Benchmarking tools for verification of constant-time execution

Arthur Costa Lopes[1], **Diego de Freitas Aranha**[1]

[1]Institute of Computing – University of Campinas (UNICAMP)

Abstract. *Developing secure implementations of cryptography, in particular those protected against side-channel attacks, is a challenging research and engineering problem. In the case of timing attacks, the task can be facilitated by employing verification tools to check constant-time behavior. Given this concern, this work explores different verification tools and their distinct forms of analysis with application to verifying cryptographic libraries. A simple syntax for describing implementations was designed and a benchmarking database was constructed to validate such tools, indicating that the combination of static and dynamic analysis is required to fully verify the timing behavior of a cryptographic implementation.*

Resumo. *Desenvolver implementações seguras de criptografia, em particular protegidas contra ataques de canal lateral, é um problema desafiador de pesquisa e engenharia. No caso de ataques de temporização, a tarefa pode ser facilitada ao se empregar ferramentas de verificação para determinar execução em tempo constante. Dado esse problema, este trabalho expora diferentes ferramentas de verificação e suas formas distintas de análise, com aplicação em bibliotecas criptográficas. Uma sintaxe simples para descrever as implementações foi projetada e uma base de implementações foi construída para comparar e validar tais ferramentas, indicando que uma combinação de análise dinâmica e estática é requisito para verificar completamente o comportamento de uma implementação criptográfica do ponto de vista de tempo de execução.*

1. Introduction

With a high demand for computational systems to store, transfer and process sensitive data, the Information Security community is responsible for designing and deploying methods that must remain secure even when widely used. The application of cryptographic techniques is a common approach to protect part of the attack surface of modern systems, but naturally even cryptography may suffer from flaws and vulnerabilities. Given this goal, there is a substantial amount of research dedicated to explore different ways to attack such systems, retrieve sensitive data and expose cryptographic keys.

Side-channel attacks became popular after their success in retrieving secret information protected by a cryptosystem, using methods that attack implementations of algorithms instead of tackling the more challenging underlying hardness assumptions such as integer factoring, discrete logarithm computation or exhaustive search in the key space [Kocher 1996]. There are many types of side-channel attacks, involving execution time, branch prediction rate, power consumption and acoustic emanations. The attacks can be passive and rely on monitoring of side-channel information only, or more invasive

and require injection of faults in the processing hardware. Passive attacks that work remotely against connected devices are among the most convenient for an attacker to mount.

In the past years, many timing attacks against cryptographic implementations came up in the literature, compromising entire cryptosystems and exposing sensitive data by using timing characteristics leaked by an insecure implementation [Bernstein 2004, Percival 2005, Aciiçmez et al. 2007]. Those attacks consist in measuring the execution/response time of an insecure implementation, which will be different for distinct inputs and may be correlated with secret information. Critical variances in execution time can be caused by careless programming practices, but also by more subtle cache-timing behavior, branch prediction/resolution effects and variable-time instructions within the processor microarchitecture. If a system is secure against timing attacks, it is called *constant time* because it produces the answer with the same latency independently of the inputs. On the other hand, if the running time is not constant, meaning that different inputs yield different running times, this system is called time-variant and may pose a vulnerability that can be exploited by a remote attacker.

Timing attacks can be extremely complex to exploit and require the combination of sophisticated techniques and observation of network traffic, as discussed above, but they can also be very simple. For example, suppose an algorithm that checks if a given string matches a certain secret password or its hash value, as shown below in Listing 1. It can be easily implemented by comparing each character from both strings and stopping the comparison immediately if a difference is found, or returning that the password is correct contrariwise. Now, given this specific implementation, consider an attack in which the opponent sends a random word, but with the first letter being *s* and measures the response time. If the first letter of the password is in fact *s*, then the execution time would be greater than the case in which the first letter was wrong, since the code will try to match the second letter. Now the attacker has a simple way to check each letter separately, reducing the complexity of the brute force attack from exponential to linear.

```
1  /*If pw == in returns 1, else returns 0 */
2  int compVar (char *pw, char *in, int len) {
3     for (int i = 0; i < len; i++) {
4        if (pw[i] != in[i]) {
5           return 0;
6        }
7     }
8     return 1;
9  }
10
11 int main(void) {
12    int result, comp=1;
13    char pw[] = "secret", in[7];
14    scanf ("%s", in);
15    result = compVar(pw, in);
16    printf ("%d", result);
17 }
```

Listing 1. Example of a time variant implementation of string comparison.

A constant-time version of the same string comparison function with the same interface can be found below in Listing 2.

```
1 /*If pw == in returns 1, else returns 0 */
2 int compConst (char *pw, char *in, int len) {
3     int r = 0;
4     for (int i = 0; i < len; i++) {
5         r |= (pw[i] ^ in[i]);
6     }
7     return (r == 0);
8 }
```

Listing 2. Example of a constant time implementation of string comparison.

Although constant time implementations appear to solve all the timing attack vulnerabilities, there are a few more steps that need to be taken in order to assure such protection. First the programmer must check if the high-level code executes in constant time across all code paths, possibly with the help of static analysis. Even though static analysis tools may check the code and rate it as secure, the programmer still must use other tools to verify dynamic behavior. The usage of both methods is required because in the transition from source code (i.e. C/C++ source file) to a final executable there are many processing steps performed by the compiler, which can generate a final version with side-channel protection disabled during the optimization or containing insecure variable-time instructions inserted during code generation.

This work presents two main contributions. The first one is CTBench, a large database containing several implementations of cryptographic functions, ready to be tested by any static or dynamic analysis tools. We evaluated the tools FlowTracker, *dudect* and *ctgrind* in terms of accuracy when analyzing if the implementations are constant time or time-variant. Another contribution was the design of a XML file format useful to annotate critical information in cryptographic code for static analysis.

This paper is organized as follows. Section 2 discusses the two analysis techniques, as well as a few examples of tools that are available in the research literature. Section 3 presents the representation format and how the benchmarking database is organized. Section 4 discusses the results obtained by the tools. Section 5 concludes the paper and discusses future work.

2. Related work

Previous attacks against implementations of AES were able to recover the full key almost in real time just by observing encryption of a few KBs of data [Gullasch et al. 2011], or by analyzing network response time [Bernstein 2004]. Others were able to recover Diffie-Hellman exponents in naive implementations of the square-and-multiply exponentiation operation [Kocher 1996] and even factor RSA keys [Schindler 2000] by measuring precisely the time required to perform some operations like encryption and multiplications, sometimes remotely [Brumley and Boneh 2005].

A more specific class of timing attacks is called cache-based attacks, which uses leakage from hits and misses in cache memory to extract information about a secret parameter (i.e. cryptographic keys or plaintext data). Cache memory provides faster

memory access to the processor for small amounts of data which are frequently used, reducing the delay to retrieve these values from the main memory. Since cache memory typically only has up to 128 KBs of capacity in its fastest L1 level, sometimes a process may not find a memory address loaded on it, and therefore the processor must retrieve the data from slower cache levels or the main memory, yielding a cache miss and consequently a delay. The attack can also work across different processing cores if there is a shared cache level. Measuring this kind of delay during the encryption or decryption of a given implementation can allow the attacker to infer properties of the key being used. This type of attack works against insecure implementations of AES, as shown by [Bernstein 2004, Percival 2005]; and the RSA cryptosystem [Yarom and Falkner 2014, Yarom et al. 2017].

There are several works in the literature that detected and exploited implementations of cryptographic algorithms using variable-time instructions or compiler interference. For example, integer multiplication instructions in ARM processors that finish early when the precision of the operands is low can be explored in side-channel attacks [Großschädl et al. 2009]. Another work targeting this approach is [Kaufmann et al. 2016], which studied attacks against the Curve25519 key exchange protocol by analyzing the 32-bit compiled version of an otherwise 64-bit secure implementation. This happened due to the Windows runtime library which added an insecure branch in order to optimize long integer multiplication. There are some microbenchmarks specifically created for different architectures to clarify programmers which instructions are safe and, more importantly, which are not secure and should be avoided or used with some additional precautions [Pornin 2017].

Given the power of timing attacks in the literature and practice of cryptography, the importance of studying how to prevent vulnerabilities and verify the security of cryptographic implementations in terms of side-channel resistance is clear. Many approaches tried to prevent such vulnerabilities by creating tools to be used by the programmer to improve the code for critical applications. The main types of analysis employed by these tools are *dynamic* and *static*, with their own features and limitations, as discussed in detail on the sections below.

2.1. Dynamic Analysis

A first method to detect if an implementation leaks timing information is to analyze its behavior while executing given some input. This kind of method is called dynamic since it relies on an experiment using the compiled version of the code during execution, and not just the source code itself.

A recently proposed tool called *dudect* [Reparaz et al. 2017] was created using the concept. It tests an implementation against several inputs, trying to find a different behaviour that yields a variant running time. In order to detect such behaviour, it uses some statistical tests, such as the Welch's t-test for detecting variations in the collected samples. An advantage of dynamic analysis is detecting low-level timing variances introduced by the instructions themselves, such as early-abort multipliers available in the ARM architecture[Großschädl et al. 2009]; or by the compiler through insecure runtime libraries [Kaufmann et al. 2016].

Another tool for this kind of analysis is *ctgrind* [Langley 2010], based on the pop-

ular *Valgrind* tool [Nethercote and Seward 2007] for memory management and leakage analysis. The *ctgrind* tool changes the behavior of *Valgrind* to check if secret data was accessed and then concludes that a branch instruction depends on secret information, ultimately marking the implementation as insecure. The modified behaviour can be used through special functions provided by *ctgrind* to treat secret parameters as uninitialized data, triggering the warnings.

The main limitation of these tools is that they cannot check if a given implementation is in fact constant-time, because this would imply that every possible input was tested and that the average execution time of every sample has the same value, which is impossible given the number of possibilities. Furthermore, unlikely inputs carefully crafted by an attacker to produce variable time behavior may not be explored in a small number of samples. However, they can still provide some supporting evidence of timing behavior and help the programmer fix his/her implementation.

Each tool has a different method to be used which can vary according to the functions executed by the program. Both dynamic tools studied use a similar representation process to test a given implementation. The *dudect* tool requires a C source file with a function to generate the random input and calculate the execution time of a sample, and a wrapper for each function the user wants to check. The user must just initialize any required variables and use the randomly generated data as input. Another parameter that can be changed is the number of samples taken, usually ranging from then thousand to one million. The tool will output two possible results:

- *For the moment, maybe constant time*, which means that so far every sample taken has a measurement very close to each other, so it cannot confirm that the function is indeed variant or constant time.
- *Probably not constant time*, which means that the mean from two or more samples have enough difference to indicate that the function is not constant-time.

The *ctgrind* tool requires a C source file containing the function to be tested and the special initialization of the secret data. To compile the code, the user must also use some special flags provided in the Makefile in order to be able to use the *Valgrind* software to check the binary.

2.2. Static Analysis

Another type of analysis, known as static analysis, is widely applied to find vulnerabilities involving sensitive variables used in a cryptosystem. This kind of analysis receives the moniker static because it works based on an intermediate representation of the code, instead of the execution of a compiled binary.

Static analysis tools can detect several types of problems within an implementation without compiling and executing the software itself. To perform this kind of analysis the user simply executes the tool using an auxiliary file containing information about the code that he wants to check and the source file. The user will usually receive several warnings describing how the secret information flows within the application. Some recent tools like FlowTracker[Rodrigues et al. 2016] and *ct-verif* [Almeida et al. 2016] use techniques for analyzing programs in intermediate representation languages.

The *ct-verif* tool requires the source file to be annotated using a special syntax and converts a compiled program represented in the LLVM intermediate representation to a

special representation in the Boogie language which allows to extract provable guarantees of constant-time behavior. This allows the user to detect branch instructions using secret information as one of the parameters, therefore being vulnerable since the running time will depend on those parameters. Unfortunately, the tool appears rather preliminary for general use and under development, so it was not possible to benchmark it in this work.

FlowTracker [Rodrigues et al. 2016] is a static tool developed to detect and analyze the implicit flow of information within an implementation. It can be used to keep track of sensitive input, such as cryptography keys, and determine if its value may have some influence in the running time, for example by being the parameter for a branch instruction or the index of a vector possibly stored in cache memory. The main limitation of these tools is that it cannot check all dependencies from a given code, therefore vulnerabilities will not be found if insecure functions or instructions are used.

FlowTracker has a distinct kind of input since it will not compile and execute the code. As explained in Section 3 the input consists of an XML file with precise information about the functions being tested. The output is a little bit more involved. There are two file formats generated by the tool.

- *Subgraph file*: Indicates the nodes in which secret information was propagated, from the declaration to a branch.
- *ASCII file*: indicates the lines in the code that propagate sensitive information to a branch or array index. These can be used by the programmer to detect and fix the vulnerability.

Table 1 below shows the main advantages and limitations of each tool.

Table 1. Comparison of different tools for analyzing constant-time behavior.

Tool name	Type	Limitations	Advantages
dudect	Dynamic	Cannot prove that an implementation is constant time. Takes a long time to run for covering a large portion of inputs.	Usability and detection of microarchitecture effects.
ctgrind	Dynamic	Cannot prove that an implementation is constant time or test all possible inputs.	Usability (reasonably easy to set up and run).
ct-verif	Static	Preliminary and under development.	Provides formal constant-time guarantees at high level, but does not detect microarchitecture effects.
FlowTracker	Static	Cannot check all dependencies automatically or detect microarchitecture effects.	High efficiency and coverage of all possible inputs through evidence collected from information flow analysis.

3. Benchmarking database

In this section, we describe how our benchmarking database was constructed, by starting from the annotation process. Because each dynamic or static tool has specific characteristics, they also require a representation file containing a compilation of the main properties that the user must give in order to perform the analysis. Given that most of information from these files is the same and can be shared among different tools, such as name of the function, parameters and some others, we decided to create a standard XML file to be used by different tools. With this general file, the user can easily analyze the code being written by various tools, expanding the verification and therefore being able to find a larger number of vulnerabilities. Another advantage of this representation is that the user can create this file once and use several times in different tools by just changing a few tags, facilitating the automation of the verification process and detection of potential regressions due to changes in the code or compiling toolchain.

The XML created to describe an implementations consists of a main tag representing each function to be tested (tag `function`), including the name and all parameters, separated in two sections: `public` and `secret`. Also, the user can use a special tag `return` to determine if the return from a given function must be treated as a secret by assigning the tag to *true*. Just like any XML file, the user can use comments to specify the meaning of each parameter received by the function. For illustration purposes, the XML file below refers to the representation of Algorithm 1 presented in Section 1.

The FlowTracker static analysis tool was already to designed to receive an input XML with annotations in a similar format, although less usable. Some illustrative examples can be found in the project website[1]. The FlowTracker format required the programmer to refer to sensitive parameters only by using their number in the prototype list of arguments. We performed all the necessary modifications to adapt FlowTracker parsing in order to integrate the new representation file with the tool. The entire benchmarking database is publicly available on GitHub[2].

```
<functions>
   <sources>

      <function>
         <name>compVar</name> <!--Function to be analyzed-->
         <return>false</return> <!--Return value is not critical-->
         <public>
            <parameter>in</parameter> <!--Input String-->
         </public>
         <secret>
            <parameter>pw</parameter> <!--Password-->
         </secret>
      </function>

   </sources>
</functions>
```

Listing 3. XML file used to annotate code in Listing 1.

[1] `http://cuda.dcc.ufmg.br/flowtracker/example.html`
[2] CTBench - `https://github.com/arthurlopes/ctbench`

Table 2. List of algorithms in our benchmarking database, containing implementations from the BearSSL and NaCl cryptographic libraries and examples from the *dudect* dynamic analysis tool.

Library	Algorithm Category	Constant	Variant
BearSSL	Symmetric	18	4
	MAC	1	2
	Hash	3	5
	RSA	3	4
	ECC	0	4
dudect examples	AES	1	1
	ECC	1	1
	Others	1	1
NaCl	Authenticated encryption	6	0
	Hash	4	0
	Curve25519	1	0

Another contribution proposed by this paper was the creation of a benchmarking database containing several cryptograpihc functions used to analyze the tools. To create such a database we used implementations from the cryptographic libraries BearSSL[3], NaCl[4] [Bernstein et al. 2012] and examples provided by the *dudect* tool. In total the database has about 60 implementations detailed in Table 2 below.

More precisely, the database contains three libraries, the first one is the *dudect* examples, consisting of two AES implementations, two comparison algorithms and two Elliptic Curve Cryptography (ECC) implementations. The second one, BearSSL, contains about twenty implementations of the AES and four of the DES block ciphers. Regarding hash functions, there are implementations of both MD5 and SHA. There are also three implementations of Message Authentication Codes (MAC) and a few of ECC algorithms. The last library is NaCl, containing six algorithms related to Authenticated encryption, two hash implementations (SHA256 and SHA512) and one implementation of scalar multiplication in a specific prime curve (Curve25519) [Bernstein 2006].

The groundtruth classification in constant or variable-time implementation was obtained through the official documentation of each project. To complete the database, all representation files used by the FlowTracker, as well as every C source file required by *dudect* were created in order to run the tools correctly.

4. Experimental results

The results were obtained by evaluating the static and dynamic verification tools across our entire benchmarking database.

We started by analyzing the database and rating each implementation as constant or variable time with the static tool FlowTracker. After the static analysis was finished we

[3]BearSSL - A smaller SSL/TLS library - `https://www.bearssl.org`

[4]NaCl: Networking and Cryptography library - `https://nacl.cr.yp.to`

moved to the dynamic approach.

Next, the *dudect* tool was used to check all implementations classified as constant time by FlowTracker, which as expected produced the output *probably constant-time implementation*, since every sample taken had the same mean, producing an inconclusive result. To obtain a reliable result we executed most implementation about ten million times. For some implementations that required more time to run this value was reduced to five million or increased to twenty million, always running for at least fifteen minutes.

Only one implementation showed different behaviour due to an insecure function used (*memcmp* with early-abort comparison). FlowTracker classified this implementation as constant because it did not analyze the code for this function, but fortunately *dudect* detected the vulnerability. This implementation is a comparison function from the *dudect* examples, the variant *cmpmemcmp* using `memcmp()` implemented in variable time.

After using the *dudect* to check all implementations, we executed the same tests with the *ctgrind* tool. The original *ctgrind* patch was adapted to the latest release 3.13 of Valgrind since the GitHub version was created in 2010 and was not compatible with recent versions. The tools produced equivalent results, an expected outcome given that both use similar analysis (dynamic).

Finally we studied the behavior of implementations classified as variable time by FlowTracker. Both dynamic tools were able to find vulnerabilities in every implementation. For those implementations, FlowTracker was able to provide the corresponding subgraph indicating where the secret information is being used, along with the affected branch instructions. The tools *dudect* was able to find samples with different means and *ctgrind* found traces of uninitialized memory being used. Considering all those results we can conclude that the code indeed does not run in constant time regarding the input and the tools behave as intended.

5. Conclusions and future work

Given the amount of code analyzed from the libraries, those tools can indeed be used by a programmer to improve the code and mitigate timing attacks. The usage of those tools is a recommended practice that can be easily applied by developers to help them create more secure implementations, avoiding the leakage of sensitive and crucial data. We observed that a combination of static and dynamic analysis must be used to fully characterize the timing behavior of a cryptographic implementation, since the binary compiled version can use insecure functions or instructions that the static analysis is not able to detect.

We plan to extend the benchmarking database to include more libraries and examples to be used by several tools, analyzing their performance, effectiveness and the correctness. With a large database we will able to find more examples of implementations that result in different results when using dynamic and static analysis.

Additionally, we will study how those analysis tools could be integrated to commonly used development tools, in order to facilitate the usage by developers who are not fully aware about side-channels attacks work against insecure implementations. In terms of usability, our representation format will be extended to include information about external dependencies such that static analysis tools can produce a warning in case a specific portion of code is not available for analysis.

6. Acknowledgements

We thank Intel and FAPESP for financial support though process 14/50704-7.

References

Aciiçmez, O., Koç, Ç. K., and Seifert, J. (2007). Predicting secret keys via branch prediction. In *CT-RSA*, volume 4377 of *Lecture Notes in Computer Science*, pages 225–242. Springer.

Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., and Emmi, M. (2016). Verifying constant-time implementations. In *USENIX Security Symposium*, pages 53–70. USENIX Association.

Bernstein, D. J. (2004). Cache-timing attacks on AES. URL: `http://cr.yp.to/papers.html#cachetiming`.

Bernstein, D. J. (2006). Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer.

Bernstein, D. J., Lange, T., and Schwabe, P. (2012). The security impact of a new cryptographic library. In *LATINCRYPT*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer.

Brumley, D. and Boneh, D. (2005). Remote timing attacks are practical. *Computer Networks*, 48(5):701–716.

Großschädl, J., Oswald, E., Page, D., and Tunstall, M. (2009). Side-channel analysis of cryptographic software via early-terminating multiplications. In *ICISC*, volume 5984 of *Lecture Notes in Computer Science*, pages 176–192. Springer.

Gullasch, D., Bangerter, E., and Krenn, S. (2011). Cache games - bringing access-based cache attacks on AES to practice. In *IEEE Symposium on Security and Privacy*, pages 490–505. IEEE Computer Society.

Kaufmann, T., Pelletier, H., Vaudenay, S., and Villegas, K. (2016). When constant-time source yields variable-time binary: Exploiting curve25519-donna built with MSVC 2015. In *CANS*, volume 10052 of *Lecture Notes in Computer Science*, pages 573–582.

Kocher, P. C. (1996). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer.

Langley, A. (2010). ImperialViolet: Checking that functions are constant time with Valgrind. `https://www.imperialviolet.org/2010/04/01/ctgrind.html`.

Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM.

Percival, C. (2005). Cache missing for fun and profit. In *Proceedings of BSDCan*.

Pornin, T. (Accessed in July 2017). BearSSL - Constant-Time Mul. `https://www.bearssl.org/ctmul.html`.

Reparaz, O., Balasch, J., and Verbauwhede, I. (2017). Dude, is my code constant time? In *DATE*, pages 1697–1702. IEEE.

Rodrigues, B., Pereira, F. M. Q., and Aranha, D. F. (2016). Sparse representation of implicit flows with applications to side-channel detection. In *CC*, pages 110–120. ACM.

Schindler, W. (2000). A timing attack against RSA with the chinese remainder theorem. In *CHES*, volume 1965 of *Lecture Notes in Computer Science*, pages 109–124. Springer.

Yarom, Y. and Falkner, K. (2014). FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732. USENIX Association.

Yarom, Y., Genkin, D., and Heninger, N. (2017). CacheBleed: a timing attack on OpenSSL constant-time RSA. *J. Cryptographic Engineering*, 7(2):99–112.