Cache-Priming Countermeasure for Table-Based AES

Lorenzo Deflorian

DD2520 Applied Cryptography

Introduction

In this assignment I harden a software implementation of AES-128 against cache-based access-driven side-channel attacks. The starting point is a straightforward table-based AES in Rust, using the standard 256-byte S-box and inverse S-box. In this design, `SubBytes` and the key expansion perform secret-dependent table lookups, whose cache footprints can be observed by an attacker via techniques such as Prime+Probe or Flush+Reload on the last-level cache.

My chosen countermeasure is a *cache-priming strategy* for the AES S-boxes. Before each use of the S-box tables in encryption, decryption and key expansion, the implementation deliberately touches all entries of both the forward S-box and the inverse S-box in a fixed, key-independent pattern. The goal is that the attacker always observes that all cache lines containing S-box data are used, making it much harder to infer which particular S-box entries were accessed by the secret-dependent operations.

Implementation of the Countermeasure

The implementation lives in the Rust crate `aes-v2`. The AES state, finite-field arithmetic, S-boxes, AES-128 core and ECB helpers are implemented in this crate. In what follows I focus on the parts that implement the side-channel countermeasure.

Priming function

The heart of the countermeasure is the function `prime_sboxes` (defined in `aes-v2/src/lib.rs`, lines 221–242):

```
pub fn prime_sboxes() {
    let mut acc: u8 = 0;

    for &v in AES_SBOX.iter() {
        acc ^= v;
    }

    for &v in AES_INV_SBOX.iter() {
        acc ^= v;
    }

    let _ = std::hint::black_box(acc);
}
```

Key properties of this function:

- It scans *all* 256 entries of the forward S-box `AES_SBOX` and all 256 entries of the inverse S-box `AES_INV_SBOX`. The loop order is fixed and does not depend on the key, plaintext, ciphertext, or any other secret.
- It accumulates all values into a dummy variable `acc`, and passes it through `std::hint::black_box`. This prevents the Rust compiler (LLVM) from optimizing away the memory accesses as dead code, while still keeping the result independent of the AES computation.
- The function is marked `#[inline(never)]`, which makes it more likely that it remains a distinct block of instructions and is not merged in unexpected ways that could interfere with the intended cache behaviour.

Because the S-boxes are static arrays, this loop pattern reliably accesses all bytes of both tables. On a real machine, the CPU will fetch their cache lines into the data cache hierarchy during the execution of `prime_sboxes`.

Integration into AES-128

The AES-128 core is implemented by the `AES128` struct and its methods. The countermeasure is integrated at all points where the S-box tables are used:

- **Forward cipher (encryption):** In the `cipher` method (`aes-v2/src/lib.rs`, lines 257–275), I call `prime_sboxes()` before each of the 9 "full" rounds and again before the final round (calls at lines 263 and 271). Each of these rounds applies `sub_bytes`, which uses the forward S-box.
- **Inverse cipher (decryption):** In the `inv_cipher` method (`aes-v2/src/lib.rs`, lines 277–296), I call `prime_sboxes()` before each inverse round (before `inv_sub_bytes`) and once more before the final inverse SubBytes (calls at lines 284 and 293). These operations use the inverse S-box.
- **Key expansion:** In the `key_expansion` method (`aes-v2/src/lib.rs`, lines 405–450), I call `prime_sboxes()` before every invocation of `sub_word`, which is the part of the key schedule that applies the forward S-box to 4-byte words (call at line 416).

As a result, every time the AES algorithm is about to perform secret-dependent S-box lookups, it first performs a fixed, secret-independent scan over the entire S-box memory region. This ensures that, at the granularity of cache lines, the attacker sees that all S-box cache lines are "hot", regardless of which specific indices are later accessed by the actual AES round.

The functional behaviour of AES-128 is unchanged by this modification. I verified this by running the existing unit tests in `aes-v2`, which encrypt and then decrypt known test vectors and check that the output equals the input.

Security Reasoning

Threat model

I consider an attacker who:

- Can run code on the same physical machine and share the last-level cache (e.g., a co-resident process or virtual machine).
- Can perform an access-driven cache attack such as Prime+Probe or Flush+Reload on the S-box memory locations.

- Observes which cache sets or lines are used by the victim during AES encryption or decryption, and uses this information across many traces to reconstruct secret key bits.

The original table-based AES implementation without countermeasures leaks information through which S-box entries are used. In particular, the address `&AES_SBOX[state[(row,col)] as usize]` determines a cache line index that depends on the secret internal state of AES, which in turn depends on both the plaintext and the key. In a typical cache design, the 256-byte S-box spans multiple cache lines (for example, 4 lines of 64 bytes each), and the attacker needs only to distinguish which of these few lines were accessed in a given round to obtain significant information.

Effect of the priming countermeasure

With the priming countermeasure enabled, the high-level cache behaviour for the S-boxes in each round is as follows:

1. The victim calls `prime_sboxes()`, which scans *all* bytes of both S-box tables in a fixed order. At this point, every cache line that contains part of the S-box or inverse S-box is brought into the cache and marked as recently used.

2. Immediately afterwards, the victim executes SubBytes (or InvSubBytes or SubWord) using secret-dependent indices into `AES_SBOX` or `AES_INV_SBOX`. These accesses land in cache lines that are already present and recently used, so they do not substantially change the set of S-box cache lines that appear active.

3. The attacker probes the cache sets corresponding to the S-box region. Regardless of the key and plaintext, the attacker sees that *all* S-box cache lines have been accessed, because they were all primed by the countermeasure.

Thus, at the level of granularity of cache lines, the attacker learns almost nothing about which particular S-box entries have been used: every line looks used in every protected round. This destroys the main source of leakage that typical cache-based AES attacks exploit.

Note that the priming does *not* make the access pattern truly constant time at the finest granularity: the secret-dependent indices still impact which byte offsets within a cache line are accessed. However, practical access-driven attacks on modern CPUs generally do not have per-byte resolution but rather operate at the level of cache sets or lines. Within this standard model, the implemented countermeasure is effective.

Additional considerations

The countermeasure slightly increases the running time of AES because each round now performs additional memory accesses over the S-boxes. This is a trade-off: we spend extra time and bandwidth to reduce side-channel leakage. For this assignment, I accept the performance cost and focus on the qualitative security improvement.

Furthermore, the countermeasure is applied not only to encryption and decryption rounds but also to key expansion, because the key schedule also uses the S-box via `sub_word`. If the key expansion were left unprotected, an attacker observing cache accesses during the key schedule could still extract information about the secret key. By priming before each `sub_word`, I ensure that the same "all lines are hot" property holds for the key schedule as well.

Assumptions

My security reasoning relies on the following explicit assumptions about the runtime, compiler and hardware:

- **Memory layout of S-boxes:** I assume that `AES_SBOX` and `AES_INV_SBOX` are allocated as contiguous arrays of 256 bytes each in memory, in a single region that maps to a small number of cache lines (for example, four 64-byte lines per S-box). This is standard for static arrays in Rust and for the underlying ABI.
- **Cache line size and granularity:** I assume a typical cache line size (e.g., 64 bytes) and that the attacker can observe activity at the granularity of lines or sets, but *not* at the granularity of individual bytes within a line. This matches the standard Prime+Probe / Flush+Reload model in the literature.
- **Compiler does not remove priming accesses:** I assume that the Rust compiler and LLVM backend honour the semantics of `std::hint::black_box` and do not optimize away the loops over the S-boxes. The use of `black_box` is specifically intended to prevent such dead-code elimination.
- **No reordering across AES rounds that breaks priming:** I assume that the compiler does not reorder `prime_sboxes()` calls across AES rounds in a way that would separate them from the subsequent SubBytes/InvSubBytes/SubWord calls. Because `prime_sboxes()` has visible side effects at the level of *observable* behaviour (it calls `black_box`), the compiler is not allowed to arbitrarily move it.
- **Attacker focus on S-box region:** I assume the attacker focuses on cache sets corresponding to the S-box memory region. Other parts of AES (e.g., MixColumns implemented with arithmetic rather than tables) do not introduce new table-based cache channels in this implementation.
- **No use of hardware AES instructions:** I assume that the hardware AES instruction set extensions (AES-NI) are either not available or not used by this code. If they were used, the S-box table implementation would be replaced by a hardware implementation, which is outside the scope of this software-level countermeasure.

Under these assumptions, the cache-priming countermeasure makes the S-box access pattern for each protected AES operation effectively independent of the key and plaintext at the level of cache lines, which is the main source of leakage exploited by standard cache-based AES attacks.

Limitations and Alternatives

This countermeasure does not protect against all possible side channels. For example, it does not address timing variations due to other parts of the implementation, microarchitectural channels unrelated to the data cache, or attacks with higher spatial or temporal resolution than cache sets. It also does not prevent an attacker from distinguishing when *no* AES operation is running versus when it is.

Other standard countermeasures exist:

- Using hardware AES instructions (AES-NI) to avoid software S-box tables entirely.
- Allocating S-boxes in non-cacheable memory or locking them into special cache ways (not easily accessible from user-space Rust).
- Randomizing or masking S-box tables together with corresponding changes to the AES computation.
- Bit-sliced AES implementations with no table lookups.

For this assignment, I selected cache priming because it is implementable in safe, portable Rust without requiring special operating system support or inline assembly, and it directly targets the specific leak of table-based S-box cache accesses.

Conclusion

I implemented a cache-priming countermeasure for a table-based AES-128 implementation in Rust by scanning all entries of the S-box and inverse S-box before each use in encryption, decryption and key expansion. This makes the cache footprint of the S-box tables largely independent of the secret key and plaintext at the granularity relevant to access-driven cache attacks. Given the explicit assumptions about memory layout, compiler behaviour and hardware caching, this countermeasure is a reasonable and effective way to reduce cache-based side-channel leakage in this context.