe-365: Race condition in switch, 2024.
- [51] MITRE. Cwe-367: Time-of-check time-of-use race condition, 2024.
- [52] Daniel Moghimi. Downfall: Exploiting speculative data gathering. Security research disclosure; subsequently presented at USENIX Security 2024, 2023.
- [53] NIST. Secure software development framework (ssdf) version 1.1, 2022.
- [54] NIST. The nist cybersecurity framework 2.0, 2024.
- [55] NXP Semiconductors. Mpc5777c reference manual, 2020.
- [56] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology – CT-RSA 2006*, 2006.
- [57] Jaehyun Park, Jisung Lee, Joonwon Choi, Jaewoo Kim, et al. RowPress: Amplifying read disturbance in modern DRAM chips. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023.



- [58] Raspberry Pi Ltd. Rp2350 datasheet, 2024.
- [59] Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Flip Feng Shui: Hammering a needle in the software stack. In *Proceedings of the 25th USENIX Security Symposium*, 2016.
- [60] RISC-V International. Risc-v instruction set manual, volume i: Unprivileged architecture, 2024.
- [61] RISC-V International. Risc-v instruction set manual, volume ii: Privileged architecture, 2024.
- [62] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Fallout: Leaking data on Meltdown-resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.
- [63] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. In *Network and Distributed System Security Symposium*, 2019.
- [64] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.
- [65] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 750–757, 2016.
- [66] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls. In *ACM Conference on Computer and Communications Security*, 2007.
- [67] Paul Turner. Retpoline: A software construct for preventing branch-target-injection. Google, 2018.
- [68] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Yuval Yarom, Raoul Strackx, Frank Piessens, and Elie Bursztein. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2021 IEEE Symposium on Security and Privacy*, 2021.
- [69] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Yossi Yee, Elie Bursztein, Jean-Philippe Picod, Kyriakos Kallergi, Kari Kostiaainen, Daniel Gruss, et al. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*, 2018.
- [70] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *2016 IEEE Symposium on Security and Privacy*, 2016.
- [71] Stephan Van Schaik et al. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy*, 2019.

- [72] Yingchen Wang, Martin Kösters, Seán O’Connell, et al. Hertzbleed: Turning power side-channel attacks into remote timing attacks on x86. In *Proceedings of the 31st USENIX Security Symposium*, 2022.
- [73] Ingo Wikner and Kaveh Razavi. Retbleed: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium*, 2022.
- [74] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.



License

This document is distributed under the **Creative Commons CC BY-NC-ND 4.0** license (Attribution – NonCommercial – NoDerivatives).

Full license text: <https://creativecommons.org/licenses/by-nc-nd/4.0/deed.en>