Cache-Timing Attacks Still Threaten IoT Devices

Sofiane Takarabt, Alexander Schaub, Adrien Facon, Sylvain Guilley, Laurent Sauvage, Youssef Souissi, Yves Mathieu

Cache-Timing Attacks still threaten IoT devices

Sofiane Takarabt[1,2], Alexander Schaub[2], Adrien Facon[1,3], Sylvain Guilley[1,2,3], Laurent Sauvage[2,1], Youssef Souissi[1], and Yves Matthieu[2]

[1] Secure-IC S.A.S., 15 Rue Claude Chappe, Bât. B, 35 510 Cesson-Sévigné, France
[2] LTCI, Télécom ParisTech, Institut Polytechnique de Paris, 75 013 Paris, France
[3] École Normale Supérieure, département d'informatique, 75 005 Paris, France

Abstract. Deployed widely and embedding sensitive data, IoT devices depend on the reliability of cryptographic libraries to protect user information. However when implemented on real systems, cryptographic algorithms are vulnerable to side channel attacks based on their execution behavior, which can be revealed by measurements of physical quantities such as timing or power consumption. Some countermeasures can be implemented in order to prevent those attacks. However those countermeasures are generally designed at high level description, and when implemented, some residual leakage may persist. In this article we propose a methodology to assess the robustness of the MbedTLS library against timing and cache-timing attacks. This comprehensive study of side-channel security allows us to identify the most frequent weaknesses in software cryptographic code and how those might be fixed. This methodology checks the whole source code, from the top level routines to low level primitives, that are used for the final application. We recover hundreds of lines of code that leak sensitive information.

1 Introduction

Formerly known as PolarSSL, MbedTLS library provides many cryptographic implementations and primitives that can be easily used by developers to design or implement new applications for embedded systems. However side channel attacks are known to be an efficient way to break many of those applications. They exploit physical measurements like power consumption, electromagnetic emanation and even timing duration to recover the secret key. Using different techniques based on statistical tools, an attacker is able to extract a secret key using only one (or very few) measure(s) for non-unprotected implementations. The timing attack is the first side channel attack presented by P. Kocher [14] in order to recover the exponent bits of RSA. More perfected versions of timing attack are derived to break more secured implementations like Square-and-Multiply Always and Montgomery Ladder. Combined with power acquisition, a key can be extracted in less than one thousand traces [10].

Securing implementation against some attacks becomes more and more challenging for designers and developers. They should take care about all attacks that use timing properties, inputs dependency [15,6,16] and different others parameters. However some developers still use different (already) available library to implement their cryptographic applications. So, even if they design a (secure) software from the top level description (loop iterating over the scalar, like Montgomery Ladder, Atomic Multiply Always, etc.) based on some primitive functions, they usually do not check whether or not those primitives actually respect the constant-time coding constraints. In the following we present our results based on MbedTLS cryptographic library.

2 Previous works

2.1 Timing attacks and cache-timing attacks

Timing attacks exploit the timing variations induced by different inputs. For example, in asymmetric cryptography, operations like modular multiplication or division can cause timing variations in the execution time, which might be exploited to retrieve secret keys. We mention also the cache-based timing attack [18,19,20,7,11] that exploit difference between the access time of slow main memory, or RAM, and the much faster processor cache. If the value to be accessed depends on secret values, the number of cache hits and misses can be correlated to the secret values. This can eventually lead to a full recovery. This kind of attack was first presented by D. Bernstein in [3], where he targets the OpenSSL tabulated AES implementation that was supposed to be constant-time. This attack is a profiled template attack, which exploits the fact the looking up the same data twice is faster than looking up at two different addresses.

Considering the cache-timing attack, one can distinguish between two kinds of attacks:

- **Passive**: Only the inputs can be controlled and no additional process needs to be run on the targeted device. The timing variations are only caused by the cache miss and hit of the tabulated computation.
- **Active**: This attack needs an additional process, that is able to "erase" the cache contents, which forces the victim process to reload the data (or instructions) from the main memory (for instance, FLUSH+RELOAD [20] attack).

Many cache-timing attacks have been published in 2018. They all exploit timing differences in either in the control flow or the data access patterns.

See a reasoned presentation of cache-timing attacks (as of December 2019) in Fig. 1. We focus on such attacks in this paper.

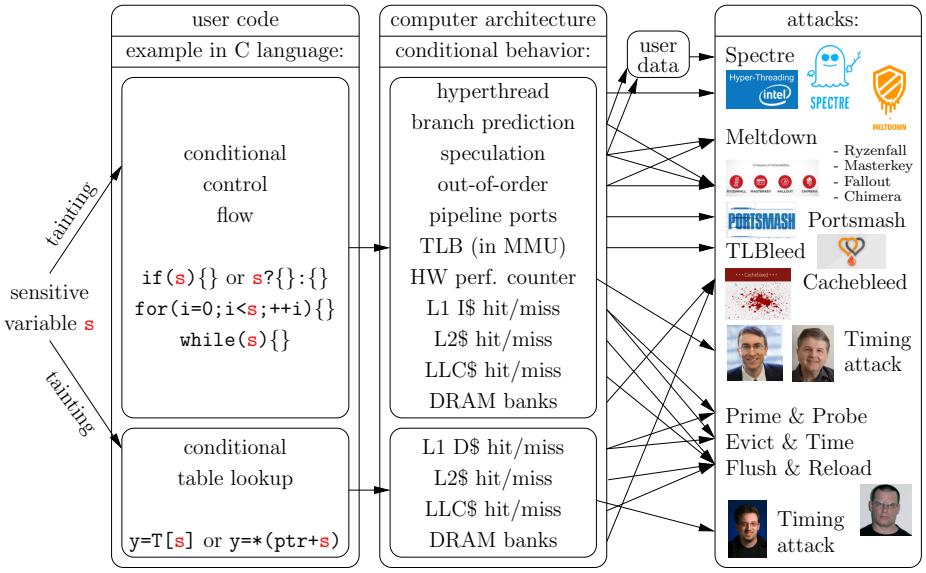

Fig. 1. Genealogy of cache-timing attacks

Algorithm 1: ECSM fast implementation: Double-and-Add—*Prone to cache-timing attacks*

// Input :
- P: base point
- $k = (k_0 \dots k_{n-1})_2$: scalar

Output: $[k]P$

$Q \leftarrow O$; O is the point at infinity
$i = n - 1$
while $i > -1$ **do**
  $Q \leftarrow [2]Q$; Point Doubling
  if $k_i = 1$ **then**
    $Q \leftarrow Q + P$; Point Addition
  end
  $i = i - 1$
end
return Q

In Algorithm 1 we give the high level description of a naive implementation of Elliptic Curve Scalar Multiplication (ECSM). The bits of k are scanned from left to right, and a conditional point addition is performed in order to deal with bit values at 1. If an attacker is able to probe the cache memory, he can recover the bits of k by measuring the access time to an intruction in the addition function which was initially flushed from the cache. If the access is long, then the victim code has not called the point addition. Otherwise, it has. When the ECSM is used to perform a signature generation or any other private operation, the secret key will be recovered.

Cache vulnerabilities are not only caused by conditional operations. As the size of the cache memory is limited to few thousand of bytes, the contents needs to be erased and reloaded from the main memory. A concrete example of such a situation is when the processor deals with relatively big tables. As mentioned in the section 2.1, the cause of the leakage is that the tables of AES cannot be fully loaded into the cache. Therefore, when different indexes of the array are accessed, data needs to be reloaded, which causes non-constant time execution.

Cache timing attacks can be prevented at various levels:

- **Source code**: by balancing the control flow, by fetching all values from a table, etc.
- **Compilation time**: by alignment constraints of tables to minimize cache effect (since cache lines are nowadays quite large); refer for instance to the following declaration of aligned table `T`: `unsigned char T[256] __attribute__((aligned));`
- **Assembly code**: with `cmov` and `setcc` constant-time operations.

2.2 Existing tools

Our goal is to identify the lines of code which can produce timing leakages when executed on a processor. Those leakages can be caused by a conditional operation, non-constant time operations and also cache access.

Some debugger tools were recently used for side channel analysis, like gdb and Valgrind [5,4]. Some of the identified vulnerabilities depend on the target device, but others are actually present in the source code. In the first case, the code should be fixed at low level (assembly instructions). In the second case, this can be done at high level source code (C).

3 Our Methodology

3.1 Leakage types

The cache vulnerabilities are exploited by time measurements. These variations are caused when the data requested by the processor is present or not in the cache memory. This data can be simple variables, arrays or even functions. The last two are the most vulnerable cases. In fact, if the array is relatively large, accessing different indexes will cause time variation. If those indexes can be controlled in some way, an attacker can correlate it with the sensitive data. In the case of functions, the same method can be applied, by guessing which functions have been executed, and correlating the observed time measurements with the sensitive data.

3.2 Principle of the tool

Evaluation of a source code against timing and cache-timing attacks should track the sensitive variables over all sub-routines, and check if any time variation can be caused by conditional branching or array access. The static analysis tool is based on four main steps:

1. **Input Preparation.** To analyze a source code, we need to specify the sensitive variable involved in computation.
2. **Dependency & Vulnerability.**
   (a) **Dependency Analysis.** The specified variable will be tracked over all sub-functions and if any dependency is detected, it will be logged.
   (b) **Vulnerability Identification.** All the dependency of the code with the sensitive data is built and analyzed. The instruction are filtered by keeping only some patterns that cause time variation (conditional branch), or array access (cache vulnerabilities).
3. **Vulnerability Analysis.** A post-processing is then applied on the reported leakages to classify them, remove the potential false positives (see Sec. 3.3) and produce a report readable to the designer.

The global work flow is illustrated in figure 2.

3.3 False positives

The tool can report false positives in those cases:

- The incriminated line of code is not called in the execution context

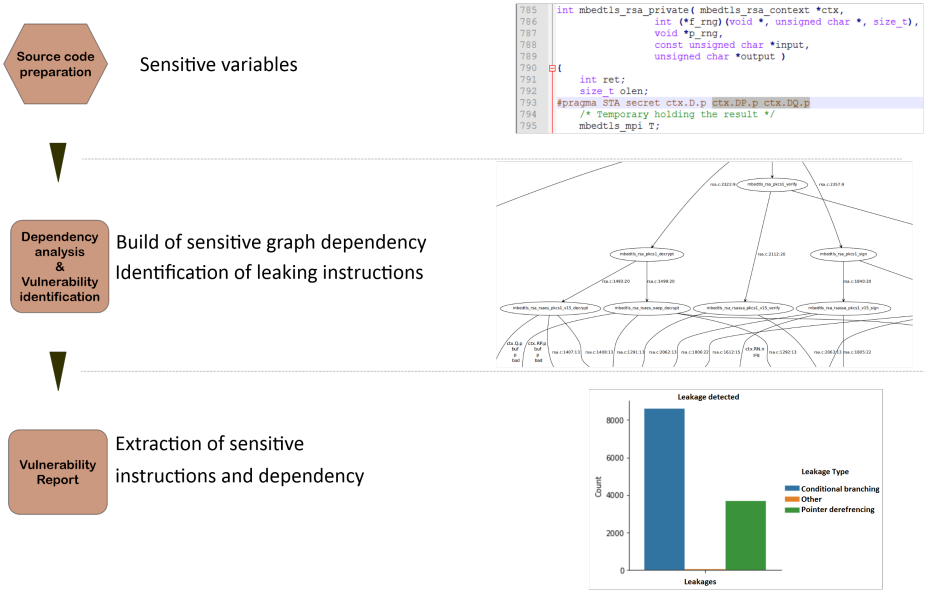

Fig. 2. Static analysis tool principle

- The vulnerability leaks too little:
  - either too little times;
  - or the leakage is too local, e.g., a table access where the table is so small it fits in a line of cache;
  - At the opposite, the leakages are too frequent in time, thereby making it challenging, if not impossible, to catch them all and/or to synchronize with them.
- Leakages sometimes happily disappear at compilation, e.g., when a structure such as `if(s){y=a;}`, which is compiled with a *conditional move* (`cmov`) instruction
- Leakages which occur in in exploitable places, e.g., in the middle of a hash function (which clearly cannot been related simply to the sensistive variable due to difficulty of preimage finding problem).

4 Evaluating MbedTLS source code

In this section we present our results on the main cryptography implementations: RSA, ECDSA, AES, DES and other block ciphers, from MbedTLS version 2.14.1. Our tool, named Catalyzr, analyzes the whole source and detects all the conditional branches and array accesses (called also pointer dereferencing) that depend on the specified (sensitive) parameters. The

full results with graph dependency and leakage location are given in appendix A.

4.1 Analysis of the RSA implementation

It is known that the naive implementation of RSA is vulnerable against side channel attacks [2,17,1]. Similar attacks also exist in the case of ECC [9]. MbedTLS implements countermeasures against some of these attacks, like Address-Bit DPA [12,13], timing and power analysis, etc. Indeed, those protections work pretty well, but some sensitive parts of the code can be exploited by an attacker in order to easily disturb the device, like in the case of cache-timing attack. As described in the figure 2, we take the self test function ($Mbedtls\_rsa\_self\_test$) in the library and tag the sensitive variable (namely the secret key) to track ($rsa.D$). All the conditional branches depending on the tagged variable are listed by the tool.

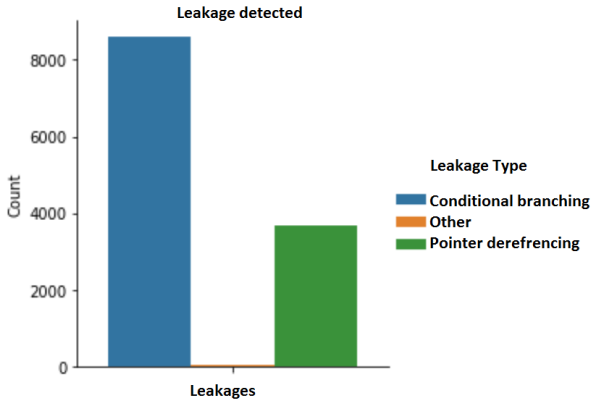

Fig. 3. Source of vulnerabilities detected on RSA signature

The designer should carefully analyze each result in order to determine if such warning is really exploitable in his use case. The figure 3 summarizes the source of such a dependency. In the case of private operation the (protected) *Mbedtls_mpi_exp_mod* function performs modular exponentiation based on sliding window, Montgomery Multiplication and Montgomery Reduction.

The Montgomery Reduction performs fake subtraction to prevent timing attacks (see figure 4), when the result is lesser than the modulus N.

```
1593        if( mbedtls_mpi_cmp_abs( A, N ) >= 0 )
1594            mpi_sub_hlp( n, N->p, A->p );
1595        else
1596            /* prevent timing attacks */
1597            mpi_sub_hlp( n, A->p, T->p );
```

Fig. 4. Extra reduction in the Montgomery Modular Multiplication-*bignum.c* file

```
965
966         while( c != 0 )
967         {
968             z = ( *d < c ); *d -= c;
969             c = z; d++;
970         }
```

Fig. 5. Carry leakage for ECDSA and RSA

The figure 5 shows the vulnerable line code in the *mpi_sub_help* function. This kind of vulnerabilities can be exploited by an attacker by chosen inputs, which may induce time variation induced by the carry propagation.

4.2 Analysis of ECDSA implementation

In the case of ECDSA signature, we have analyzed the *Mbedtls_ecdsa_sign* function. We tagged the scalar r used to sign the message. As in the previous section, we note that these leakages are found in the primitives that implement the arithmetic field operation using big integers.

The figure 6 shows a vulnerable code line that breaks out of the loop when the number of limbs has been determined. Such an optimization induces also a time variation (non-constant time implementation), which is susceptible to a timing attack.

As we can see, the variable n is then passed to the *mpi_sub_help* function which performs subtraction using only the n first limbs. The loop is therefore flagged as potential timing vulnerability, as shown in figure 7. This function is used in both RSA and ECDSA signatures, and this vulnerability is common for both. The more interesting leakage is the manner how we deal with the carry in the subtraction function.

In fact, we have tested the *Mbedtls_mpi_exp_mod* function with real data, to simulate actual algorithm execution times. The first identified leakage (figure 6) may not be significant as the size of inputs remains the

```
999         X->s = 1;
1000
1001         ret = 0;
1002
1003         for( n = B->n; n > 0; n-- )
1004             if( B->p[n - 1] != 0 )
1005                 break;
1006
1007         mpi_sub_hlp( n, B->p, X->p );
1008
```

Fig. 6. Vulnerable code location for ECDSA and RSA

```
955   static void mpi_sub_hlp( size_t n, mbedtls_mpi_uint *s, mbedtls_mpi_uint *d )
956   {
957       size_t i;
958       mbedtls_mpi_uint c, z;
959
960       for( i = c = 0; i < n; i++, s++, d++ )
961       {
962           z = ( *d < c );     *d -= c;
963           c = ( *d < *s ) + z; *d -= *s;
964       }
```

Fig. 7. Vulnerable code location and annotation for ECDSA and RSA: subtraction function

same, and no variation is observed. However the total time to perform the subtraction is different due to the second vulnerability (figure 5). The feasibility of a timing attack depends on the ability of an attacker to measure the time of a decryption with high accuracy, which is the case in most of embedded platforms.

4.3 Analysis of AES implementation

In the MbedTLS AES implementation, the SubBytes operation is performed using a table of 256 bytes (*Sbox*). If the size of the cache is large enough, the whole table Sbox can be fully loaded. Regarding the cache vulnerabilities, no time variation should occur in this case. However, in the case of active attack, an attacker can probe the cache contents, and gain knowledge about the lines and banks accessed during the Sbox computation.

The tool has also listed some vulnerabilities in the key-scheduling step (*mbedtls_aes_setkey_enc*), where the Sbox accesses depend directly on the key value.

```
578                 RK[4] = RK[0] ^ RCON[i] ^
579                 ( (uint32_t) FSb[ ( RK[3] >> 8 ) & 0xFF ]     ) ^
580                 ( (uint32_t) FSb[ ( RK[3] >> 16 ) & 0xFF ] << 8 ) ^
581                 ( (uint32_t) FSb[ ( RK[3] >> 24 ) & 0xFF ] << 16 ) ^
582                 ( (uint32_t) FSb[ ( RK[3]     ) & 0xFF ] << 24 );
```

Fig. 8. Vulnerable code location and annotation for AES

In figure 8, all the lines from 595 to 598 are listed as vulnerable. The exploitation of this vulnerability depends on the ability of an attacker to probe the cache content [18,20]. More leaking code was found in the encryption and decryption functions, *mbedtls_internal_aes_encrypt* and *mbedtls_internal_aes_decrypt* respectively, as the cache behaviour depends on a controllable parameter(plaintext or ciphertext).

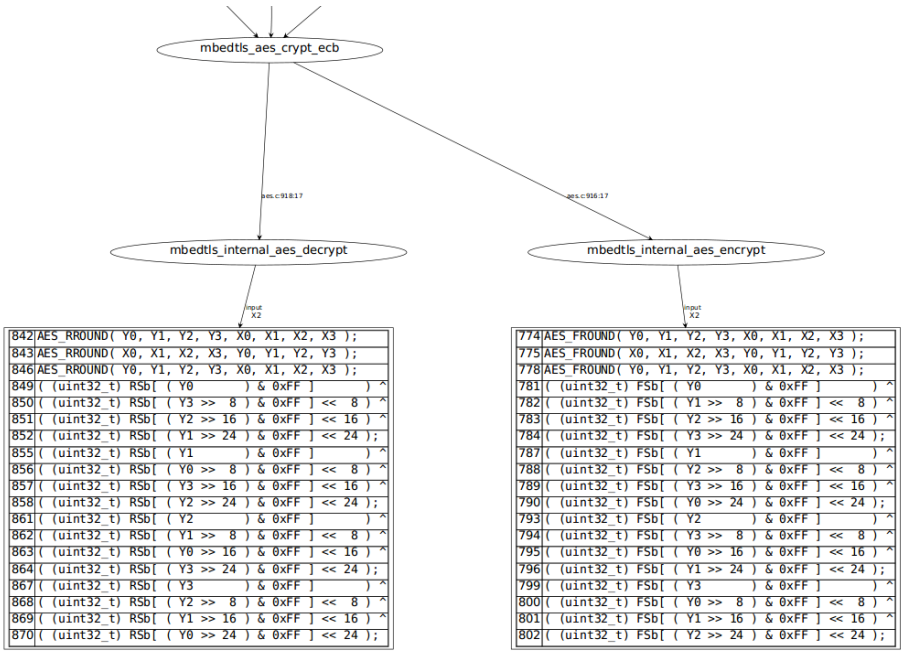

Fig. 9. AES leakage tracking and code location

The figure 9 shows the inter-procedural graph that gives all the leakages. The bubble-shape gives the function name, and the tables list the leaking code with the line number (first column). The corresponding file is shown in the arrow title. At the top figure, we have the *mbedtls_aes_crypt_ecb* function that calls either

- *mbedtls_internal_aes_encrypt* (line 918), or
- *mbedtls_internal_aes_decrypt* (at line 916)

in the *aes.c* file. The lowest tables show the leaking lines (first column), that depend on the input. In this case, the attack is less difficult, because we need to probe only one Sbox at each encryption, and by repeating at most 256 time the same (or wisely chosen) message, we can deduce the key value. If Sbox table cannot be fully loaded into cache memory, time variation can be observed. With chosen inputs, this variation can be controlled and leads to high correlation between those inputs and encryption time. This is equivalent to the attack described in [3].

4.4 Analysis of DES implementation

The MbedTLS DES implemention uses eight tables for SubBytes operation, each has 16×4 half-bytes. In order to analyze this algorithm we have tagged the *des3_test_keys* in the file *des.c*.

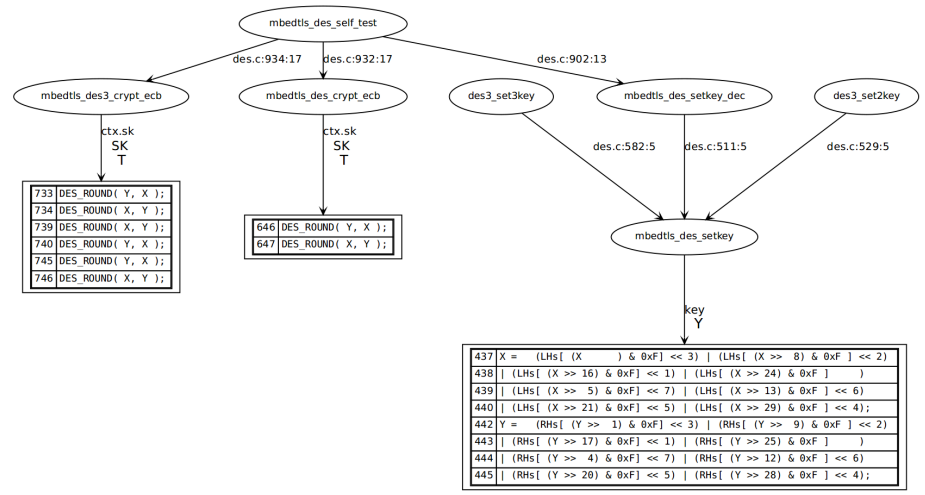

Fig. 10. DES graph with leakage dependency: *mbedtls_des_self_test* function

The figure 10 shows all the vulnerabilities listed by the tool. In the left side we have the two encryption functions based on simple DES and triple DES. In this case we have the DES_ROUND macro that perform the SubBytes operation, which depends on the secret data $ctx.sk$ and SK. In the right side the lines 437 to 440, and 442 to 444 show the vulnerable part in the key-schedule function. The first leakage is related to the encryption datapath and as explained in the previous section, it is less difficult to exploit. The second one is present only at the key schedule step, which make it very difficult to exploit.

4.5 Analysis of Blowfish implementation

Blowfish is a symmetric algorithm also based on Sbox-tables to perform the SubBytes operation. Four tables of 256 32-bits word are used to perform this operation. This leads to 4 KB of data. Loading the whole table in cache memory may not be possible on constrained devices. This function is called F, and it is called from the $blowfish\_enc$ function that encrypts a plaintext x using a secret key. We tagged the key ($ctx.P$) as a sensitive variable to track, and the tool has pointed four vulnerabilities in the F function.

```
89      y = ctx->S[0][a] + ctx->S[1][b];
90      y = y ^ ctx->S[2][c];
91      y = y + ctx->S[3][d];
```

Fig. 11. Vulnerable code location and annotation for Blowfish

The figure 11 shows the first leaking code line, which contains two array accesses. The tool has pointed two times the line 89, and one time line 90 and 89. The variables a, b, c, and d, depend on the input x of the function F, which are actually `xor`ed with the secret key in the $blowfish\_enc$ function. Regarding the cache-timing attacks, it is difficult to exploit such vulnerabilities. For example, to identify the value of a (b, c or d), the attacker need to probe cache line to see which word is loaded, and hence deduce the value of the key if the input (or the output) is known.

4.6 Analysis of Camellia implementation

For the Camellia algorithm, we have tagged variable $camellia\_test\_ecb\_key$ that is used as a secret key. Similar to the previously presented algorithm,

Camellia resorts to a table for the SubBytes operation. Four different tables of size 256 bytes are used for this purpose, which leads to 1 KB memory.

```
304       I0 = ((uint32_t) SBOX1((I0 >> 24) & 0xFF) << 24) |
305            ((uint32_t) SBOX2((I0 >> 16) & 0xFF) << 16) |
306            ((uint32_t) SBOX3((I0 >> 8) & 0xFF) << 8) |
307            ((uint32_t) SBOX4((I0     ) & 0xFF)     );
308       I1 = ((uint32_t) SBOX2((I1 >> 24) & 0xFF) << 24) |
309            ((uint32_t) SBOX3((I1 >> 16) & 0xFF) << 16) |
310            ((uint32_t) SBOX4((I1 >> 8) & 0xFF) << 8) |
311            ((uint32_t) SBOX1((I1     ) & 0xFF)     );
```

Fig. 12. Vulnerable code location and annotation for Camellia

The tool has underlined the array access in $camellia\_feistel$ function as shown in figure 12. Depending on the cache size, the time to access those tables may differs from one message to another. We note also that, for each byte position, a different Sbox table is used, which leads to high probability of cache miss events.

5 Discussion

In section 4.1, 4.2, we present some of our results that we hope will be taken into account in future release of the library. Those vulnerabilities are not necessarily known by developers, as they implement new applications based on already existing software for low-level primitives. We designed the tool in order to help such developers to check their implementations, as an end-to-end workflow integration check. We see that in most cases, the top level functions are well protected against the cited attacks. However, leakages are detected at the low level primitives, that are not updated in order to respect the specified constraints. We have seen that some of those leakages can be exploited by a simple timing analysis.

In the case of symmetric implementations, most leakages are related to array accesses. In fact, those vulnerabilities are target dependent. They should be analyzed by considering the cache specification. Since caches might be shared by different applications, cache vulnerabilities can arise even when the SBoxes might fit into cache, because less memory is available for each application. This can lead to the vulnerabilities mentioned in

sections 4.3, 4.5, 4.6. In all cases, we have identified all the array accesses that depend on the sensitive parameter. The most interesting parts are those depending also on the encryption datapath, like Sbox access, in the encryption functions. The vulnerabilities which depend only on the key are very difficult to exploit (case of key-schedule), because the attacker would need to repeat many times the cache probing to have only some information about the line accessed and then the value that was processed. Besides, in most of the optimized implementations this step is performed once, which makes the attack almost impractical.

In table 1 we summarize the details about the leakages reported by each

Table 1. Summary of leakage reported

Function	# Leakage	# Functions	# Lines
mbedtls_rsa_self_test	11131	40	147
mbedtls_ecdsa_sign	12588	34	124
mbedtls_aes_self_test	95	4	59
mbedtls_des_self_test	85	3	16
blowfish_enc	4	1	3
mbedtls_camellia_self_test	83	2	13

function (named in the first column). In the second column we give the total number of reported leakages. Those are all the possible paths through the control flow graph (this estimation is optimistic, since not all paths are exercised—more precisely, the paths can be taken, but are maybe not depending on the inputs). The third one shows the number of leaking functions (that induce the leakage). And the last one corresponds to the number of leaking code lines. This information can be deduced from the inter-procedural graph given in appendix A.

6 Conclusion and perspectives

In this paper we have presented some (automated) static analysis on MbedTLS library. The reported leakages are either related to a non-constant time implementation (as it was supposed to be), or to a potential cache vulnerability. In the first case, we have seen that a simple timing attack is possible, not only for the analyzed algorithms, but also for the future applications that will be based on the same routines. Exploiting the sensitive parts by a cache-active attack may be very difficult or impracticable in some cases. This depends on the ability of an attacker (in terms

of speed, regularity, etc.) to probe the cache. However in some conditions, equivalent time variation could occur and reveal information about the processed data. This is the case when the cache size is limited, or when the machine is so loaded that it is shared with other threads.

As a perspective, we intend to attribute each identified leakage to existing attacks, such as exploitation of "extra-reductions" in RSA/ECC Montgomery Modular Multiplication [10] or the exploitation of the correlation between the computation duration and the length of the nonce in ECDSA signature generation algorithm [8].

Acknowledgments

This work has benefited from a funding via TeamPlay (`https://teamplay-h2020.eu/`), a project from European Union's Horizon2020 research and innovation programme, under grand agreement N° 779882. Besides, this work has been partly financed by NSFC grant N° 61632020, and French PIA (*Projet d'Investissment d'Avenir*) grant N° P141580, of acronym RISQ (*Regroupement de l'Industrie pour la Sécurité post-Quantique*).

References

1. Cyril Arnaud and Pierre-Alain Fouque. Timing attack against protected RSA-CRT implementation used in PolarSSL. In *Cryptographers' Track at the RSA Conference*, pages 18–33. Springer, 2013.
2. Aurélie Bauer, Eliane Jaulmes, Victor Lomné, Emmanuel Prouff, and Thomas Roche. Side-channel attack against rsa key generation algorithms. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 223–241. Springer, 2014.
3. Daniel J Bernstein. Cache-timing attacks on AES. 2005.
4. Joppe W Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. Differential computation analysis: Hiding your white-box designs is not enough. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 215–236. Springer, 2016.
5. Antoine Bouvet, Nicolas Bruneau, Adrien Facon, Sylvain Guilley, and Damien Marion. Give me your binary, I'll tell you if it leaks. pages 1–4, 04 2018.
6. Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 16–29. Springer, 2004.
7. Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and reload — A cache attack on the BLISS lattice-based signature scheme. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 323–345. Springer, 2016.

8. Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks Are Still Practical. In Vijay Atluri and Claudia Díaz, editors, *ESORICS*, volume 6879 of *Lecture Notes in Computer Science*, pages 355–371. Springer, 2011.
9. Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 292–302. Springer, 1999.
10. Margaux Dugardin, Sylvain Guilley, Jean-Luc Danger, Zakaria Najm, and Olivier Rioul. Correlated Extra-Reductions Defeat Blinded Regular Exponentiation. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2016.
11. Adrien Facon, Sylvain Guilley, Matthieu Lec'hvien, Alexander Schaub, and Youssef Souissi. Detecting cache-timing vulnerabilities in post-quantum cryptography algorithms. In *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*, pages 7–12. IEEE, 2018.
12. Kouichi Itoh, Tetsuya Izu, and Masahiko Takenaka. Address-bit differential power analysis of cryptographic schemes ok-ecdh and ok-ecdsa. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 129–143. Springer, 2002.
13. Kouichi Itoh, Tetsuya Izu, and Masahiko Takenaka. A practical countermeasure against address-bit differential power analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 382–396. Springer, 2003.
14. Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.
15. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
16. Thanh-Ha Le, Cécile Canovas, and Jessy Clédiere. An overview of side channel analysis attacks. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 33–43. ACM, 2008.
17. Yuto Nakano, Youssef Souissi, Robert Nguyen, Laurent Sauvage, Jean-Luc Danger, Sylvain Guilley, Shinsaku Kiyomoto, and Yutaka Miyake. A pre-processing composition for secret key recovery on android smartphone. In *IFIP International Workshop on Information Security Theory and Practice*, pages 76–91. Springer, 2014.
18. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers Track at the RSA Conference*, pages 1–20. Springer, 2006.
19. Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.
20. Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732, 2014.

Appendix A

Here we give all the inter-procedural graphs that show the dependency and the leakage location for each algorithm.

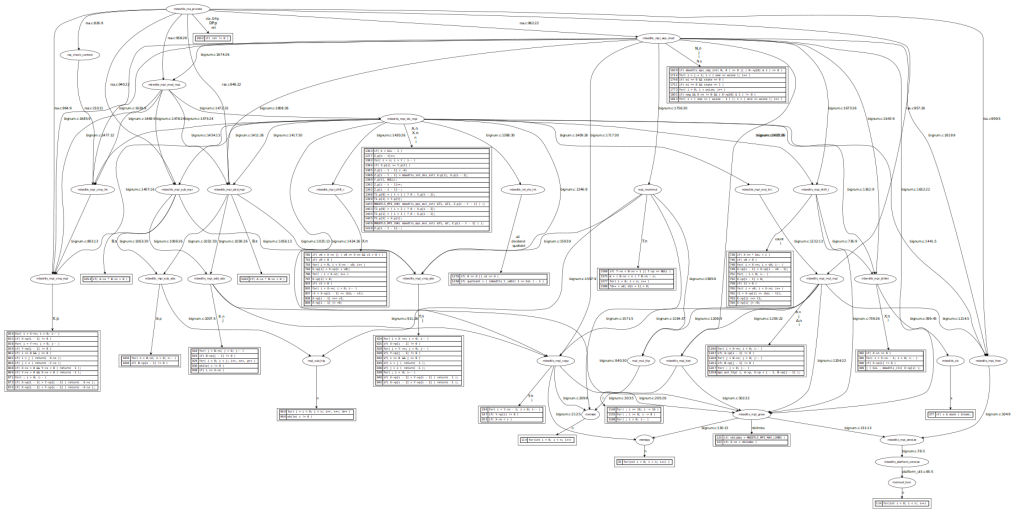

Fig. 13. Full RSA graph with leakage dependency for *mbedtls_rsa_private* function

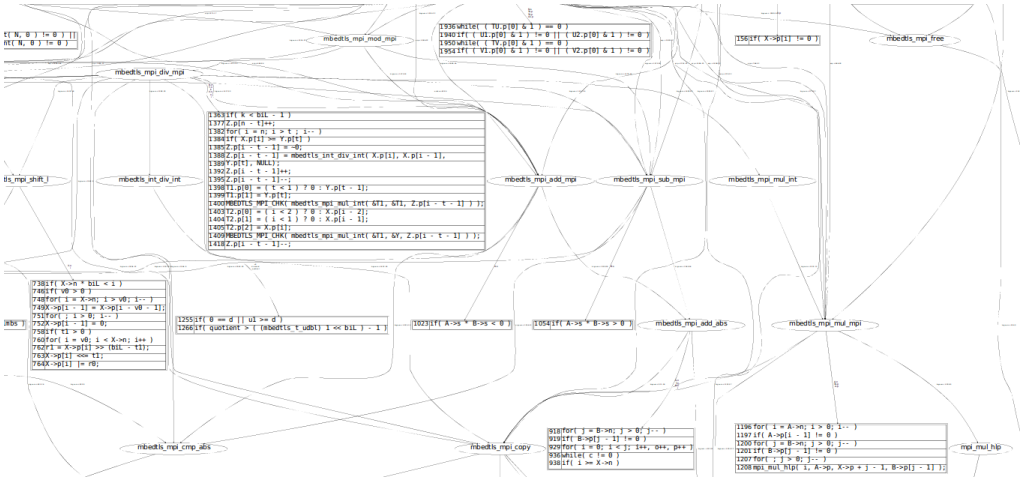

Fig. 14. Part of ECDSA graph with leakage dependency for *mbedtls_ecdsa_sign* function

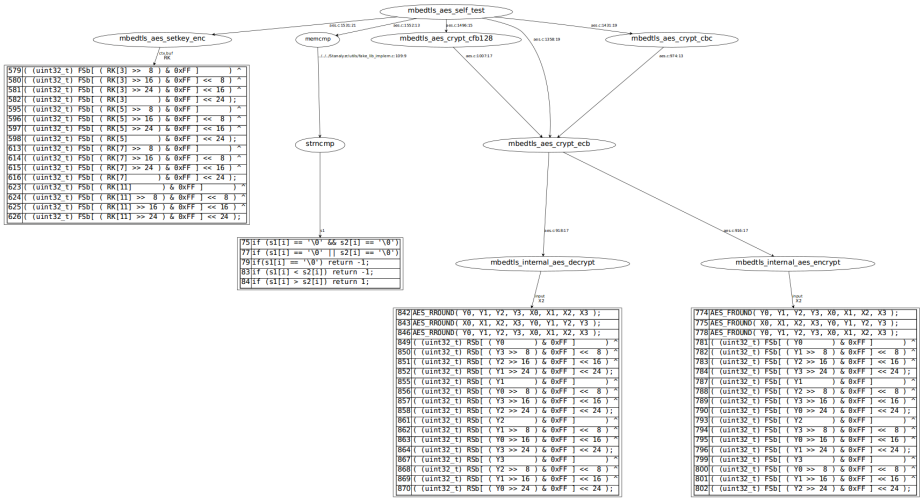

Fig. 15. Full AES graph with leakage dependency: *mbedtls_aes_self_test*

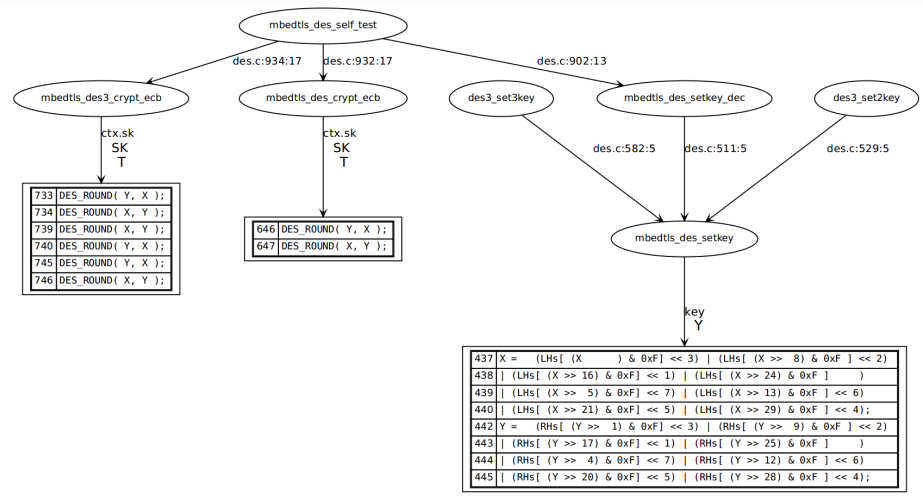

Fig. 16. Full DES graph with leakage dependency: *mbedtls_des_self_test* function

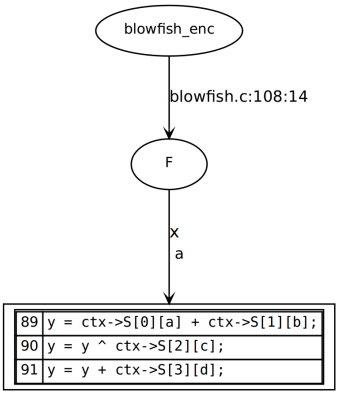

Fig. 17. Full Blowfish graph with leakage dependency: $blowfish\_enc$ function

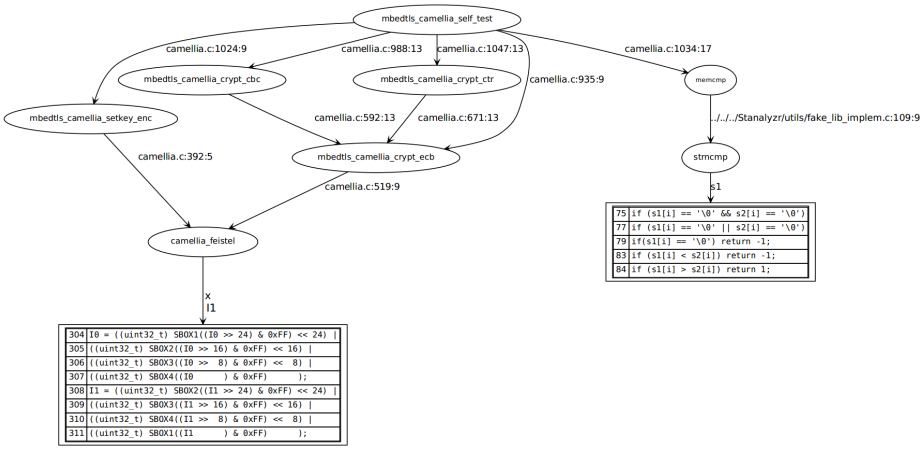

Fig. 18. Full Camellia graph with leakage dependency: $mbedtls\_camellia\_self\_test$ function