Cache-timing Attack Detection and Prevention Application to Crypto Libs and PQC

Sebastien Carre, Adrien Facon, Sylvain Guilley, Sofiane Takarabt, Alexander Schaub, Youssef Souissi

Cache-timing Attack Detection and Prevention
Application to Crypto Libs and PQC

Sébastien Carré[1,2], Adrien Facon[1,3], Sylvain Guilley[1,2,3],
Sofiane Takarabt[1,2], Alexander Schaub[2], and Youssef Souissi[1]

[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. With the publication of Spectre & Meltdown attacks, cache-timing exploitation techniques have received a wealth of attention recently. On the one hand, it is now well understood which some patterns in the C source code create observable unbalances in terms of timing. On the other hand, some practical cache-timing attacks (or Common Vulnerabilities and Exposures) have also been reported. However the exact relationship between vulnerabilities and exploitations is not enough studied as of today.

In this article, we put forward a methodology to characterize the leakage induced by a "non-constant-time" construct in the source code. This methodology allows us to recover known attacks and to warn about possible new ones, possibly devastating.

Keywords: Cache-timing attacks, leakage detection, leakage attribution, discovery of new attacks.

1 Introduction

Writing secure cryptographic software is notoriously hard, since mistakes can often be turned into an advantage by attackers to really extract the secrets. For instance, corruption of computations is known to allow for catastrophic failures, such as cryptographic algorithm breaks [17]. Consider for instance:

- the Bellcore [4] attack on RSA with Chinese Remainder Theorem (CRT-RSA),
- the differential fault analysis (DFA [3]) on AES (ISO/IEC 18033-3),
- verification skips in signature schemes (recall the case of the double `goto` inadvertent copy-and-paste [16]).

Any bug in the implementation (e.g., possibility to perform a buffer overflow) which allows for replacing an intermediate value (as for Bellcore and

DFA) does lead to a successful cryptanalysis. In the case of the verification skip in the signature schemes, the bug is already in the source code and allows the attacker to bypass the cryptography under some particular conditions.

Therefore, it is essential to write correct (bug-free) cryptographic software. Still this is not sufficient, since other attacks can still be applied. Typically, side-channel attacks are also known as particularly threatening. Indeed, they exploit some non-functional albeit observable side-effects caused by computations to profile the cryptographic code, and to come back to the secret non-invasively. The reason for these attacks to be feared is that, in most of the time, they cannot be detected.

One particular side-channel attack which received a great deal of attention are the so-called cache-timing attacks. Indeed, the observation is carried out directly by the machine which executes the victim code. Therefore, the resolution is high and the noise is low. Furthermore, it is not necessary for the attacker to possess the machine. The pre-condition for the attack is simply to be able to use a cryptographic service, just as the victim would.

By monitoring the time shared resources need to react, the attacker learns whether or not the victim has been soliciting those said resources. Shared resources are typically the multiple pipelines allowing for Hyper-Threaded computations, the use of cache memories for data, code, address translations (as in Memory Management Units or MMUs), the optimized management at the Dynamic Random Access Memory (DRAM) side, etc. Not all those resources are termed “caches”, but still the exploitation of the fact they can be contended by the concurrent usage request of a victim and an attacker have them leak observable information. This information is often measured as a timing variation, except for those situations where it is sufficient to directly measure the side-channel, e.g., in a hardware performance counter. When attacks consist in measuring a timing, the attacks are usually both *passive* and *active*, regarding non-functional resources: typically, a shared resource is set in a given state (e.g., a line of cache is flushed), and whether this state is modified by the attacker (e.g., the concerned line of cache is loaded by the victim) reveals a conditional behavior of the code. Sometimes, the attacks are refined in that some hardware peculiarity (e.g., branch prediction, out-of-order execution, etc.) enables indirectly the observable variability, correlated to some internal variable handled by the attacker. The operational use of cache-timing attacks is illustrated for instance to bypass kernel-level protections [14], to create covert-channels [21], to attack code in enclaves (CacheQuote [6]), etc. A big picture for so-called cache-timing vulnerabilities (at C code level) is depicted in Fig. 1.

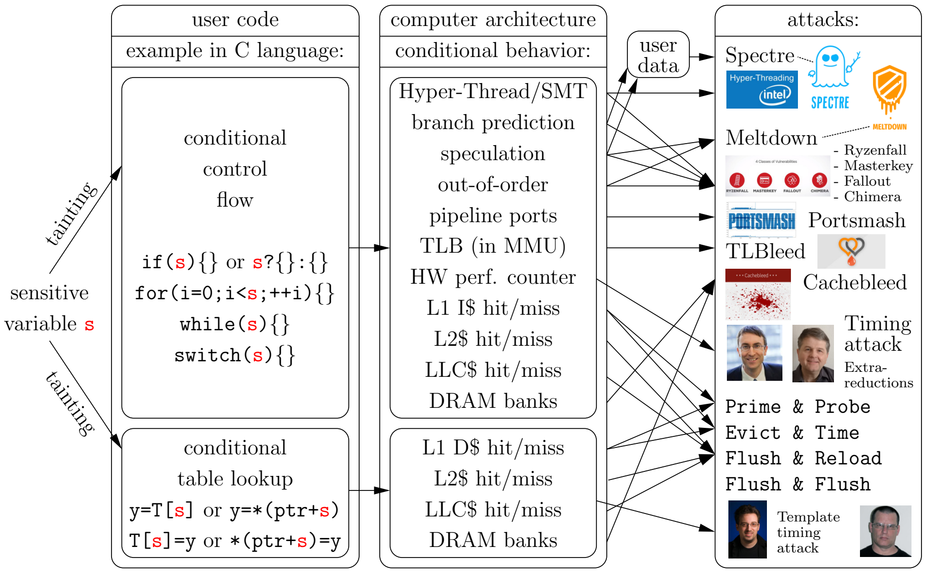

Fig. 1. Illustration how conditional code in one secret can manifest as observable side-channel leakage, and some renown exploitations

In this figure, attacks are related to the contended resources which leak. The survey paper [13] also details the relationship between micro-architecture and exploits. Nevertheless as of today, it is unclear how seriously a timing bias can be effectively exploited. This is precisely the intent of this paper.

The rest of this article is structured as follows. Known exploitation methods are presented in Sec. 2, and they are attributed to a purported hardware bias. Then comes our contribution in Sec. 3: we show there a methodology to assess the severity of a cache-timing leakage. Finally, the conclusion is given in Sec. 4. Some examples of codes are relegated to the appendix A.

2 Cache-timing issues

Issues related to cache-timing dependency on sensitive variables can lead to a variety of attacks, namely:

- On RSA:
  - Simple power analysis [19] (horizontal leakage)
  - Extra-reduction analysis [9]
  - BigMac Attack on windowed exponentiation [24]
- On ECDSA (attacks other than that directly transposable from RSA):
  - LLL cryptanalysis due to *observable* short nonces [5,12]
- On AES:
  - Timing attack [18]
  - Higher-order timing attacks [7]
  - Template attacks [23]

3 Cache-timing analysis methodology

3.1 State-of-the-art

Cryptographic libraries are thoroughly analyzed for vulnerabilities, and despite a lot of efforts devoted to this topic, libraries need more checking. Indeed, the application of protection can really affect strongly the performances. For instance, the use of sliding-window algorithm for exponentiation is known to leak but is believed hard to exploit. Still, using a perfectly regular exponentiation algorithms would collapse the performances. Hence the question whether or not the countermeasure is practically needed. This has pushed attackers to try harder, and actually a not so abstract attack on a key extraction has been put forward recently at CHES 2017 [2].

The Post-quantum cryptographic (PQC) algorithms have been analyzed for leakages. The affected parts contributing to leakages have already been classified systematically in [11, §V]:

- noise sampling operations, amongst them Gaussian noise is really sensitive,
- insecure Galois Field operations, especially in fields of characteristic two [8],
- variable time error correction algorithms,
- use of insecure large number libraries, such as GMP (GNU Multi-Precision, https://gmplib.org/).

Let us now explore a systematic leakage discovery methodology.

3.2 Methodology presentation

The presented methodology combines on the one hand *static* and *dynamic* analyses, and on the other hand *source* and *assembly* analyses.

Step 1 : Static analysis. The first step is a static analysis on source code. In this study, the tool is termed as Stanalyzr. Subsequently, the code is represented as an abstract syntax tree (AST), and the sensitive variables (secret keys, but also all parameters whose knowledge would allow to recover the secret keys, such as random numbers / noise involved in cryptographic protocols) are propagated in the tree. A vulnerability is the coincidence of a sensitive variable `s` and of either a conditional control flow operation (recall `if(s){}`, `for(i=0;i<s;++i){}`, `while(s){}`, and `switch(s){}` constructs illustrated in Fig. 1) or a conditional table lookup (recall `y=T[s]` and `y=*(ptr+s)`, or vice-versa, constructs illustrated in Fig. 1).

Let us illustrate in Fig. 2 the vulnerabilities found in RSA signature of MbedTLS. The listings 1.2 and 1.3 in appendix A show some practical leakages found automatically. The *extra-reduction* leakage illustrated in listing 1.4 is that which is analyzed in [1], and which can be exploited by cache-timing attacks even in advanced scenarios (e.g., regular exponentiation algorithms [9]). As illustrated in Fig. 2, the list of vulnerabilities can be regrouped according to their calling patterns. Indeed, for a versatile routine, there can be many functions actually requesting it. This is of great interest, since the more often a vulnerability is executed, the more likely it will leak exploitable information. Actually, one has to keep in mind that cache-timing attacks face a practical challenge, as:

- when applied against asymmetrical cryptography, which is typically randomized, the attack must succeed in one single trace;
- when applied against symmetrical cryptography (refer for instance to [23]), the key is unchanged for multiple operations, but the algorithms are very fast (around thousand clock periods, where the attackers aim at extracting hundreds of bits).

Step2 : Assembly code analysis. The second step consists in analyzing the generated assembly code after compilation of the C source code. The purpose if to check whether the vulnerability is still present. In some cases, the compiler manages to remove (unintentionally though) the problem upon assembly code generation. Table 1 illustrates typical translation of C structures into assembly.

It can be seen in Tab. 1 that some conditional operations can be translated in constant-time assembly instructions, such as `cmov` (conditional move, atomic) or such as `setcc` (set conditional, atomic). Indeed, these translations benefit the execution speed: as they do not break the

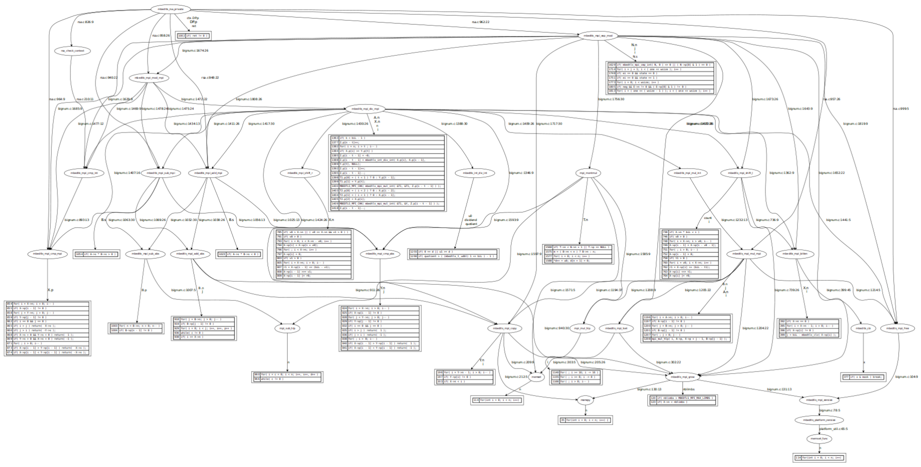

Fig. 2. Vulnerabilities identified in MbedTLS source code for RSA signature (courtesy of [22])

Table 1. Translation of cache-timing vulnerable C operations into assembly

C construct	**Pseudo-assembly construct**	**Vulnerable?**
`if(s){}`	`cmov s` or `setcc s`	no
`if(s){}`, `for(i=0;i<s;++i){}`, `while(s){}`, and `switch(s){}`	`test s`, `jump` *`address`*	yes
`y=T[s]` or `y=*(ptr+s)`	`load s`	yes
`T[s]=y` or `*(ptr+s)=y`	`store s`	yes

control flow, they can be executed without risking a cache or a speculation fault, thereby accelerating the execution. Furthermore, those translations happen only (paradoxically enough) when the code is compiled with optimization options.

The access to tables are almost certainly not fixed, since the techniques to make unconditional table accesses (bitslicing, extrapolation of table using Lagrange polynomial, lookup of all elements and subsequent addition of values after Boolean mask by the address indicator, etc.) are way too evolved. Additionally, the known tactics to protect table lookups feature extremely high timing overhead, hence shall be added manually. The vulnerability due to pointer dereferencing (except for tables with very small, e.g., two, number of entries) thus remains from C to assembly. For further reference on vulnerabilities at assembly-level, we redirect the reader to [20].

Step 3 : Statistical analysis. Finally, the code is executed dynamically, and `breakpoints` are set on the assembly lines previously identified as vulnerable. The information to be extracted is as follows:

- Count the number of occurrences while running the code — as mentioned, the more often the leak is executed, the more chance it is exploitable;
- Check for execution patterns. If they are too fast (e.g., as bursts), it might be hard to measure them individually.

The temporal distribution of vulnerabilities identified on mbedTLS thanks to static analysis is represented in Fig. 3 (obtained with Intel PIN, and presented in increasing lines of code), for the 800 first instructions (out of 3,679,883 making up a complete RSA 2048-bit).

It can be noticed that many vulnerable lines of codes are actually stepped several times. More precisely, some patterns are clearly visible, which highlight loop operations within functions. Therefore, statistical analysis greatly helps understand which line of code is particularly leaking.

Step 4 : Real-world exploitation. Ideally, this method is complemented by a real world measurements (e.g., using FLUSH+FLUSH [15] methodology, as that from the Catalyzr$^{\text{TM}}$ tool [10]), so as to assess in which respect the leakages are exploitable. Actually, regarding lookup tables, some accesses are indistinguishable, since they occur in the same line of cache. The final check allows to validate whether the risk is real.

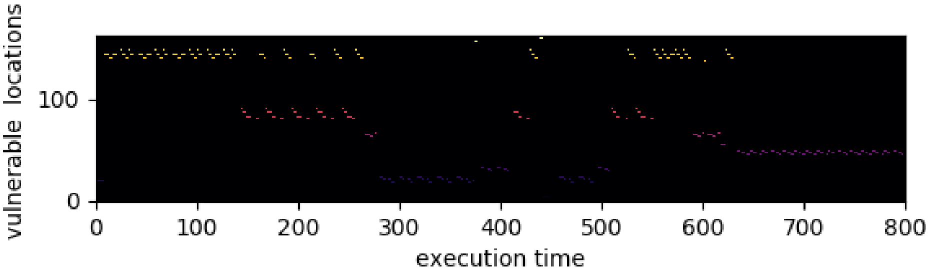

Fig. 3. Activation times (labelled in number of instructions) of vulnerabilities found by static analysis, for RSA signature.

3.3 Methodology application

The latest version of MbedTLS library (version `2.14.0`) at the time of writing this paper is studied. It is written with security in mind. Indeed, as a example, it features some functions allowing for conditional operations to be carried in a way which cannot be exploited by cache-attacks. An illustration is provided by function `mbedtls_mpi_safe_cond_assign` (where `mpi` stands for `multi`precision `inter`ger), located in `library/bignum.c` and given for reference as Listing 1.1 in Appendix A.

4 Conclusion

This paper has introduced a practical methodology to analyze observable cache-timing biases with respect to their possible exploitation. The methodology consists in several steps, namely: vulnerability identification in source code, vulnerability tracking in assembly code, statistics on the dynamic occurrence of the vulnerability, and eventually, real measurements using FLUSH+FLUSH methodology.

We have shown how known attacks are recovered in a software cryptographic library, and we point towards numerous new (uncovered yet albeit possibly devastating) ones.

Acknowledgments

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

References

1. Cyril Arnaud and Pierre-Alain Fouque. Timing attack against protected RSA-CRT implementation used in polarssl. In Ed Dawson, editor, *Topics in Cryptology - CT-RSA 2013 - The Cryptographers' Track at the RSA Conference 2013, San Francisco,CA, USA, February 25-March 1, 2013. Proceedings*, volume 7779 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2013.
2. Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding Right into Disaster: Left-to-Right Sliding Windows Leak. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 555–576. Springer, 2017.
3. Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997.
4. Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In Walter Fumy, editor, *EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.
5. 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.
6. Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):171–191, 2018.
7. Jean-Luc Danger, Nicolas Debande, Sylvain Guilley, and Youssef Souissi. High-order Timing Attacks. In *Proceedings of the First Workshop on Cryptography and Security in Computing Systems*, CS2 '14, pages 7–12, New York, NY, USA, 2014. ACM.
8. Jean-Luc Danger, Youssef El Housni, Adrien Facon, Cheikh T. Gueye, Sylvain Guilley, Sylvie Herbel, Ousmane Ndiaye, Edoardo Persichetti, and Alexander Schaub. On the Performance and Security of Multiplication in $GF(2^N)$. *Cryptography*, 2(3):25, 2018.
9. 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.

10. Adrien Facon, Sylvain Guilley, Matthieu Lec'Hvien, Damien Marion, and Thomas Perianin. Binary Data Analysis for Source Code Leakage Assessment. In Jean-Louis Lanet and Cristian Toma, editors, *Innovative Security Solutions for Information Technology and Communications - 11th International Conference, SecITC 2018, Bucharest, Romania, November 8-9, 2018, Revised Selected Papers*, volume 11359 of *Lecture Notes in Computer Science*, pages 391–409. Springer, 2018.
11. Adrien Facon, Sylvain Guilley, Matthieu Lec'hvien, Alexander Schaub, and Youssef Souissi. Detecting Cache-Timing Vulnerabilities in Post-Quantum Cryptography Algorithms. In *3rd IEEE International Verification and Security Workshop, IVSW 2018, Costa Brava, Spain, July 2-4, 2018*, pages 7–12. IEEE, 2018.
12. Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "make sure DSA signing exponentiations really are constant-time". In Weippl et al. [25], pages 1639–1650.
13. Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering*, 8(1):1–27, 2018.
14. Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In Weippl et al. [25], pages 368–379.
15. Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, volume 9721 of *Lecture Notes in Computer Science*, pages 279–299. Springer, 2016.
16. iOS 7.0.6. CVE-ID CVE-2014-1266. Description: Secure Transport failed to validate the authenticity of the connection. This issue was addressed by restoring missing validation steps. Impact: An attacker with a privileged network position may capture or modify data in sessions protected by SSL/TLS, February 2014. https://nvd.nist.gov/vuln/detail/CVE-2014-1266.
17. Marc Joye and Michael Tunstall, editors. *Fault Analysis in Cryptography*. Information Security and Cryptography. Springer, 2012. ISBN: 978-3-642-29655-0; DOI: 10.1007/978-3-642-29656-7.
18. Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.
19. 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.
20. Clémentine Maurice and Moritz Lipp. What could possibly go wrong with <insert x86 instruction here>?, December 2016. 33rd Chaos Communication Congress (33c3), Hamburg, Germany. https://lab.dsst.io/slides/33c3/slides/8044.pdf.
21. Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

22. Sofiane Takarabt, Alexander Schaub, Adrien Facon, Sylvain Guilley, Laurent Sauvage, Youssef Souissi, and Yves Matthieu. Cache-Timing Attacks still threaten IoT devices. In *Codes, Cryptology and Information Security - Third International Conference, C2SI 2019, Rabat, Morocco, April 22-24, 2019, Proceedings*. Springer, 2019.
23. Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology*, 23(1):37–71, 2010.
24. Colin D. Walter. Sliding Windows Succumbs to Big Mac Attack. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *CHES*, volume 2162 of *Lecture Notes in Computer Science*, pages 286–299. Springer, 2001.
25. Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 2016.

A Some excerpts from *secure* and *vulnerable* functions from mbedTLS

```
/*
 * Conditionally assign X = Y, without leaking information
 * about whether the assignment was made or not.
 * (Leaking information about the respective sizes of X and Y is ok however.)
 */
int mbedtls_mpi_safe_cond_assign( mbedtls_mpi *X, const mbedtls_mpi *Y, unsigned char assign )
{
    int ret = 0;
    size_t i;

    /* make sure assign is 0 or 1 in a time-constant manner */
    assign = (assign | (unsigned char)-assign) >> 7;

    MBEDTLS_MPI_CHK( mbedtls_mpi_grow( X, Y->n ) );

    X->s = X->s * ( 1 - assign ) + Y->s * assign;

    for( i = 0; i < Y->n; i++ )
        X->p[i] = X->p[i] * ( 1 - assign ) + Y->p[i] * assign;

    for( ; i < X->n; i++ )
        X->p[i] *= ( 1 - assign );

cleanup:
    return( ret );
}
```

Listing 1.1. Conditional assignment function, which does not reveal whether the assignment has been completed or not

```
/*
 * Initialize one MPI
 */
void mbedtls_mpi_init( mbedtls_mpi *X )
{
    if( X == NULL )
        return;

    X->s = 1;
    X->n = 0;
    X->p = NULL;
}
```

Listing 1.2. Example of vulnerable data-management code, as identified statically (the leakage is in the `if` statement)

```
/*
 * Signed addition: X = A + B
 */
int mbedtls_mpi_add_mpi( mbedtls_mpi *X, const mbedtls_mpi *A, const mbedtls_mpi *B )
{
    int ret, s = A->s;

    if( A->s * B->s < 0 )
    {
        if( mbedtls_mpi_cmp_abs( A, B ) >= 0 )
        {
            MBEDTLS_MPI_CHK( mbedtls_mpi_sub_abs( X, A, B ) );
            X->s =  s;
        }
        else
        {
            MBEDTLS_MPI_CHK( mbedtls_mpi_sub_abs( X, B, A ) );
            X->s = -s;
        }
    }
    else
    {
        MBEDTLS_MPI_CHK( mbedtls_mpi_add_abs( X, A, B ) );
        X->s = s;
    }

cleanup:

    return( ret );
}
```

Listing 1.3. Example of vulnerable arithmetic code, as identified statically (the leakages are in the `if` statement)

```
/*
 * Montgomery multiplication: A = A * B * R^-1 mod N  (HAC 14.36)
 */
static int mpi_montmul( mbedtls_mpi *A, const mbedtls_mpi *B, const mbedtls_mpi *N, mbedtls_mpi_uint mm,
                         const mbedtls_mpi *T )
{
    size_t i, n, m;
    mbedtls_mpi_uint u0, u1, *d;

    if( T->n < N->n + 1 || T->p == NULL )
        return( MBEDTLS_ERR_MPI_BAD_INPUT_DATA );

    memset( T->p, 0, T->n * ciL );

    d = T->p;
    n = N->n;
    m = ( B->n < n ) ? B->n : n;

    for( i = 0; i < n; i++ )
    {
        /*
         * T = (T + u0*B + u1*N) / 2^biL
         */
        u0 = A->p[i];
        u1 = ( d[0] + u0 * B->p[0] ) * mm;

        mpi_mul_hlp( m, B->p, d, u0 );
        mpi_mul_hlp( n, N->p, d, u1 );

        *d++ = u0; d[n + 1] = 0;
    }

    memcpy( A->p, d, ( n + 1 ) * ciL );

    if( mbedtls_mpi_cmp_abs( A, N ) >= 0 )
        mpi_sub_hlp( n, N->p, A->p );
    else
        /* prevent timing attacks */
        mpi_sub_hlp( n, A->p, T->p );

    return( 0 );
}
```

Listing 1.4. Example of vulnerable arithmetic code, as identified statically (the leakage is in the `mbedtls_mpi_cmp_abs` statement—and holds, irrespective of the `/* prevent timing attacks */` (incorrect) indication)