Beyond the Scope of This Paper: Practical Flash-Accelerator and Cache-Collision Timing Attacks on Microcontrollers

Colin O'Flynn
Electrical and Computer Engineering Department
Dalhousie University
Halifax, Canada
Email: colin@oflynn.com

***Abstract*—Cache and cache-collision attacks against AES are well-known since 2005, but their application to small (Cortex-M sized) microcontrollers has been limited in prior work, and in many cases the devices are assumed to be immune when they do not have a cache. However, the flash architecture of almost all microcontrollers introduces the same timing collision required for the well-known attacks and as a greatly underestimated channel.**

Recovery of AES keys using as little as 750 traces will be demonstrated on six popular cryptographic libraries across eight architectures and over twenty devices. Practical recovery on high-level applications is demonstrated against SPI bootloaders, Ethernet applications, CTAP2 over USB, and 802.15.4. wireless protocols.

In addition, a new open-source tool called FlashWhisperer is introduced to assist with evaluation of microcontrollers, cryptographic libraries, and applications. Countermeasures already exist in the form of bitsliced implementations, and this work also discusses other countermeasures that existing libraries offer.

1. Introduction

This is a paper which should have been written fifteen years ago, and builds upon work which is more often used in "current" papers as a historical reference. But, as the authors[1] of *this* paper will argue, we may be too quickly ignoring important work that was left as the metaphorical "exercise for the reader". In chasing novelty, are we missing practical and important attacks that have been more powerful than realized?

The cache-timing attacks on AES that form the core of this paper are twenty years old. Bernstein's first-order attack [1], Osvik, Shamir and Tromer's access-driven attacks [2], Bonneau and Mironov's cache-collision attacks [3] and Acıiçmez, Schindler and Koç's remote time-driven attack [4] established the problem on processors: a shared, set-associative, tagged cache; an attacker who can evict lines or observe their eviction; and a table-based cipher whose lookups depend on the key.

1. This draft uses "we" and "authors" due to anonymous planned peer review submissions being a single-author paper.

A bare-metal microcontroller which has no or a very limited cache would seem to be thus unexposed, and the assumption that small embedded processors such as the ubiquitous Cortex-M class devices is not vulnerable is seen in vendor documentation. But this is not true, and in 2010, Gallais, Kizhvatov and Tunstall implemented AES on a precursor to the Cortex-M, an ARM7TDMI, and noted that its Memory Acceleration Module is a single 16-byte line, and observed that *an attack would be possible where cache hits show where two adjacent memory accesses load the same cache line*, and wrote: *but this is beyond the scope of this paper* [5]. What lies beyond the scope of [5] will now be laid out, and the implications made clear.

The leakage observed in [5] occurs as the flash for most microcontrollers is arranged in a wider array than the bus, typically this is an explicit optimization. The flash may be 64, 128, or 256 bits wide for a device with a 32-bit instruction and data bus. Thus some *latch* stores a flash line and allows the flash to prepare for the next read cycle, which means a read within the latch is faster than a read which requires another flash row to be loaded.

This is the fundamental observation that is at least fifteen years old. This same observation has been raised in other work, including tools for flagging non-constant time cryptography in embedded systems [6]. But without an actual attack demonstrated, was there a practical impact?

To the authors knowledge, no realistic (application level) demonstration has brought this attack the attention it deserves[2]. This paper will close this gap to demonstrate key recovery in standard embedded cryptographic libraries and implementations in as few as 750 traces (operations) *without* key enumeration. This will be broadened to over 20 different microcontrollers, including architectures such as Arm Cortex-M, PowerPC, RISC-V, and MIPS32. Six different libraries will be demonstrated to leak in different configurations, and multiple application-level leakages will be demonstrated. This includes key recovery across Ethernet, USB, SPI, and 802.15.4 RF links. These application-level

2. Due to the long standing knowledge of the attack and fast advancement of LLMs, it is assumed some demos may exist of this. The authors were not aware of any during development of the paper. As we will also demonstrate, the fast advancement in LLMs opens up new challenges in what to consider novel research.

attacks are demonstrated against specific implementations of standard protocols or applications, including SNMPv3, vendor "secure bootloaders", CTAP2, and 802.15.4 (6LoWPAN). No quarter will be given.

As will be discussed in Section 3, results can very widely based on small changes in part initialization (flash and cache settings), library and linker settings, and even layout of code in memory (which may change between compilations). This may have contributed to the lack of prior concern for the attack, since it may have appeared to be a non-exploitable leakage in some devices. As we will also demonstrate, cache-timing countermeasures present in libraries may be effective on one part, but ineffective on closely related parts, causing confusion to developers and users.

1.1. Contributions

The main contributions of this paper are as follows:

- A "novel" channel using the flash row register or accelerator for timing attacks of AES on cacheless microcontrollers.
- A robust study of microcontroller architectures to demonstrate both the flash and cache attack across multiple architectures with widely used embedded libraries including Mbed TLS (now TF-PSA-Crypto).
- A robust study of settings within microcontrollers including cache and flash timing settings.
- Demonstrations of the attack in practical scenarios, including on wired (Ethernet, SPI, USB) and wireless (802.15.4) protocols using off-the-shelf development boards and applications. This includes negative results where the attack is not possible.
- A new open-source tool, FlashWhisperer, to detect and validate the leakage on embedded systems.
- Examples of embedded countermeasures that can be applied by developers using software AES.

All code for all devices and attacks is also made available. This also includes additional instructions and guidance for LLM-based tools to rapidly port the evaluation to other target devices.

1.2. Threat Model

The attacker observes the total time of an AES operation and knows the corresponding ciphertext (for the last-round attack) or plaintext (for the round-1 variant). They do not execute code on the victim, do not share a cache with it, and do not prime or probe anything. The key is fixed across the capture.

The time measurement is demonstrated in five ways, with increasing distance (and thus timing jitter) to the victim:

- An on-chip cycle counter: the characterization instrument.
- A bus edge: a logic probe on the chip select of a serial flash or UART, the victim code generating the bus edges.
- A transport: a special "attacker" USB host timing a FIDO2 authenticator.
- A network: a special "attacker" microcontroller times Ethernet frames to a connected host, both directly and through a network switch.
- A wireless link: a passive observer measures timing through wireless protocol responses.

The authors also emphasize that the attack is *less* dangerous in practice, as the attack is only applicable to software implementations. When the attack *could* have first been published (fifteen years ago) hardware accelerators were uncommon in low-cost embedded microcontrollers, and this paper would have been a major concern for implementation authors. Although hardware accelerators are now a common feature of even low-cost microcontrollers, the authors will demonstrate that libraries continue to ignore the hardware accelerator giving rise to unintentional flash-timing leakages in what could be otherwise secure modern devices. And many low-cost general-purpose microcontrollers devices still do not have AES accelerators.

1.3. Organization

The paper will begin with a short summary of cache and cache-collision attacks in Section 2, as well as summarizing the microcontroller flash layout that defines the channel. This will be made clear by then providing AES collision results across multiple microcontrollers in Section 3, including various factors we evaluated including both device-level and firmware-level. We will then discuss the specific instrumentation created for this work in Section 4. This will then lead into more practical results, starting with various cryptographic libraries in Section 5, and then higher-level applications in Section 6.

2. Background

We will first discuss the known cache and cache-collision attacks, before discussing some of the required observations about the configuration of the embedded flash and cache interactions.

2.1. Cache and Cache-Collision Attacks

For a table-based AES, two lookups $T[n]$ and $T[k]$ fall in the same memory line exactly when $[n/\delta] = [k/\delta]$, where δ is the number of table entries per line. We can consider learning that an a value n is on a given memory line as learning only the higher-order bits of n, since both n and k share the same higher-order bits. This is equivalent to a *right-shift* by s, noted with $>> s$ in the following.

Assume a known-plaintext attacker that thus knows p (with byte i being p_i), and is trying to recover k. Averaging total encryption time over many random inputs, grouped by the value of $(p_i \oplus p_j) >> s$, produces a dip at the group where $(p_i \oplus p_j) >> s = (k_i \oplus k_j) >> s$, which is where the two lookups collide [3].

TABLE 1. COMPARISON OF FIRST AND LAST ROUND ATTACKS

Measurement	δ	Cells tied with min	Distinct cell values
first-round	8	2048	32
last-round	1	256	160
last-round	8	1	6267

In the case of a known-ciphertext, the attacker is able to exploit the missing mix column operation in the last round of AES. The collision thus becomes $S^{-1}[c_i \oplus k_i] >> s = S^{-1}[c_j \oplus k_j] >> s$ where $S^{-1}[n]$ is the AES S-Box lookup (the attack is applicable to both S-box and T-Table implementations, but for this quick discussion the S-Box version is shown). This is the *expanded final round attack* described by Bonneau and Mironov in [7]. These attacks often leave some bits unknown (residual), where round-2 collisions may be used to close these residuals [2], [4].

Crucially, the last-round attack has the advantage that the row quantization happens *after* the non-linear S-Box lookup. The first-round attacker has only linear operations to apply, which limits the number of bits that can be discovered for a given lookup layout (and thus increasing the total residuals to be recovered or brute forced). This is quickly summarized in Table 1. The first-round attack has only a limited number of distinct values, meaning a number of cells will be tied and require further work.

Note that the last-round attack with $\delta = 1$ (where the memory line has only one table entry) requires a final search of 2^8. Ideally the memory line has multiple table entries. More details on the calculations of Table 1 are available in the artifact associated with this paper.

2.1.1. Chain Cache-Collision Attacks. As discussed in 2.1 the last-round attack provides the most direct and valuable results. But in many practical scenarios this is not possible - for example in general AES-CTR mode may only expose the input of the AES block, unless both plaintext and ciphertext are known. Another problem is that the measured oracle may aggregate many block operations, and may be further diluted by the measurement quantization.

Recovery then runs from the input side as a ladder (or chain): each stage tests one controlled input byte against its neighbour in the read order, gated against a null drawn from the same capture, and the surviving relations chain into the key with the uncontrolled bytes left as residual. That is to say during the capture as bytes are recovered they are fixed in capturing the next stage. Section 6.3 uses this through thirteen stages, and extends this to a special matched filter closing the residual to reach a key-space size that can be enumerated quickly.

2.2. Microcontroller Flash Layout

Microcontrollers use non-volatile flash memory for instructions and data constants. Flash memory is more space efficient than RAM, thus typically microcontrollers have a smaller amount of RAM than they do flash memory. Flash memory accesses are also often slow, and as a result most microcontrollers contain two specific settings or optimizations specific to flash memory.

The first feature, is that when accessing flash memory the access time is per-row. On a typical 32-bit microcontroller, the effective speed can be improved by reading a wider row into a *row latch*. Typically, this may be 64-bit, 128-bit, or 256-bits meaning a 2, 4 or 8 word read. Many variations of this basic latch may exist, including for example a separate latch for the data and instruction path, or a second latch that prefetches the next row.

In the most basic form (the row latch), this is below the cache or other core-level optimizations. The row latch is effectively *part* of the flash memory itself. A read of an address outside the currently stored row will stall the core until the new row is read out.

The second feature is the introduction of *wait states*. Many microcontrollers have a user-configurable wait state, which must be set to an appropriate value based on the core frequency. At low frequencies, *no* wait-states are needed. This means at lower frequencies there *may* be no observable effect of the flash latency, since the flash memory can read arbitrary addresses in sufficient time. The exact same device at a higher frequency needs wait states, which means the effect of the row register can be observed more readily. The optimistic reader is cautioned that we will quickly discover some devices have other effects that mean even at no wait state, exploitable leakage still exists.

The inclusion of the row register along with normal caches results in many permutations of different setups for otherwise similar microcontrollers. Figure 1 shows the basic layout most commonly used, where dash lines or boxes indicate optional features. On the right-hand side, starting from the data bus (D-Bus) many devices have some form of *data cache*. Most devices where it is present have an option to enable or disable this cache.

The *row register* feeds the data cache (or directly feeds the D-Bus when the cache is not present or disabled). Most devices have a row register, but devices that do not require wait states at the highest specified frequency (typically slower or older devices) there is no need for the row register.

Some devices further allow the row register to be disabled, which means every data access causes a flash lookup.

The instruction bus (I-Bus) also needs to access the flash memory. Again, there are many variations of this interconnection. The data cache may be a small unified cache holding both I-Bus and D-Bus access. The I-Bus may connect to the row register directly, or may have other architectures like an instruction cache between the row register and I-Bus.

3. Microcontroller Results

A short summary of the results is held in Table 2. This table is split into several different architectures, to demonstrate this is not limited to just one specific device. Of them, the popular Cortex-M is the majority of the evaluated microcontrollers. The specific device used in [5] was the

LPC2124, closing the gap completely on the work left open by showing a full recovery of AES in all implementations.

For each microcontroller, different flash and cache configurations were tested. This table shows either the most interesting, or a combination of a reset (default after reset) and what the typical vendor-supplied initialization code configures (“shipped”).

The core frequency and WS (wait state) column are often linked, where higher frequencies may require higher wait states. The geometry of the row (normally the “flash latch”) and the cache (which may be enabled or disabled) are also given.

The results are compared for three popular implementations: S-Box, 1.25 KB “compact” T-Table, and 4.25 KB “full-sized” T-Table (four separate T-Tables). The results are the number of encryption operations (“traces”) required to recover the key, where full recovery was possible without enumeration. Results listing N/16 mean that N bytes were recovered and may be possible with enumeration; i.e., 15/16 is effectively a full recovery, but 8/16 is not. “flat” means the results were time-constant, and thus no attack was possible. Where an attack fails fully (0/16), this means the results were *not* constant-time but no leakage was not in the form exploited in this paper. Typically the resulting variation is due to the instruction path interfering with the data path on some parts.

Only a few parts do not leak in any configuration, and these are typically older or simpler parts. Most parts *can* be configured to a constant-time setting for some builds, the full details are in the Appendix and associated artifact.

We can break down changes in configuration to two main factors. The first is a *gain factor* which is anything that changes the impact of a miss. The second is a *structure* which changes which accesses miss. We will break these down further into microcontroller configuration and firmware configuration.

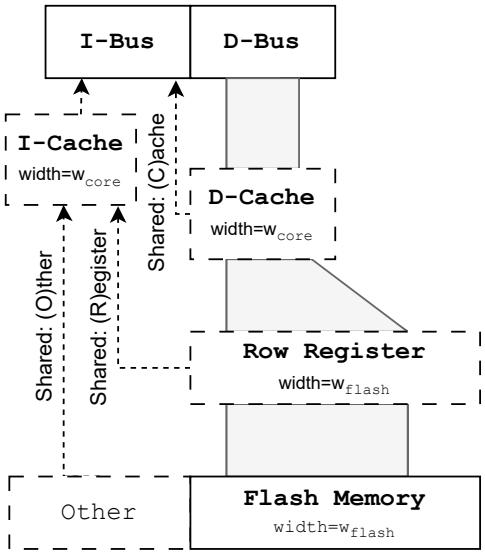


Figure 1. Flash interface to I-Bus and D-Bus of the core.

3.1. Microcontroller Factors

This section includes configurations or options that are part of the microcontroller itself. Some may be fixed in hardware, and some may have device-specific configuration.

3.1.1. Wait State. A gain factor. Due to the slower speed of flash memory, most microcontrollers have an ability to insert *wait states* during flash access. On some devices this is automatically adjusted, and not user configurable.

Zero wait states *does not* guarantee there is no channel. Where the instruction side competes for the flash array rather than evicting the row, the collision term is worth one cycle even with zero wait-state settings. This is seen on the STM32G474 and STM32L152, which both appear to “arbitrate” the wait-state latch rather than “evict”. This does not occur for all builds due to the instruction dependency – it happens with the X-CUBE-CRYPTOLIB build, but *not* with our usual calibration build.

3.1.2. Read Modes and Regions. A gain factor. Devices may have different flash read modes, such as the AT-SAMD21 having a *low-power* mode which introduces an additional wait state (but not labeled as such).

Some devices had flash mapped to both zero-wait-state and wait-state modes at different addresses (CH32V203). The implementation was not described, but the effect was that the part was constant-time (no channel) when the table was read from the zero-wait address, but there was a channel at the wait state address.

3.1.3. Accelerator, Cache, or Latch Enable/Disable. A structural factor. Devices have the option to switch on or off the normal data and instruction caches, and *some* have configuration for the “flash accelerator”. The documentation for this may not clearly separate the two, and interaction between them may be unexpected.

An example is shown in the *STM32G474* in Table 2. With the cache off, all three implementations leak due to the flash row width being 8 bytes wide. Both the S-box (1-byte table entry) and T-table (4-byte table entry) have measurable collisions.

However, turning on the 256-byte data cache allows the S-Box to become fully resident. Both T-table implementations leak as collisions are still visible, since they cannot fit in the cache.

The cache size does not guarantee residency however. On the MK21FN1M0 the same size cache results in a successful recovery against the S-Box implementation, as the remaining line holds the code carrying the S-box pointer that the cipher reloads every round. That load is itself a flash data read, so the table contends with the pointer to itself. This is more likely a fluke of compiler choices than an artifact of the part, but demonstrates that minor shifts in program can result in large shifts in leakage.

3.1.4. Clock Speeds. The core clock should not have an effect. Some devices, however, automatically adjust the flash

TABLE 2. CROSS-PART TABLE — HARDWARE, CONFIGURATION, AND WHAT THE CIPHER DOES

Part	Configuration			Geometry		Result, S-Box	Result, T-table	
PN	**note**	**MHz**	**WS**	**row**	**cache**	**256 B**	**1.25 KB**	**4.25 KB**
ARM Cortex-M								
STM32F303	reset	8	0	8 B	none	flat	flat	flat
STM32L152	reset	16	0	8 B	none	6/16[2.5k]†	flat	flat
↪ FLASH_…†	shipped	32	1		none	20k	2 500	2 500
STM32G474	reset	16	0	8 B	off	flat†	flat†	flat†
↪ DCEN 1…†	shipped	16	1		256 B	0/16†	20k	25k†
↪ FLASH_…†	shipped	16	4		256 B	flat†	25k†	12.5k†
↪ FLASH_…†	NOT a fix	16	4		off	16/16†	5 000†	1 250†
STM32L475	reset	16	0	8 B	256 B	0/16[275k]†	11/16†	12/16†
↪ ART of…†	anti-fix	16	4		off	4 000†	2 000†	750†
STM32F429	shipped	16	5	16 B	128 B	25k†	50k†	12.5k†
STM32L552	reset	16	n/s	2x8 B	off	full key†	12/16[200k]†	12/16[200k]†
↪ ICACHE…†	shipped	16	n/s		8 KB	6/8[100k]†	0/16[100k]†	0/16[100k]†
STM32F746	reset	n/s	n/s	32 B	off	8/16†	8/16†	0/16†
STM32H723	shipped	64	7	32 B	D off	12.5k†	2 500†	1 250†
nRF52840	reset	64	n/a	16 B	none	6 250†	3 125†	3 125†
RA2A1	reset	0.5	0	8/16 B‡	off	flat†	flat†	flat†
↪ FCACHE…†	shipped	48	0		= row	200k†	12/16[400k]†	1 562†
M2351	shipped	12	1	8 B	4 KB	200k†	0/16[100k]†	sigma 22†
S5D3	shipped	120	2	16 B	= row	8 000†	2 000†	1 000†
CC1352R1	shipped	24	n/a	16 B	8 KB	10k†	5 000†	8/16[50k]†
SAMD21J18A	reset	23	0	8 B	64 B	0/16[400k]†	0/16[400k]†	1/16[400k]†
↪ RWS 1…†	shipped	45	1		64 B	15/16[100k]†	12.5k†	12.5k†
SAMG55J19	reset	8	0	16 B	off	0/16[100k]†	2/16[100k]†	1/16[100k]†
↪ FWS 6…†	shipped	24	6		off	12.5k†	3 125†	780†
↪ FWS 15…†	post-prog	24	15		off	6 250†	3 125†	780†
SAME51J20A	reset	48	auto	16 B	off	5 000†	8/16[1 250]†	0/16[400k]†
XMC1100	reset	32	1	none	none	flat†	flat	flat
MK21FN1M0	reset	21	auto	16 B	256 B	100k†	11/16†	25k†
ADuCM3029	reset	26	n/a	8 B	none	20k	5 000†	2 500†
ARM classic (ARM7/ARM9)								
LPC2124	reset	15	7	16 B	off	flat	flat	flat
↪ MAMCR…†	shipped	15	7		latches	75k†	25k	5 000†
Power Architecture								
MPC5606B	reset	16	3	16 B	none	3 750†	2 500	750
Infineon TriCore								
TC234LP	reset	102	5+1	32 B	off	16.38k†	8 192†	3 072†
TI C28x								
F280049C	shipped	100	5	16 B	on	5 000	n/a†	n/a†
MIPS32 (PIC32)								
MZ2048EF	reset	200	7	16 B	4 KB	flat†	0/16[200k]†	4/16[200k]†
TI MSP430								
MSP430F5529	reset	1	n/a	none	none	flat	flat	flat
RISC-V								
CH32V203	shipped	144	n/a	n/s	none	50k†	12.5k†	6 250†

†:abbreviated here, see appendix; *flat*:one distinct cycle count; *n/s*:not attacked; *n/a*:not applicable; ‡:measured with uncertainty; *n/m*:not measured.

3.2. Firmware Factors

These factors are ones which a programmer controls, via either the source code or the compiler options.

3.2.1. Flash vs. RAM Configuration. A structural factor. All work in this paper used a "RAM null" to confirm the leakage validity, that is, moved the tables into RAM to

confirm that the measured channel disappeared. Almost all small microcontrollers that this paper is concerned with have zero wait state RAM (which is normally true SRAM, not slower DDR which opens cache-timing attacks on RAM storage).

Most of the evaluated libraries had an explicit option to store tables in RAM or ROM. The RAM tables may be generated by a function instead of simply being copied out of FLASH, which trades flash (code) space for RAM usage.

3.2.2. Address, row offset, and alignment. A structural factor. Changes in the location of the tables in memory which occur naturally due to the linker adjusting the table start address had a large impact on the leakage channel. We observed anywhere from $2\times$ – $10\times$ the traces required. For evaluation purposes, we forced different alignments by adding constant data to the build and confirming the table alignment in the resulting output (`.elf` file).

A more unexpected change is also that the address of the *instructions* also impact the recovery. As both instruction and data come from flash, there is an interplay between them, and even changes in address may result in changes to the channel. On the AURIX TC234, sliding the table across its 32-byte row moves the mean five cycles and changes nothing about attack results; sliding the cipher's code across the same 32 bytes moves the mean 1164 cycles and the outcome between all sixteen key bytes at 16384 traces and an incomplete recovery at twice the traces.

3.2.3. Optimization Changes. A structural factor. We observed different *read orders* between compilation options, which directly impacts the colliding pairs. The order may not follow the source (C/C++) code, and was always verified from the binary. When performing the attack in a black-box situation, a sweep of all orderings must be done.

Other optimization changes affect the channel, in some cases removing it entirely. On the STM32F429 for example, one build at `-O2` reloaded the table address on every iteration, which meant a data lookup that evicted the prior table row, and thus caused no collision between iterations. The same code compiled at `-O3` used a register to keep the table address between iterations, and thus left a measurable collision channel.

4. Instrumentation and Evaluation

Each part was characterized on its own cycle counter, with interrupts disabled inside the measured region, and the key loaded before every capture. At least two configurations are taken per part: the reset default, and whatever the vendor's normal initialization code uses. The latch geometry is measured by reading two locations and sweeping their location across various boundaries, which gives the granularity from where the cost steps and the depth from how far a hit survives.

In practical scenarios, an attacker will be measuring some external indicator of AES length. To validate the leakage is not an artifact of the on-chip measurement and to understand the impact of external measurements, we consider the addition of noise.

We first take results using the victim's own cycle counter as just described. The next result taken from the same device uses an external microcontroller that times a trigger edge the victim drives, which is the tightest interval available off-chip. The final measurement is an external microcontroller that times the victim's *reply* (the request-to-response turnaround) where the victim has no special timing changes.

An attack separating populations of traces by their mean is limited by the standard error of that mean, so noise independent of the key hypothesis adds in quadrature and leaves the per-collision effect alone. Traces scale as the square of the total spread:

$$N \;=\; N_0\,\frac{s^2}{s_0^2} \;=\; N_0\,\frac{s_0^2+\Delta^2}{s_0^2} \tag{1}$$

where N_0 and s_0 are the on-chip baseline, s an instrument's total spread and Δ its own contribution to it. The left form reads Table 3; the right is how the calibration below is parameterized, and can be used for other added noise comparisons.

We calibrated (1) by adding noise of known size to a real capture, where only the noise is synthetic. Over 35 points it holds to a median measured/predicted of 1.16 (0.49–1.92), and is indifferent to the noise's shape: normal reads 1.27, uniform 1.10, interval-proportional 1.27, and a slow drift correlated over roughly a thousand traces (AR(1), ρ = 0.999) 0.95; correlated drift being cheaper than white noise of the same size, since it is nearly common-mode between the hypothesis classes.

The quantization effect is the one a real off-chip instrument has, and needs more care. Scored against its nominal step it reads 1.75, which is an artifact of the label: random-phase rounding is wider than the $q/\sqrt{12}$ the step implies, and at a nominal four cycles its *real* sd is 6.49. Against its *measured* sd it reads **1.15**, and an observer (attacker) knows only the measured sd.

One additional correction is required for the external measurements: an instrument can shrink the victim's variation as well as add to it. So we measure each one's *sensitivity*, or how far its reading moves per extra cycle of victim work, regressed on the victim's own count, and apply that correction to find the *Effective sd* used as s in Equation (1).

The external measurements were taken using a 64 MHz driver timer against a 32 MHz victim core, so a faithful instrument should read 2.00 ticks per cycle. The GPIO trigger instrument reads 2.00 and passes the victim's variation through intact. The UART reply instrument reads 1.56, so only 78 % of it ever arrives. This lower sensitivity is due to the UART baud rate timing quantizing the signal to the next UART clock edge in the internal baud rate generation block.

Table 3 shows the application of Equation (1) to derived predicted trace counts, and compares them to actual traces required. So a measured sd converts to a trace estimate

TABLE 3. TRACES TO RECOVERY ON THREE DIFFERENT INSTRUMENTS.

Instrument	Eff. sd (cyc)	Predicted	Measured	Meas/Pred
On-chip (Ref)	2.20	—	4 000	—
GPIO trigger	3.26	8 815	8 000	0.91
UART reply	9.73	78 434	100 000	1.27

before a single recovery is run, and demonstrates that the true external GPIO trigger along with external UART reply measurement remain viable in practical trace counts.

It should be emphasized that while this measurement is useful for scaling traces on the same target, the sd measurement alone cannot be used to compare *different* targets or builds.

5. Library Results

While the prior section concentrated on the results and specific source of them on the microcontroller architecture, this section is moving to demonstrate that the attack remains a threat on many current cryptographic libraries.

The focus of this section is to evaluate libraries in default or recommended configurations, an overview of the results given in Table 4. Two of those listed are *not* true libraries (tiny-AES-c + TinyCrypt) but classic reference AES implementations which are found in numerous other embedded projects.

Of the libraries, every implementation claiming constant time held. For example, BearSSL's `aes_ct` returns one distinct cycle count over a whole capture ($\sigma = 0.000$) and thus no recovery was possible.

The normal Mbed TLS `MBEDTLS_AES_ROM_TABLES` usage was defined as a code optimization in versions before 4.2.0. As a result of our disclosure the security implications of this option are now documented. Note that Mbed TLS is now renamed to TF-PSA-Crypto.

TABLE 4. EMBEDDED AES IMPLEMENTATIONS

Library + Build Settings	Table Layout	Mean Cyc.	Traces
Mbed TLS `ROM_TABLES`	4T + S	2421	5k
↪ + `FEWER_TABLES`	1T + S	2184	10k
wolfSSL	4T	4286	10k
↪ + `SMALL_TABLES`	S	5323	25k
↪ + `BITSLICED`	none	85947	–
X-CUBE-CRYPTOLIB	1T	3080	20k
↪ + `AES_SMALL`	S	2764	10k
LibTomCrypt	4T	2147	25k
↪ + `LTC_SMALL_CODE`	1T	1996	25k
OpenSSL	4T	1998	10k
↪ + `AES_CONST_TIME`	none	16179	–
GmSSL	S	6544	50k
tiny-AES-c	S	7141	50k
TinyCrypt	S	11343	50k
BearSSL `aes_ct`	none	7789	–

5.1. Claimed Attack Resistance

Mbed TLS claims resistance against a remote attacker and only limited resistance against a local one. The work presented in this paper is clearly "quasi local" due to the narrow cache collision timing. Demonstrations in Section 6.2.4 demonstrate this is applicable beyond a network switch. No specific guidance is given by Mbed TLS about cache timing attack resistance.

ST specifically names the risk of cache timing attack (CTA) in their library, with a special `CMOX_CTA_PROTECTED_DATA` linker label with a documented procedure for moving the tables out of cacheable memory. As we demonstrate it is *insufficient* to simply move them out of cachable memory, they must be moved into a *constant-time memory* such as RAM.

Finally, wolfSSL claims a cache timing countermeasure enabled by default using a table prefetch compiled. The gap is not in the claims but in the model behind them. Every mechanism these name is a *cache*, and across all libraries we see the general view that one needs to either ensure the tables are cache resident, or ensure they are not cached at all.

5.1.1. Library Countermeasure Effectiveness. ST's `CMOX_CTA_PROTECTED_DATA` linker label is described in the package for sensitive applications, which needs to be configured to *the desired non-cachable memory* and later for the user to check *for the correct non-cacheable memory start address*. For most microcontrollers this is misleading, as someone who configures it to bypass the DCACHE but still access flash memory will make the leak *worse*. This exact change is what causes the stock PIC32 library used in Section 6.2.3 to leak, as that part provides a specific non-cache memory access address that a user may assume is the secure choice.

Instead, a user needs to check there is both no cache and no flash register in the path. The ST documentation calls out the ART as a possible cache, but does not call out the more general structure of this across almost all parts. Secure usage is most easily done by moving the tables to non-cached SRAM. Some devices for example have a minimal DCACHE only on the flash path, or provide an explicitly tightly coupled (uncached) memory.

wolfSSL is the most interesting from a countermeasure evaluation perspective, as wolfSSL has two explicit defenses. The one that is on by default is not effective on the small flash row register. The `PreFetchTe` feature enabled by default walks the table once before the rounds and leaves the cipher's own lookups untouched; but most microcontrollers will not have the table become resident as they do not have a large enough cache. And if the device has a *unified* cache, instruction accesses will evict part of the table between the loading and usage.

The second countermeasure is an option called `WOLFSSL_AES_TOUCH_LINES` which *mostly* works. It rewrites every lookup to read one entry per line behind a constant-time mask, and where it fits the part the cell

TABLE 5. IMPACT OF MISMATCHING WC_CACHE_LINE_SZ

..._LINE_SZ	Res. bits	Cycles	Recoveries
disabled	–	4901	16/16 @ 25K
32 (old default)	2	74 941	12/16 @ 250K
16 (old lowest)	1	135 572	12/16 @ 750K
8 (new default)	0	—	flat (safe)

goes bit-exactly constant at the costs of 16 to 30 times more cycles. This option may be enabled for the Arm PSA constant-time build. In this mode WC_CACHE_LINE_SZ sets that stride, in versions up to 5.9.4 it defaulted to 32 on every target that is not x86-64 (beyond 5.9.4 was fixed as a result of our disclosure). If the library is run on a part with a smaller flash row, such as the STM32G474 with an 8-byte row (see Table 2), it leaves some visible index bits. This setting is not checked by the compiler, as the compiler is typically not aware of the flash register size. Thus this setting is missing the specific guidance for correct usage, and simply switching it on is insufficient.

To calculate the number of bits remaining and explore the effect on the attack, we consider first that there are two possible tables with two different strides (called s here) for a given setting of WC_CACHE_LINE_SZ, which will be referred to as CLSZ:

- 4-byte (4B) tables (Te, Td): s = CLSZ
- 1-byte (1B) tables (Tsbox, Td4): s = CLSZ $\div$ 4

For a given flash row width w then we can define:

$$\text{remaining index bits} \;=\; \max\Big(0,\; \log_2 \frac{s}{w}\Big) \qquad (2)$$

While this should be set to 8, versions up to 5.9.4 accept only 16 as the lowest value, which means any build using the standard library for parts with 8-byte flash rows will leave a signal.

Table 5 shows the impact of the default values, where both the default and lowest allowed setting leak index bits. In response to our disclosure of this, new versions of the library use a default (8) that does not leave any visible leakage.

Finally, wolfSSL has an optimized assembly backend for Cortex-M, and enabling it means the in the library ignores the countermeasure settings completely.

From the evaluation of both libraries countermeasures, it is clear that guidance on usage of cache-timing countermeasures is not sufficiently clear for end users of smaller microcontrollers. The focus on caches without considering the flash-access portion leaves users with code that may still leak timing information despite paying a very heavy performance penalty.

6. Application Results

Finally, we demonstrate that the attack is not limited solely to library functions called under the attackers control, but can be exploited when measurements occur over practical channels.

In general, applying the attack requires several conditions to be satisfied:

- Software table-based AES.
- Known last-round output (ideal) or first-round input (larger search).
- Ability to observe sufficient *different* operations.
- A timing measurement channel.
- A sufficiently long-lived key to remain interesting.

The first, software table-based AES, is listed to emphasize to readers jumping to this section that any device which uses hardware AES or bitsliced AES is out of scope. Of note the *majority* of RF/SoC devices targeting IoT protocols (such as Bluetooth and 802.15.4 cores) have hardware cryptography built in. This does not guarantee it will be used, as will be demonstrated on one attack.

The second condition is for the known ciphertext or plaintext to be exposed. The last-round attack will result in the most efficient attack, but as will be demonstrated is not *required* for a successful practical attack. In some cases the known plaintext may be partial, for example known headers used in the encryption. In the case of counter mode, knowledge of the plaintext and ciphertext allows recovery of the last-round output. If portions of the AES-CTR mode are attacker controlled (nonce or counter), this allows an effective last-round attack.

While the number of encryptions required for the "pure" measurements was reasonable, the noisier channels demonstrated in this section require more traces. This can be problematic on embedded systems, as low data rate channels may require long capture times to achieve 10^5 or more traces that are needed for key recovery.

Finally, the measurement channel must have *some* signal present around the timing of AES. Some protocols use timeslots for responses, which often means the response time is fixed and no signals remains. If the timeslot is *not* fixed, internal task switching or communication queues *do* add substantial jitter, but this simply increases the required trace count rather than making the attack impossible.

All of the above is generally also predicated on the existence of a long-lived key the measurement can be done against. The simple *existence* of a long-lived key is not enough – for example a long-lived root key may be used only when a device joins the network, and other keys are derived from this long-lived key.

As will be demonstrated, despite these conditions several examples can be found which satisfy all of the requirements in practice.

6.1. SPI Bootloader

A common structure in an embedded system is a bootloader which loads an encrypted flash image from external memory. This is found in many IoT devices, with some having very simple bootloaders which are almost an ideal construction for the attack we presented [8].

Many of these bootloaders run bare-metal, and thus there is very little timing jitter. The AES timing is measured by

measuring successive flash blocks being read over the SPI bus, as each block is decrypted there is a variation until the next one is read.

To ensure we are not artificially creating an ideal bootloader, we demonstrate an attack against ST's X-CUBE-SBSFU, which is the "Secure Boot and Secure Firmware Update" solution for STM32 devices.

We ported ST's 2_Images_ExtFlash project from the B-L475E-IOT01A to a NUCLEO-L152RE. ST's application and Secure Engine sources compile out of the pinned v2.7.0 checkout; the read/decrypt/hash loop we attack, `sfu_fwimg_no_swap.c`, is unmodified. The only security-relevant change is that ST's `se_mbedtls_config.h` ships `MBEDTLS_AES_ROM_TABLES` commented out, so the AES tables are generated into SRAM. We uncomment this option to build ROM tables, so we cannot claim the attack works on a fully stock bootloader.

6.1.1. Cipher Inputs and Measurement. The SBSFU protocol uses AES-CBC decrypt, which is an ideal target since an attacker directly targets the decrypt input (equivalent to AES last-round). An attacker is also able to choose arbitrary data using a SPI emulator or programmer, and thus has effectively unlimited observations.

Many bootloaders using AES-CTR (MCUboot, wolfBoot) cannot be attacked in this way, as they use the block input as a counter input, which limits the total number of different observations as (a) the majority of input AES bytes are fixed or unknown nonce, and (b) the counter input itself is limited to a maximum image size which on small embedded systems means the AES-CTR input only varies on the lowest 4 – 16 bits depending on the system.

On the SBSFU we are targeting, it contains a ECDSA verification over the headers, which includes a SHA-256 hash of the *plaintext*. This means the attacker requires a valid update header for the attack to proceed. For the bootloader to authenticate the body requires first decrypting it under the real key, and the tag is compared only after the loop has run over every block. As invalid ciphertext is only discovered *after* decryption of the entire image, an attacker simply resets the system and can again presents new (different) ciphertext for unlimited trace captures.

The call `VerifyFwSignatureBeforeDecrypt()` in `sfu_fwimg_no_swap.c` roughly performs the following:

```
while (fw_index < pfwImage->fwSize) {
  SFU_LL_FLASH_Read(enc_chunk, fw_index)
  SE_Decrypt_Append(dec_chunk, enc_chunk)
  SE_AuthenticateFW_Append(dec_chunk, fw_tag)
  fw_index += chunk_size
}
if (fw_tag == pfwImage->signedTag) {
    return SFU_SUCCESS
}
```

The same check will be done again when the firmware image is written to internal flash to prevent TOC/TOU attacks. For our purpose, we can measure the time from the end of one flash block read to the start of the next block read.

TABLE 6. THE PLAINTEXT TO REALISTIC (CIPHERTEXT-ONLY) ATTACK COMPARISON ON THE SPI BOOTLOADER.

	known-key evaluation	**realistic**
Target Index	10th, `RSb` round-10 state	1st, `RT0` $C \oplus K_{10}$
Needs the plaintext Distinct inputs available	yes unbounded	no unbounded
Traces for recovery	14 000	50 000
Residual Test target	none pt/ct pair	2^{23}, 18 s public `FwTag`

Note this measurement contains the all other calls outside of the AES, including the SHA-256 operation as well.

6.1.2. Results. This results in the full key from an attacker holding only one legitimate firmware update (to have a copy of the signed header), and an emulator on the serial flash chip. A firmware update alone does not provide sufficient inputs, in this example it would provide only 1024 distinct input values.

For evaluation, Table 6 shows the comparison between the more powerful known-plaintext attack and the actual attack implemented. The known-plaintext is also called the *evaluation* attack as it requires the generation of *many known firmware images*. An attacker having only a *single* known firmware image would still be limited to the 1024 blocks.

For this reason the practical attack uses either a flash emulator or simply a flash programmer to reprogram the SPI memory chip. The attack in our case was done using a simple instrument built onto another development board to monitor the SPI GPIO pin timing, and the actual SPI flash was reprogrammed.

6.2. SNMPv3 on Ethernet/IP

Simple Network Management Protocol (SNMP) is an IP protocol for monitoring and configuring network devices, typically used by switches, routers, and modems. The newer SNMPv3 adds cryptographic support, with the recommended level of `authPriv` providing authentication and encryption.

The SNMPv3 does this using a per-device key, which is derived from a shared password. Devices with the shared password can determine others keys, but recovering a key from a single device cannot recover the password. SNMPv3 is normally deployed only on trusted networks, and it is not designed to be exposed on the public internet.

The attacker being link-local is thus the reasonable threat model, as an attacker has temporarily local access to the network (either by compromise of another device or by physical access).

For this proof of concept attack, the attacker must either observe valid SNMPv3 traffic *on the wire* (a regular ethernet switch would not expose it to other ports), or they must be able to *trigger SNMPv3 requests with the in-use key*. For

the latter assumption, they can trigger replayed requests, as the replays are accepted for ± 150 s, and each replay has different AES output as a new salt is used. So very slow traffic can be amplified to quickly capture millions of operations.

6.2.1. Cipher Inputs and Measurement. SNMPv3 privacy is AES-128-CFB128 over UDP. Keystream block 1 is $E(K, IV)$ and block n is $E(K, C_{n-1})$, so $P \oplus C$ recovers AES outputs while C_{n-1} is the corresponding AES input and travels in the clear. The response plaintext is ASN.1 BER and is byte-identical on every reply, and the privacy key is localized from the passphrase against the engine ID, so it changes only when the passphrase does (no session, rekeying, or handshake). The agent draws a fresh privacy salt per response, so a byte-identical request still drives a different AES input on every exchange.

The timing instrument is a second Nucleo board using the EthTimer project developed for this work. One exchange is 1.14 ms and collection runs at roughly 500 exchanges per second.

6.2.2. STM32CubeF4 Implementation. The first implementation tested was from STM32CubeF4. This uses the lwIP stack and Mbed TLS cryptography library, and we switched on the `ROM_TABLES` flag to move tables to flash. This was built for the STM32F429 Nucleo board which contains the required Ethernet connectivity.

6.2.3. Microchip Harmony PIC32 Implementation. A second implementation was against the PIC32, using the Harmony 3 vendor example from Microchip which builds with wolfCrypt. This required two changes: the tables were explicitly moved to an *uncached* memory alias at `KSEG1`, and the flash wait state increased. Such changes could happen on a real build, as a user following guidance written in the referenced manuals may think moving tables to non-cached memory is safer, but unlike other demonstrations this was *not* a standard or recommended change for this exact build. This was built for the *PIC32MZ Embedded Connectivity (EF) Starter Kit*.

6.2.4. Results. We compare the key recovery across multiple settings in Table 7. For the STM32, the input (round 1 + 2) recovery requires $10\times$ the traces of the last-round attack which shows the typical penalty of the first-round vs last-round attack. The replay-only attack requires replay of four real captured packets. The slower recovery of the PIC32 shows that there can be substantial variation amoung similar sized embedded devices due to variations in stack and architectures.

A comparison of the actual variation for different builds is shown in Table 8. The two options which resulted in a recovered key were the only ones used in Table **??**. *Tables in* is where the S-box reads actually are: RAM, flash, or resident in the cache. z is the collision statistic pooled over all 120 byte pairs; ✓ marks a run that recovered the exact AES-128 privacy key, and — one that recovered nothing.

TABLE 7. RESULTS OF KEY RECOVERY ON SNMPV3 OVER ETHERNET

Setup	Observes	Cost
STM32, last round, driven	output, $P \oplus C$	100 k / 4.5 min
STM32, last round, replay only	output, replayed	360 k / 10 min
STM32, rounds 1 + 2	input, C_{n-1}	1 M / 45 min
PIC32, last round	output, $P \oplus C$	3 M / 2.2 h

TABLE 8. DIFFERENT COMPILER SETTINGS AND RESULTING MEASUREMENTS OF z FOR SNMPV3 BUILDS

Build	Tables In:	z	Key Recovered?
STM32: lwIP + Mbed TLS			
default	RAM	-0.58	—
`ROM_TABLES`	flash	-10.64	✓
PIC32: Harmony 3 + wolfCrypt			
`PFMWS`=3	cache	-1.33	—
`PFMWS`=7	cache	-0.34	—
KSEG1, `PFMWS`=3	flash	-3.95	—
KSEG1, `PFMWS`=7	flash	-3.39	✓

STM32 rows are 250,000 SNMPv3 exchanges each, where the recovery needs a 15/16 vote plus a 2^8 escalation on byte 0; PIC32 rows are 3,000,000 each, where the key emerges in 19,285 candidates and every null holds at all sixteen table line offsets. The two blocks' z values are not comparable with each other: different victims, pair sets and trace counts. Note that the `PFMWS` $= 3$ row carries a *larger* $|z|$ than the row that yields the key — a pooled statistic does not prove a channel; only the recovery does.

These results were done directly wiring the two boards together with an Ethernet cable, but an interesting result is also to show that typical store and forward Ethernet switches do not appear to add substantial jitter. This is presumed to be related to the relatively fast clock rates compared to the leaking signal itself, even with the store and forward operation the *relative* delay information survives. Table 9 shows the STM32 implementation on three different 10/100 network switches (A: Netgear GS308, B: TP-Link TL-SG105, C: D-Link DGS-108), where the actual link was running at 100 Mbit. Using a 10 Mbit switch resulted in a failed recovery attempt, assumed due to the slower sampling destroying the small cache timing measurement.

6.3. USB Token

The demonstration here will be on the CTAP2 implementation in RIOT on the nRF52840 USB dongle board. This is one of the default targets for RIOT built using a stock configuration.

CTAP2 (Client to Authenticator Protocol) is a protocol between a security key (authenticator) and a computer. The authenticator here is an example app provided by the RIOT framework running on a nRF52840. This protects a private key using an AES wrapper for a given *relying party* (RP), which is typically a website domain you are authenticating to using the private key.

TABLE 9. TIMING 250K SNMPV3 EXCHANGES ACROSS DIFFERENT ETHERNET SETUPS.

Setup	ΔLatency	sd μs	Result	+Depth
Cable	—	5.9	15/16	26
100Mb A	+35.7 μs	5.7	14/16	2522
100Mb B	+32.5 μs	4.7	13/16	10713
100Mb C	+35.7 μs	5.70	14/16	49427

If a RP wants to authenticate, it would send an `authenticatorGetAssertion` to the authenticator with a list of credential IDs, after which the authenticator signs the challenge with the matching private key. The RP confirms this matches the stored public key for the account.

A common optimization used to avoid the authenticator having to store an unknown number of private keys is to store a single AES key, and make the credential ID be the encrypted private key. When the authenticator receives a `authenticatorGetAssertion` request, it simply needs to decrypt the provided data, and see if it results in a valid private key. If so, it must be one created by the authenticator and can sign the request. If not (the user may have credential IDs from other tokens registered), it returns `ERR_NO_CREDENTIALS`.

6.3.1. Cipher Inputs and Measurement. The goal will be to thus to measure the time between sending a `authenticatorGetAssertion` with arbitrary data and measuring the `ERR_NO_CREDENTIALS` response. This will recover the AES key used for wrapping all credentials.

With knowledge of the key, any user who receives the list of credential IDs from a given RP is able to recover the private key and successfully authenticate.

With this USB measurement channel, the host must poll the end device (the authenticator). If the end device is not ready, it will NAK the responses. This is used as a measurement channel by counting the number of NAK requests until the response is ready, as shown in Figure 2. This provides a quantity of USB polls, where the actual time in seconds depends on the host-controlled polling period. Here that polling period is an 8–15 μs spin loop rather than the 5 ms bInterval a PC would use, so roughly 30 μs of step resolves a collision worth about half a poll on average.

The host stays silent for the first 9 ms of the reply, because every IN token raises a device interrupt that fetches through the same flash latch being measured, which, if it occurs during the AES operation would destroy the signal we are measuring. Interrupts after the AES do not impact the results (besides adding noise), so practically an attacker just needs to measure the normal response time, and set the sampling to start shortly before the normal response comes.

Every probe is pinned to the same USB frame phase, because otherwise the comparison classes sit at different phases and that difference alone can manufacture a dip. This is possible as the STM32 host (attacker board) provides a measurement of the current USB phase relative to the start of frame (SOF) delimiter the USB bus uses.

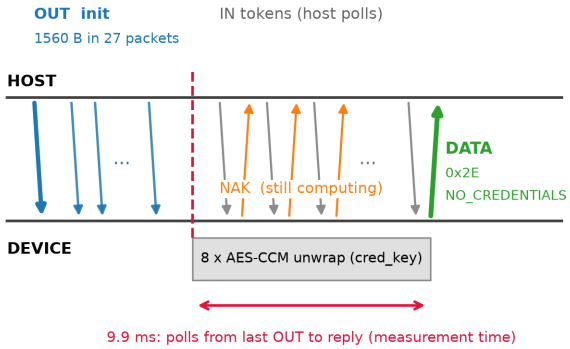
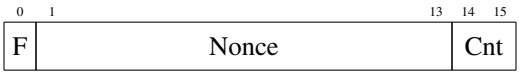


Figure 2. USB polling used as a measurement method.

While the nRF52840 USB dongle the demo is built against *has* hardware AES, the default RIOT build for this platform does not enable it, and instead uses a compact T-table implementation. RIOT wraps each credential with AES-128-CCM: CTR mode over the 136-byte record plus a CBC-MAC, with the 13-byte nonce shipped inside the credential ID itself, so the 152-byte blob has 19 block operations. The plaintext is the wrapped private key and therefore unknown, the attack must be the input side attack (round 1 + 2), and the input to round 1 is the counter block laid out as:



The layout fixes four of the sixteen AES input bytes. AES[0] is the CCM flags byte, pinned to L-1 = 1; AES[13] is nonce[12], which TinyCBOR overwrites with NUL; AES[14] is the counter's high byte, zero for every block of an unwrap; and AES[15] is the counter's low byte, stepping 0..9 rather than taking a value the attacker picks. The remaining twelve bytes are the chosen nonce, which enters every counter block and the CBC-MAC.

In the compiled read order, which defines where table collisions will occur, several of the fixed (and known) bytes do not affect our results as they will collide with a controlled byte, meaning we recover an effective key. Only k[0] is stranded as it is first in that order and its sole neighbour is k[15], also uncontrolled. This will require two additional efforts: a 'matched filter' pass which runs to recover k[15], and then enumeration on the final result.

6.3.2. Results. The default RIOT build uses in-flash tables, and a full key recovery is possible against just the cipher in 20 000 traces. Knowing the key is recoverable, the previously described USB measurement technique is then used for the timing data. The attack proceeds in three stages, summarized in Table 10.

The attack starts with thirteen collision-chaining stages, each a 64-class sweep (the 16,B row holds four 4-byte T-table entries, so a collision resolves the top six index bits) of the phase-gated USB timing oracle, recovering the top

TABLE 10. STAGES OF THE CTAP2 ATTACK RESULTS.

Stage	Time	Keyspace
13 chained collisions	19.5 h bus	2^{50}
k[15] matched filter	5 h bus	2^{44}
Offline search	12 min CPU/GPU	1

six bits of fourteen key bytes as differences. During this chaining, the recovered bytes are pinned as they are found. At the end of this phase a free anchor byte, two unresolved bits on each of the thirteen relations, and the two untouched bytes leave $2^{8+26+8+8} = 2^{50}$ key space, which is plausibly searchable.

This is further reduced with a specific 'matched filter' stage to recover k[15], the CCM counter's low byte. The partner it must collide against changes from block to block and the dip arrives split across several offset classes rather than grouping into one. Because the counter pattern is fixed by the mode and therefore known, each candidate relation predicts exactly which classes should dip and in what proportion, and scoring all 64 candidates against their own predicted patterns (a matched filter) recovers what no single class can. It is run as its own five-hour pass rather than folded into the chain because the dip is roughly half as deep and needs about three times the depth with nothing pinned. It is also optional, as the chain closes without it and this is simply reducing the key search space. The results dropping $k[15]$ from eight unknown bits to two, and the space to 2^{44}.

Because a credential ID is public and its first plaintext block is a hash the attacker chose, the residual keyspace is searched entirely offline to recover the final wrapping key.

6.4. 802.15.4 RF Wireless on Contiki-NG

802.15.4 is the lower level of many radio protocols, including ZigBee, Thread, and 6LoWPAN. It uses AES-CCM for security and authentication on radio packets. The majority of 802.15.4 system on chip (SoC) devices that integrate the radio alongside a microcontroller core have hardware AES, which is not vulnerable to the attack demonstrated in this paper.

Nevertheless, prior work has demonstrated remote side-channel attacks against software AES as a proof of concept of the threat [9]. Some implementations *do* use a separate radio chip from the microcontroller, which is what we target in this demonstration.

Specifically we use an off-the-shelf ST X-NUCLEO-S2915A1 (S2-LP radio based) with a STM32F401 Nucleo carrier as the microcontroller, running the X-CUBE-SUBG2 Contiki-NG 6LoWPAN stack with link-layer security enabled. This particular stack uses a S-box implementation of AES.

This is running at stock settings for the demo (84 MHz core, 2 wait states, flash accelerator and caches enabled). As the F401 data cache is only 128 bytes, the 256-byte S-Box will not stay resident.

In this case the attacker is passively observing the secure wireless link. The attacker is able to monitor the over the air traffic, as well as measure the time from the end of a packet sent to the victim and the transmitted acknowledgment (ACK) from the victim. The time delay includes the AES processing required for the victim to decrypt the message and check the authentication codes.

6.4.1. Cipher Inputs and Measurements. AES-CCM mode is used, where the majority of data is fixed or known (addresses, nonce, counter). This would be difficult to attack due to the lack of varying input bytes, so again an assumption about the presence of known plaintext is made.

For the 6LoWPAN/UDP protocol, this is considered reasonable as at minimum many of the header bytes contain known data such as addresses and modes. For this proof of concept the network has two nodes. A coordinator node continuously sends a constant request to the end node.

The attacker node is passively listening, and based on knowledge of the over-the-air encrypted ciphertext along with the plaintext recovers the output of the AES block. This is recorded along with a timestamp of the delay between the end of a message transmitted, and the victim returning the ACK.

6.4.2. Results. The final attack is successful using a combination of about 2.4M to 2.5M observations and a search that takes under 1 minute on recent computer. Table 11 shows these results, which were repeated for two different keys to confirm the results. The reader is cautioned that the practicality of this attack will depend greatly on the transmission rate, as if the end node was sending data only every 60 seconds, this would imply a 5 year window to collect the traces. Our test bench was sending requests between 20 – 40 requests/second.

TABLE 11. COMPARING ON-CHIP AES TO ACK TIMING ($tACK$) ACROSS BOTH LOCAL AND RF MEASUREMENTS.

Setup	Victim	Frames	+ Search space
on-chip AES	instr.	50k	2^8
$tACK$, local	instr.	800k	2^8
$tACK$, sniffer	stock, key A	2.4M	5.9×10^7
$tACK$, sniffer	stock, key B	2.5M	7.6×10^8

The cipher is a large share of the observable rather than a perturbation on it: toggling only the security level at the MAC layer moves the median turnaround from 617 µs to 1221 µs, of which ∼579 µs (∼48 600 cycles, 47 % of the secured turnaround) is AES-CCM*. The per-collision differential is a few cycles inside that, against a robust σ of 50 cycles.

Being a passive listener rather than the victim itself costs $\sigma_{\text{add}} = 1.72\,\mu\text{s}$, a $9.3\times$ trace penalty, measured by joining the two instruments on the 802.15.4 frame counter.

This is using synthetic and known data sent based on UDP messages. The complete setup is available in the

artifact associated with this paper. An example where non-synthetic data is required is next showin using a different 802.15.4 stack.

6.5. 802.15.4 RF Wireless on RIOT

RIOT implements a 6LoWPAN and 802.15.4 stack using the nRF52840. While this SoC has a hardware AES core, it was not used for the default stack before our disclosure (the same as the RIOT USB CTAP2 work).

The RIOT 802.15.4 stack ACK response cannot be used as a timing oracle, as the ACK is sent before the message authentication code is checked. For this reason a different method is used. On the RIOT stack with security enabled all 6LoWPAN PING responses come back *secured*, even from an *unsecured* request. This makes for an ideal attacker oracle, as they are able to learn the timing (from their request to response), the plaintext (since the ping response carries the payload), and the ciphertext (from the over the air message).

This allows the more powerful last-round attack to be done against a default RIOT target, compiled with the 2026.07 release.

6.5.1. Cipher Inputs and Measurements. The same AES-CCM is used as in Section 6.4.1, but as this is a ping reply the plaintext is known by design. Note that on Contiki-NG the same attack cannot be used as an unsecured ping request does not result in a secured ping response.

6.5.2. Results. It should be noted that both RIOT and Contiki-NG implement 6LoWPAN, but the setup, timing oracle, and results vary substantially. For RIOT the attack is done using a nRF52840 USB Dongle to time secure ping replies using the timer on the attacker radio. There are two post-processing steps required to measure the AES time.

First, 802.15.4 uses Carrier Sense Multiple Access (CSMA) which results in the response occurring in one of eight time slots. Because the CSMA timer is started *after* the AES operations, we can remove *nominal* timeslot location to find the residual amount that depends on the AES operation.

The second is that the ACK which occurs before the AES request becomes now a *timing reference* for our measurement. We could otherwise time our measurement from our request, but the ACK provides the lowest noise as it shows a timestamp of when the node fully received and processed the message.

These timings are shown in Figure 3. A full overview of the various backoff slots are shown in Figure 3-a. Assuming each slots adds 320 μS leaves the results of Figure 3-b, which shows Slot 0 having a slightly different offset. But this distribution is partially due to the AES timing difference. There is still some contribution of processing time, which can be removed out by using the ACK response time. The ACK response time distribution is shown in Figure 3-c.

As a result of the processing of the recovery timing, we are able to recovers the key in 70 000 – 90 0000 replies, which takes about 1.5 hours to capture.

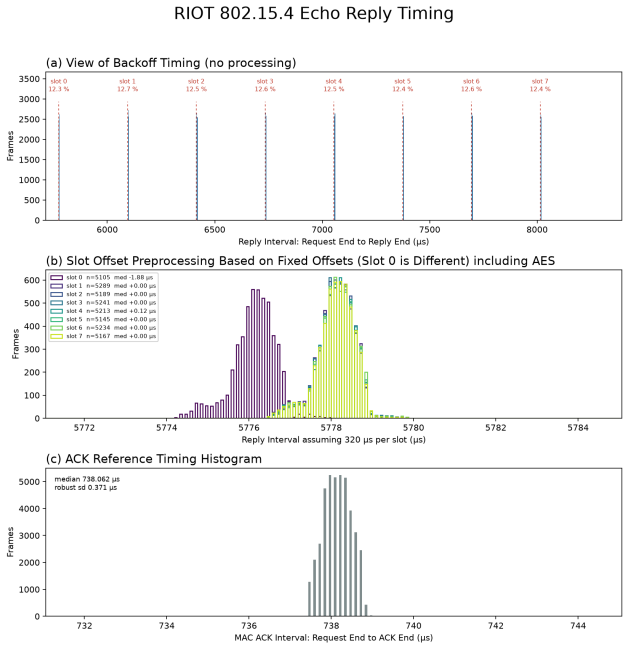


Figure 3. 802.15.4 uses a random-backoff (CSMA) that requires removing the fixed offsets, and additionally the ACK serves as a timing reference to reduce jitter further.

7. Countermeasures

As demonstrated, this attack is applicable to table-based software implementations of AES. Any hardware implementation will be immune by design, with the caveat that one should be sure a "portable library" is not using a software fall-back due to missing hardware support. Other options to remove the channel are using bitsliced implementations, moving the tables to RAM, and as final step performing timing-dependent checks at runtime.

7.1. Bitsliced Implementations

With the introduction of cache-timing attacks, non-table based implementations using vector permute or bitslicing have been proposed, along with a related but faster fixslicing [10]. Fast bitsliced Cortex-M3 and -M4 libraries under a CC0 license were released by Schwabe and Stoffelen in 2016 to provide a practical option that would close all attacks in our work [11]. This implementation *cannot* be accidentally made vulnerable by a compiler choice or later port to a different hardware platform with different memory characteristics. The primary disadvantage of bitsliced implementations is they have a strong speed to code size trade-off. This can make them impractical on small embedded systems.

7.2. RAM Tables

Some libraries used a `static const` deceleration for tables, which generally results in them placed into flash

memory. Others had an explicit toggle to switch between RAM and flash. The flash option (if not a default) may be flipped by a developer running low on RAM. Most devices tested in our work had zero-wait RAM access, meaning tables in RAM are generally a safer choice. This is of course not universal, so use of RAM tables needs to be a careful design choice validated on the hardware in use.

7.3. Timing-Dependent Detection

A final option we demonstrated as a countermeasure is using the instrumentation developed in our evaluation to perform compile time or runtime timing dependent detection. Compile time is done using a hardware run on an instrumented platform.

Runtime relies on the firmware performing a first pass during device boot to establish a baseline, and then later encryptions are compared to the baseline. While runtime detection s effective, it is practical only for bootloaders and similar bare-metal applications running with interrupts or timing variation due to instruction fetch.

8. Tooling and Artifacts

Several open-source (Apache 2 licensed) tools were developed, along with complete reference data.

8.1. FlashWhisperer

The main results of this paper have been built into a tool called *FlashWhisperer*, which includes various examples of firmware and analysis that were used to generate Table 2. This tool implements the various AES cache-collision attacks, including dealing the offsets that are normally present and other effects of real-life implementations.

This tool is available at `https://github.com/colinoflynn/flashwhisperer`.

To demonstrate the tool, a complete build matrix to build the targets used in Table 2 is also released as a separate repository. This compares different builds of AES at different part-specific settings, such as flash wait states, cache settings, and any special-purpose modes. This can be used by developers to understand the leakage of a given part with a given build type.

Finally, the complete end-to-end demos are also released in a separate repository. This demonstrates the more practical leakage and attacks, and also includes pre-recorded traces for further analysis. Work that could reduce the trace count could be an interesting topic that is beyond the scope of this paper. Additional demos which were not described in this paper include AES key recover on COAP and TLS on the same hardware as the SNMPv3 demo.

8.2. EthTimer

A stand-alone project was also developed called EthTimer, which uses a STM32 Nucleo board to provide ethernet timestamping. This provides two modes of operation: the first, which is used by UDP, simply sends packets to a host and times the responses. The more complex TCP state is handled by forwarding packets from the host through the same timing path, allowing the computer host to handle different states and protocols. This was used for performing the demo on SNMPv3 used in this paper.

This tool is available at `https://github.com/colinoflynn/ethtimer`.

8.3. Use of LLM Tooling

Extensive use of Claude Code has been made in generating the tooling used in this paper. Historically creating realistic proof of concepts has been one of the challenges to ensure industry understands the impact of academic papers. The authors of this paper were able to generate such complex demonstrations in a large part due to advancements in LLM code generation.

8.4. Disclosure

Disclosure to vendors was provided or attempted. As the attack is a side-channel this is often out of scope on embedded systems, but due to the potential for remote attacks most vendors updated documentation in response. WolfSSL has updated their library to make default applicable to tested parts by changing the default `TOUCH_LINES` value. Mbed TLS was notified of the vulnerability and updated documentation. RIOT was notified and is updating both documentation and code (including use of hardware AES when available, as well as enabling users to move tables to RAM).

9. Application to Other Algorithms

Other algorithms appear to have similar structure to AES. The condition for a last-round recovery require that many of a round's lookups must pass through a *single* table, so that a within-round collision survives the cache, and that collision must relate to the key through a *nonlinear* function. AES's final round supplies both, as $c = S(x) \oplus k_{10}$ over one S-box, with survives different implementation designs. As Mbed TLS provides other ciphers using table-based implementations, testing for leakage across other ciphers is trivial.

A Feistel key XOR before the substitution leaves the index linear in the key, so SM4 yields only the top-nibble differences of rk_{31}. A multi-table substitution layer denies the collision so ARIA,whose four S-boxes take four of sixteen bytes each, returns *none* of the sixteen key bytes at 300 000 traces despite 19.6 cycles of spread (test results on STM32F429); and a table-free layout removes it outright. Camelliafails the first two at once, being Feistel with a four-table F, and likewise yields nothing at 16.1 cycles of spread (test results on STM32F429). While neither Camellia nor ARIA were extensively analyzed in our work, no obviously exploitable leakage was found unlike AES.

10. Conclusion

Software implementations of AES are at risk of key recovery using a flash-timing leakage path. This path is not a surprise, and was first predicted at least fifteen years ago, with the actual attack work done twenty years ago. While nothing has stopped the possibility of this attack being demonstrated before, it is clear that advancements in the simplified engineering work made possible by LLM tooling are creating new challenges for those implementing cryptographic algorithms on the imperfect world in which we live. In general, it is clear that developers of embedded systems must consider how to account for physical effects of handling secret data, yet another problem to which the solution is beyond the scope of this paper.

References

[1] D. J. Bernstein, "Cache-timing attacks on AES," https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, Apr. 2005.

[2] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Topics in Cryptology — CT-RSA 2006*, ser. Lecture Notes in Computer Science, D. Pointcheval, Ed., vol. 3860. Berlin, Heidelberg: Springer, 2006, pp. 1–20, also IACR ePrint 2005/271. The copy held in related/ is the extended version, revised 2005-11-20.

[3] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *Cryptographic Hardware and Embedded Systems — CHES 2006*, ser. Lecture Notes in Computer Science, L. Goubin and M. Matsui, Eds., vol. 4249. Berlin, Heidelberg: Springer, 2006, pp. 201–215, NOTE: The two-round complexity, $\delta = 16$ residual and the byte-wide `T4` countermeasures are in the extended version of this paper.

[4] O. Acıiçmez, W. Schindler, and Ç. K. Koç, "Cache based remote timing attack on the AES," in *Topics in Cryptology — CT-RSA 2007*, ser. Lecture Notes in Computer Science, M. Abe, Ed., vol. 4377. Berlin, Heidelberg: Springer, 2007, pp. 271–286.

[5] J.-F. Gallais, I. Kizhvatov, and M. Tunstall, "Improved trace-driven cache-collision attacks against embedded AES implementations," in *Information Security Applications — WISA 2010*, ser. Lecture Notes in Computer Science, Y. Chung and M. Yung, Eds., vol. 6513. Berlin, Heidelberg: Springer, 2010, pp. 243–257.

[6] C. O'Flynn and G. d'Eon, "I, for one, welcome our new power analysis overlords: An introduction to ChipWhisperer-Lint," White paper, Black Hat USA 2018, Las Vegas, NV, 7–9 August 2018, 2018. [Online]. Available: https://i.blackhat.com/us-18/Wed-August-8/us-18-OFlynn-I-For-One-Welcome-Our-New-Power-Analysis-Overloards-wp.pdf

[7] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," Extended online version of the CHES 2006 paper, 2006, distinguished from the proceedings version by the subsections of §8 *Attack Variants* — §8.1 decryption, §8.2 chosen plaintext/ciphertext, §8.3 the two-round attack — and by the absence of the proceedings' deferral "In the online version of the paper we discuss…". Source of the 2^{51} two-round complexity, the $\delta = 16 \rightarrow 2^{80}$ residual and the byte-wide `T4` countermeasure.

[8] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn, "Iot goes nuclear: Creating a zigbee chain reaction," *IEEE Security & Privacy*, vol. 16, no. 1, pp. 54–62, 2018.

[9] G. Camurati, S. Poeplau, M. Muench, T. Hayes, and A. Francillon, "Screaming channels: When electromagnetic side channels meet radio transceivers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 163–177. [Online]. Available: https://doi.org/10.1145/3243734.3243802

[10] A. Adomnicai and T. Peyrin, "Fixslicing aes-like ciphers: New bitsliced aes speed records on arm-cortex m and risc-v," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 1, p. 402–425, Dec. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8739

[11] P. Schwabe and K. Stoffelen, "All the aes you need on cortex-m3 and m4," in *Selected Areas in Cryptography – SAC 2016*, R. Avanzi and H. Heys, Eds. Cham: Springer International Publishing, 2017, pp. 180–194.

Appendix

TABLE 12. Cross-part table, unabridged: every part, every configuration

part	config	config note	clock (meas.)	WS	miss - hit (cyc/access)	cache	table address	256 B byte S-box	1.25 KB compact T	4.25 KB 4-table	ev
ARM Cortex-M											
STM32F303											
vendor: ST; core: Cortex-M4; ISA: ARMv7E-M; flash read width: 64-bit; row-holding…											
↪	LATENCY 0, PRFTBE on	reset	8	0	n/a	n/a		flat	flat	flat	0/N
STM32L152RE											
vendor: ST; core: Cortex-M3; ISA: ARMv7-M; flash read width: 64-bit; row-holding…											
↪	FLASH_ACR 0x06 – 64-bit access, prefetch, 0 WS	reset	16	0	0.00000	n/a		flat	flat	flat	P
↪	FLASH_ACR 0x07 – 1 WS	shipped	32	1	n/m	n/a		20k	2 500	2 500	V
STM32G474RE											
vendor: ST; core: Cortex-M4; ISA: ARMv7E-M; flash read width: 64-bit; row-holding…											
↪	ART off (DCEN 0), LATENCY 0	reset	16	0	0.000	off		flat	flat	flat	0/N
↪	DCEN 0, LATENCY 1	charact.	16	1	1.000	off		12-15/16[20k]	5 000	5 000	P
↪	DCEN 1, LATENCY 1	shipped	16	1	1.000	on, 256 B		0/16	20k	25k	V / sigma
↪	LATENCY 0 with the D-cache off	fix	16	0	0.000	off		flat	flat	flat	0/N
↪	FLASH_ACR 0x00040704 – LATENCY 4, prefetch + I-cache + D-cache	shipped	16	4	n/m	on, 256 B		flat	25k	12.5k	

continued

TABLE 12, continued

part	config	config note	clock (meas.)	WS	miss - hit (cyc/access)	cache	table address	256 B byte S-box	1.25 KB compact T	4.25 KB 4-table	ev
↪	FLASH_ACR 0x00040304 – DCEN cleared, only that bit	NOT a fix	16	4	n/m	off (DCEN 0)		16/16	5 000	1 250	
STM32L475VG											
vendor: ST; core: Cortex-M4F; ISA: ARMv7E-M; flash read width: 64-bit; row-holding…											
↪	ART on (IC+DC), LATENCY 0 – ACR 0x600	reset	16	0	0.000	on, 256 B		0/16[275k]	11/16	12/16	0/N
↪	ART off (every bit), LATENCY 4 – ACR 0x004	anti-fix	16	4	4.000	off – the latch alone		4 000	2 000	750	V
↪	prefetch only, LATENCY 4 – ACR 0x104	characterisation	16	4	4.000	off – the latch alone		6 250	4 000	1 000	V
↪	DCEN 1 alone, LATENCY 4 – ACR 0x404	characterisation	16	4	4.000	on, 256 B		0/16[100k]	50k	6 250	V
↪	DCEN 1 + prefetch, LATENCY 4 – ACR 0x504	characterisation	16	4	4.000	on, 256 B		0/16[100k]	50k	12.5k	0/N
↪	tables in SRAM1 (code left in flash)	fix	16	0-4, all of them	n/a	irrelevant (swept both ways)		flat	flat	flat	V
↪	SBOX unaligned (mod 32 = 16), DCEN 1 + prefetch, LATENCY 4 – ACR 0x504	characterisation	16	4	4.000	on, 256 B – but the table spans 9 lines of 32 B, not 8		0/16[25k]	31.25k	12.5k	0/N

continued

TABLE 12, continued

part	config	config note	clock (meas.)	WS	miss - hit (cyc/access)	cache	table address	256 B byte S-box	1.25 KB compact T	4.25 KB 4-table	ev
↪	-O3, ART off (every bit), LATENCY 4 – ACR 0x004	characterisation	16	4	4.000	off – the latch alone		0/16[12.5k]	6/16	6/16	0/N
↪	-O2 with a 352-byte unrelated code addition, ART off, LATENCY 4 – ACR 0x004	characterisation	16	4	4.000	off – the latch alone		12.5k	3 125	1 000	V
STM32F429ZI											
vendor: ST; core: Cortex-M4F; ISA: ARMv7E-M; flash read width: 128-bit; row-holding…											
↪	FLASH_ACR 0x705 – caches and prefetch on	shipped	16	5	n/m	on, 128 B		50k	50k	12.5k	V
↪	LATENCY 0, caches on	the zero-wait-state residual	16	0	0.000	on, 128 B		7/16[50k]*	9/16[150k]*	10/16[150k]*	P
↪	LATENCY 0, caches off	fix	16	0	0.000	off		flat	0/16[50k]	flat	0/N
STM32F415RGT6											
vendor: ST; core: Cortex-M4; ISA: ARMv7E-M; flash read width: 128-bit; row-holding…											
↪	FLASH_ACR 0x705	shipped	7.3735	over-programmed	LATENCY cycles	on	0x08001370, off 0, shift 4	50k	n/a	n/a	V
↪	FLASH_ACR 0x005 – every accelerator bit clear	the latch alone	7.3735	over-programmed	n/m	off	0x0800136c, off 12, shift 4	4 000	n/a	n/a	V
↪	LATENCY 0	fix	7.3735	0	0.0000	n/s		flat	n/a	n/a	0/N
STM32L552ZE											
vendor: ST; core: Cortex-M33; ISA: ARMv8-M Mainline; flash read width: 64-bit;…											
↪	ICACHE off	reset	16	n/s	n/m	off		full key	12/16[200k]	12/16[200k]	V
↪	ICACHE on, WAYSEL 1-way	shipped	16	n/s	n/m	on, 8 KB, 1-way		6/8[100k]	0/16[100k]	0/16[100k]	P
↪	LATENCY 0	fix	16	0	0.000	n/s		flat	flat	flat	0/N
STM32F746ZG											
vendor: ST; core: Cortex-M7; ISA: ARMv7E-M; flash read width: 256-bit (32 B); row-holding…											

continued

TABLE 12, continued

part	config	config note	clock (meas.)	WS	miss - hit (cyc/access)	cache	table address	256 B byte S-box	1.25 KB compact T	4.25 KB 4-table	ev
↪	both L1 off	reset	n/s	n/s	n/m	off		8/16	8/16	0/16	screen
↪	D-cache on	fix	n/s	n/s	n/m	on, 4 KB		closed	closed	0/16	screen
STM32H723ZG											
vendor: ST; core: Cortex-M7; ISA: ARMv7E-M; flash read width: 256-bit (32 B); row-holding…											
↪	I-cache on, D-cache off	shipped	64.3	7	n/m	D off, I on	0x08000060	12.5k	2 500	1 250	V
↪	I + D on	NOT a fix	64.3	7	n/m	both on, 32 KB		0/16	100k	0/16[200k]	V
↪	caches off, tables in AXI SRAM	control	64.3	7	n/a	off		0/16	0/16[50k]	0/16[50k]	0/N
nRF52840											
vendor: Nordic; core: Cortex-M4F; ISA: ARMv7E-M; flash read width: 128-bit (16 B);…											
↪	ICACHECNF – any setting	reset	64	n/a	n/m	irrelevant – every setting leaks identically		6 250	3 125	3 125	V
↪	tables in RAM	fix	64	n/a	n/a	n/a		flat	0/16[20k]	0/16[20k]	0/N
↪	S-box as code, I-cache on vs off	instruction path	64	n/a	n/a	on / off		I-cache ON: 0/16 at sigma 43.479 over 300 distinct values. I-cache OFF: 16/16 at sigma…	n/a	n/a	V / sigma
RA2A1 (R7FA2A1AB3CFM)											
vendor: Renesas; core: Cortex-M23; ISA: ARMv8-M Baseline; flash read width: DISPUTED – 8…											
↪	FCACHEEN 0, MEMWAIT 0, FLWT 0	reset	0.5	0	n/a	off		flat	flat	flat	sigma
↪	FCACHEEN 1, MEMWAIT 1, FLWT 0	shipped	48	0	2.000	on, 16 B	0x6c00, off 0, shift 4	200k	12/16[400k]	1 562	V / sigma
↪	FCACHEEN 1, MEMWAIT 1, FLWT 7	over-prog	48	7	n/m	on, 16 B		12.5k	12.5k	1 562	sigma
↪	FCACHEEN 0 or MEMWAIT 0	fix	48	0 or any	0.000	off / on		flat	flat	flat	sigma

continued

TABLE 12, continued

part	config	config note	clock (meas.)	WS	miss - hit (cyc/access)	cache	table address	256 B byte S-box	1.25 KB compact T	4.25 KB 4-table	ev
M2351 (NuMaker-PFM-M2351)											
vendor: Nuvoton; core: Cortex-M23; ISA: ARMv8-M Baseline; flash read width: 8-byte fill…											
↪	HIRC, auto-tuned CYCLE 1, FADIS 0	shipped	12	1	˜2	on, 4 KB, no off switch in silicon	0x8800, off 0, shift 3	bit-exact	0/16[100k]	n/s (σ22)	V / sigma
↪	tables in SRAM	fix	12	1	n/a	n/a		flat	flat	flat	0/N
Synergy S5D3 (R7FS5D37A3A01CFP)											
vendor: Renesas; core: Cortex-M4; ISA: ARMv7E-M; flash read width: 128-bit (16 B);…											
↪	FCACHEEN 1, FLWT 2	shipped	120	2	exactly FLWT cycles, at every clock from 15 to 120 MHz	on, 16 B		8 000	2 000	1 000	V
↪	FLWT 0	fix	30	0	0.000	on		0/8[400k]	0/16[100k]	1/16[400k]	0/N
↪	FCACHEEN 0	fix	120	2	n/a	off		flat	flat	flat	0/N
↪	tables in SRAM	fix	120	2	n/a	on		flat	flat	flat	0/N
CC1352R1F3											
vendor: TI; core: Cortex-M4F; ISA: ARMv7E-M; flash read width: 128-bit (16 B);…											
↪	MODE = CACHE	shipped	24	n/a	n/m	on, 8 KB		10k	5 000	8/16[50k]	V
↪	MODE = OFF	NOT a fix	24	n/a	n/m	off		5 000	2 500	8/16[50k]	V
↪	MODE = OFF and IDCODE_LB_DIS = 1	fix	24	n/a	n/m	off		flat	flat	flat	screen
ATSAMD21J18A											
vendor: Microchip; core: Cortex-M0+; ISA: ARMv6-M; flash read width: 64-bit (8 B);…											
↪	RWS 0, cache on, READMODE = NO_MISS_PENALTY	reset	22.68	0	0.000	on, 64 B	0x2BB4, off 4, shift 3	0/16[400k]	0/16[400k]	1/16[400k]	0/N
↪	RWS 1, cache on, READMODE = NO_MISS_PENALTY	shipped	44.7	1	0.999	on, 64 B	0x6D00, off 0, shift 3, CURRENT TREE: 0x2BB4 / line_off 4 – the .probe move changed it;…	15/16 + 2^8[400k]	12.5k	12.5k	V

continued

TABLE 12, continued

part	config	config note	clock (meas.)	WS	miss - hit (cyc/access)	cache	table address	256 B byte S-box	1.25 KB compact T	4.25 KB 4-table	ev
↪	RWS 0, cache on, READMODE = LOW_ POWER	the documented power-saving mode	22.9	0	0.999	on, 64 B	0x6D00, off 0, shift 3	16/16[100k]	25k	50k	V
↪	RWS 1, cache on, READMODE = DETER-MINISTIC	fix	44.5	1	0.000	on, 64 B	0x6D00, off 0, shift 3	0/16[400k]	0/16[400k]*	0/16[400k]	0/N
↪	RWS 1, CACHEDIS = 1	fix	44.1	1	0.000	off	0x6D00, off 0, shift 3	0/16[400k]	0/16[400k]	0/16[400k]	0/N
ATSAMG55J19 (SAM G55 Xplained Pro)											
vendor: Microchip; core: Cortex-M4F; ISA: ARMv7E-M; flash read width: 128-bit (16 B) at…											
↪	FWS 0, CLOE 1, cache off	reset	7.913	0	0.00	off (SR.CSTS = 0)		0/16[100k]	2/16[100k]	1/16[100k]	0/N
↪	FWS 6, CLOE 1, cache off	shipped	23.883	6	5.50	off – SystemInit never enables the CMCC, and CMCC_ SR.CSTS reads 0 on a shipped image	0x401A1C, off 12, shift 4	12.5k	3 125	780	V
↪	FWS 1, cache off	the row to read twice	7.913	1	0.00	off		6 250	3 125	780	!=
↪	FWS 6, cache off	over-prog	7.913	6	5.50	off	0x00401a1c, off 12, shift 4	12.5k	3 125	780	V
↪	FWS 6, cache on, PRGCSIZE 2 (4 KB)	the cache flips sign on table footprint	7.913	6	n/m	on, 4 KB		0/16[6 250]	0/16[100k]	0/16[100k]	sigma
↪	FWS 0, cache on, PRGCSIZE 2 (4 KB)	zero wait states	7.913	0	0.00	on, 4 KB		0/16[100k]	0/16[100k]	n/s (σ10.4)	0/N

continued

TABLE 12, continued

part	config	config note	clock (meas.)	WS	miss - hit (cyc/access)	cache	table address	256 B byte S-box	1.25 KB compact T	4.25 KB 4-table	ev
↪	FWS 6, cache on, PRGCSIZE 3 (8 KB)	fix	7.913	6	n/m	on, 8 KB		0/16[100k]	0/16[100k]	n/s (σ46.9)	0/N
↪	FWS 15, CLOE 0, cache off	post-prog	23.9855	15	n/m	off – read off silicon, and the reset() docstring did not say so	0x401A1C, off 12, shift 4	6 250	3 125	780	V
ATSAME51J20A (SAM E51 Curiosity Nano)											
vendor: Microchip; core: Cortex-M4F; ISA: ARMv7E-M; flash read width: 128-bit (16 B) –…											
↪	AUTOWS 1, RWS 0, CACHEDIS 00, CMCC disabled	reset	47.769	AUTOWS – behaves as RWS 1-2 at 48 MHz	1.00	off (CEN = 0)	0x000022DC, off 12, shift 4	5 000	8/16	0/16[400k]	V / 0/N
↪	CEN 1 – CMCC on	fix	47.769	AUTOWS	2.00	on, 4 KB		flat	flat	3/16[400k]	sigma
↪	RWS 0, CMCC on	zero wait states	23.922	0	0.000	on, 4 KB		flat	flat	5/16[400k]	sigma
↪	CSIZESW = 1 KB, CMCC on	anti-fix	47.769	AUTOWS	2.00	on, 1 KB		2/16[400k]	40k	400k	V
↪	CEN 0 and CACHEDIS0	fix	47.769	AUTOWS	n/m	off		0/16[400k]	flat	flat	0/N
XMC1100											
vendor: Infineon; core: Cortex-M0; ISA: ARMv6-M; flash read width: not observable –…											
↪	1 WS, CONFIG1.FIXWS = 0	reset	32	1, adaptive	n/a	n/a		flat	flat	flat	0/N
MK21FN1M0											
vendor: NXP Kinetis; core: Cortex-M4F; ISA: ARMv7E-M; flash read width: 128-bit (16 B);…											
↪	all five FMC enables on	reset	21.1	derived from the clock ratio	B0RWSC cycles	on, 256 B		100k	11/16	25k	V
↪	B0SEBE only – latch isolated	partial disable	21.1	as above	n/m	off		2 000	0/16[100k]	flat	V

continued

TABLE 12, continued

part	config	config note	clock (meas.)	WS	miss - hit (cyc/access)	cache	table address	256 B byte S-box	1.25 KB compact T	4.25 KB 4-table	ev
↪	all five FMC enables OFF – was labelled "B0SEBE 0 with the cache also off", which read…	fix	21.1	as above	n/m	off		flat	flat	flat	0/N
↪	B0RWSC 0	fix	25	0	0.000	on		flat	flat	flat	0/N
ADuCM3029											
vendor: Analog Devices; core: Cortex-M3; ISA: ARMv7-M; flash read width: 8 B – read off…											
↪	any flash configuration	reset	26	n/a	n/m	n/a		20k	5 000	2 500	V
↪	tables in RAM	fix	26	n/a	n/a	n/a		flat	flat	flat	0/N
RP2040											
vendor: Raspberry Pi; core: dual Cortex-M0+; ISA: ARMv6-M; flash read width: external…											
↪	cache on	shipped	n/s	boot2-dependent	n/m	on, 16 KB		flat	flat	spreads	0/N
↪	cache off	control	n/s	boot2-dependent	n/m	off		flat	flat	n/s	0/N
RP2350											
vendor: Raspberry Pi; core: dual Cortex-M33; ISA: ARMv8-M Mainline; flash read width:…											
↪	cache on	shipped	150	CLKDIV-dependent	n/m	on, 16 KB		0/8[400k]	0/16[400k]	0/16[400k]	0/N
ARM classic (ARM7/ARM9)											
LPC2124 (Olimex LPC-H2124)											
vendor: NXP (Philips); core: ARM7TDMI-S; ISA: ARMv4T; flash read width: 128-bit (16 B),…											
↪	MAMCR 0, MAMTIM 7, PLL bypassed	reset	14.7456	7	n/a	off		flat	flat	flat	0/N
↪	MAMCR 2, MAMTIM 7	shipped	14.7456	7	6.0	latches on	0x00005D00, off 0, shift 4	100k	25k	5 000	V
↪	MAMTIM 1	fix	14.7456	1	0.0000	latches on		flat	flat	flat	0/N
↪	MAMCR 1 – NXP's documented data-side bypass	anti-fix	14.7456	7	every direct probe confirms the bypass is TOTAL	partial		8/16	0/16[100k]	8/16[100k]	P

continued

TABLE 12, continued

part	config	config note	clock (meas.)	WS	miss - hit (cyc/access)	cache	table address	256 B byte S-box	1.25 KB compact T	4.25 KB 4-table	ev
Power Architecture											
MPC5606B											
vendor: NXP; core: e200z0h; ISA: Power e200; flash read width: 128-bit (16 B);…											
↪	PFCR0 = 0x18C780ED – buffers on, depth 1, data prefetch off	reset	16.126	3	3.000	n/a – no core cache	n/s, off 8, shift 4	3 750	2 500	750	V
↪	PFCR0 = 0x18C180ED – RWSC 0	fix	16.126	0	0.000	n/a		flat	flat	flat	0/N
↪	BFE 0 – buffers disabled	fix	16.126	3	n/a	n/a		flat	flat	flat	0/N
↪	tables in the DFlash array	NOT a fix	16.126	3	n/m	one 128-bit holding register, no prefetch		7 500	2 000	750	V
Infineon TriCore											
TC234LP											
vendor: Infineon AURIX; core: TriCore TC1.6E; ISA: TriCore; flash read width: 256-bit (32…											
↪	DCON0.DCBYP = 1 – the CPU-side buffer BYPASSED	reset	101.7	WSPFLASH 5, WSECPF 1 (Infineon's values)	+355 cycles and +5.07 sd per WSPFLASH step, at the whole-encryption level	bypassed		16.38k	8 192	3 072	V
↪	DCBYP = 0 – buffer enabled	NOT a fix	101.7	as above	n/m	enabled		has NOT broken at 26.62k	12.29k	26.62k	sigma
↪	WSPFLASH 1, WSECPF 0 – the floor	the floor of the field	101.7	1	n/m	bypassed		16.38k	8 192	NOT broken	V
↪	tables in the DFlash port	fix	101.7	WSDFLASH / WSECDF	n/a	none		constant to the measurement floor —— RE-MEASURED 2026-10-21 at code offset 22 on image…	constant to the measurement floor	flat	0/N

continued

TABLE 12, continued

part	config	config note	clock (meas.)	WS	miss - hit (cyc/access)	cache	table address	256 B byte S-box	1.25 KB compact T	4.25 KB 4-table	ev
TI C28x											
TMS320F280049C											
vendor: TI; core: C28x; ISA: C28x; flash read width: 128-bit; row-holding structure: row…											
↪	DATA_ CACHE_EN = 1	shipped	100	5 (probe); lowest legal 3 at this clock	exactly RWAIT	on		5 000	n/a	n/a	V
↪	DATA_ CACHE_EN = 0	fix	100	5	flat	off		closed	n/a	n/a	screen
↪	RWAIT = 0 with the cache still on	fix	25	0	0	on		closed	n/a	n/a	screen
MIPS32 (PIC32)											
PIC32MZ2048EFH144											
vendor: Microchip; core: MIPS32 microAptiv; ISA: MIPS32; flash read width: 128-bit (16…											
↪	PRECON = 0x00000007 – 7 wait states, PREFEN 0, Config.K0 = 3	reset	200	7	n/m	on, 4 KB, covering SRAM too		flat	0/16[200k]	4/16[200k]	0/N
↪	Config.K0 = 2 – cache off for KSEG0	anti-fix	200	7	n/m	off	0x9D001250, off 0, shift 4	5 000	3 125	780	V
↪	PFMWS = 0	zero wait states	8	0	n/m	on, 4 KB		flat	0/16[200k]	4/16[200k]	sigma
↪	tables in RAM AND Config.K0 = 2	fix	200	7	n/a	off	0x9D001250, off 0, shift 4	flat	flat	flat	0/N
TI MSP430											
MSP430F5529											
vendor: TI; core: MSP430 CPUXv2 (16-bit); ISA: MSP430 CPUXv2; flash read width: not…											
↪	XT=branchless, FCTL4.MGR 0-3, tables in flash or RAM	reset	1	n/a	n/a	n/a		flat	flat	flat	0/N
RISC-V											
CH32V203G6U6											
vendor: WCH; core: QingKe V4B; ISA: RV32IMAC; flash read width: 4 B granule above…											

continued

TABLE 12, continued

part	config	config note	clock (meas.)	WS	miss - hit (cyc/access)	cache	table address	256 B byte S-box	1.25 KB compact T	4.25 KB 4-table	ev
↪	tables below 0x00008000 – the zero-wait area	fix	144	0 – and there is no field to set it in; the region is zero-wait by address	0.000	n/a – no cache and no latch exists in this region		flat	flat	flat	0/N
↪	tables above 0x00008000	shipped	144	n/a – no wait-state field exists on this part; the cost comes from the address	49.894	n/a – no cache; the structure is a 1-entry forward-only prefetcher	0x00008000, for all three implementations – every build here ends with a byte-S-box…	50k	12.5k	6 250	V

Result cells carry the verdict only; the qualifications, including any that reverse a verdict, are in `cross_part_table.json`; * recovered but NOT verified by k10 inversion, so the score is the scorer's and not a demonstrated key; *n/s* (σx) no recovery was run at this cell; x is the spread that was measured but probably not a real leakage; *flat* bit-exactly constant, one distinct cycle count; *n/s* not attacked at this cell; *n/a* no such control exists; *auto* hardware sets this automatically; *boot2* no wait-state register exists; the linked boot2 variant sets the miss cost; *CLKDIV* no wait-state register exists; a continuous QSPI timing divider sets it; *post-prog* result of a programmer init file, probably will be cleaned up.