Give Me Your Binary, I'll Tell You If It Leaks

Antoine Bouvet*§, Nicolas Bruneau*, Adrien Facon*‡, Sylvain Guilley*†‡ and Damien Marion*†
*Secure-IC S.A.S., Think Ahead Business Line, 35510 Cesson-Sévigné, France
firstname.lastname@secure-ic.com
†LTCI, CNRS, Télécom ParisTech, Université Paris-Saclay, 75013 Paris, France
firstname.lastname@telecom-paristech.fr
‡Département d'informatique de l'ENS, École normale supérieure, CNRS, PSL Research University, Paris, France
firstname.lastname@ens.fr

***Abstract*—In this paper we present a method to identify side-channel information leakage of a cryptosystem software implementation, which is performed at the binary level, and needs only a debugger. Using a new resynchronisation method based on the control flow, leaking instructions are retrieved with only few traces and without leakage model. Advantageously this methodology is target agnostic, finding the side-channel leakages without the need to know how the software will be used.**

***Index Terms*—GDB, debugger, secure software, security verification tool, control flow, synchronisation, Side Channel Analysis, Correlation Power Analysis, AES.**

I. Introduction

Software implementations of cryptographic algorithms generate long traces. When there is a protection, such as masking, the traces are even longer, and can be desynchronised (because the goal is to analyse compiled code, which can be third party, i.e., one does not know the source code). Indeed, in the smartcard industry, the software code is hand-written with great care, and is reviewed, in order to match high level of assurance (e.g., CC EAL5+). In the IoT industry, though, the software is more complex and, for economical reasons, cannot be reviewed with the same level of attention; hence the need of a methodology to identify automatically side-channel leakage early.

Even using masking schemes, it is not easy to write secure software against side-channel attacks, mainly because security regressions can occur at each stage of the refinement flow. For instance, side-channel leakage can come from the algorithm if the protection is flawed [1] or from the source code. The intermediate representation can use a wrong processing order of some variables. At the assembly stage, the register allocation can be insecure (e.g. when information on the sensitive variable X is revealed if a register contains successively M and $X \oplus M$). The compiled code can introduce a misalignement of the code between two executions, e.g. because of conditional branching. Finally, at the gate-level netlist stage, the code can be rescheduled, causing again some problems of self-demasking in a given register.

Many works have focused on verifying leakages on simulators adapted to the final target. For instance, Program Inferred Power Analysis Simulator (PINPAS) [2] is composed of two parts: a simulator and an analyser. Developers need to choose which type of microcontroller (smart card) they want to simulate with the written code. The PINPAS simulator has been improved by Thuillet *et al.* [3] by making it able to focus the side-channel measurements on some parts of the circuit.

Barthe *et al.* introduce in 2015 [4] another approach based on formal methods. It unrolls the computation symbolically and resorts to a SAT-solver in order to find weaknesses in masking schemes. However, it faces a problem of combinatorial explosion.

Contributions: In this paper we present a security evaluation of cryptographic code, with a methodology which is in-between PINPAS and formal methods, taking the best of both ones: target agnostic (like formal methods) but able to detect leakage even of complex software (dynamic approach). Aiming to perform practical and efficient high-level security evaluation, our method allow to identify first-order leakages. Contrary to PINPAS approach, which aims at simulating at high-level accurate traces which match the physical traces, our method only require the use of debugger.

Interesingly, we show that our evaluation method reveals the same leakages that occurs while executing a cryptographic algorithm in a x86 architecture. The practicability of the proposed analysis allows to detect and fix the leaking instructions at early stage of the development cycle. As using a debbuger we have directly access to the value of the variables and thus the analysis is performed on the values in absence of leakage model contrary to traditionnal side-channel analysis methods.

Lastly, our methodology introduces a new resynchronisation method. Indeed desynchronisation could limit the accuracy of the analysis. Our methodology resolves this by storing multivariate data extracted from both data flow and the control flow.

Outline: The following is organized as follow: we introduce our methodology in Sec. II then present analysis results on an unprotected AES-128 implementation in Sec. III working on the one hand on real traces measured on an Atmel AVR XMEGA microcontroller or on the other hand on traces generated with GDB. Performance of our method are discussed in Sec. IV.

II. Setup

In our method, we use GDB, the GNU project debugger, as assembly-level debugger. This tool supports many programming languages and several operating-systems (OS) [5]. To make the method general and easy to setup, we suggest not to use cross-compilation, but rather to work directly on the host machine (in our case, a 64-bit x86 architecture), since the final device is not considered. We use as main side-channel the accumulator register (RAX), on account of its use in arithmetic operations to store results and functions return value.

A. Using GDB to extract data as a side-channel.

We recall in this section the basic commands to compile & debug a program. One assumes that the source code named `a.c` is a software implementation of a cryptosystem written in the C programming language. Using the GNU Compiler Collection (GCC), the simplest Linux command to create the executable `a.out` with debugging information is `$ gcc -g a.c`, where the `-g` GCC option is used to keep track of the link between machine code and source code.

Then, one starts GDB with the binary file `a.out` as parameter. We then monitor the selected registers by placing a watchpoint on them, the `watch` command beeing applied on the name of the targeted register e.g., `(gdb) watch $rax`.

In addition to data registers, the instruction pointer register RIP[1] is also informative, especially for traces resynchronization. The state-of-the-art method consists in resynchronising traces through data analysis. In this article, we use an new alternative method by leveraging RIP register values, namely program counters (PC), to resynchronise the data leakage traces. Hence when PC series are the same for all the datasets, the traces are properly synchronised.

Key and plaintext are set within GDB then the program is run until its end using the `continue` command. In order to focus on the relevant values, constant register values are saved as ones and register modifications are monitored.

We stress that each register can be viewed as a potential source of side-channel leakages. Advantageously these traces can be analysed without leakage model by considering directly raw values for improved performances.

The different steps needed to extract n traces are summarised in Fig. 1.

B. Analysis.

Each set of samples represents an execution trace. One analyses them with the Normalized Inter-Class Variance (NICV) [6] as metric to detect the leakages—it detects the worst leakages with few traces, thereby reducing the sample range to analyse, and speeding up the process—and the Correlation Power Analysis (CPA) [7] to perform attacks on the relevant samples—the CPA is used to recover the secret key. Using the CPA, the greatest Pearson correlation coefficient on the considered sample indicates the best subkey guess among all the possibilities. If this guess corresponds to the correct subkey, then this last one is broken. Such evaluation assumes a white-box attacker [8] since one knows the secret key and one needs to access the memory thanks to registers.

If the traces are synchronised, then each sample corresponds to only one program counter. For this reason, if the best guess is the correct subkey, then one stores the index numbers of all the correlation peaks, i.e. those corresponding to all the coefficients of the best guess, greater than the maximum of the second best guess. In fact, each of these maximal coefficients can be used to recover the correct subkey among all guesses; that is why each peak must disappear by fixing, afterwards by the developer, the corresponding leaking instruction.

C. Extraction of the leaking instructions.

In the following, we assume that all the samples are synchronised. We have now a list of time indices where the implementation leaks. In order to pinpoint these leakages in the binary, one needs to identify the leaking instructions. Since the stored PCs (and their matched registers value) are only the ones corresponding to the reference time indices, we just need to look at the source instruction and/or the assembly instructions at each activated watchpoint.

III. Results

We apply our method on an unprotected AES-128 implementation [9] written in C provided as part of the CHES 2016 CtF challenge[2]. The same analysis is here performed on two datasets: real SCA traces (III-A) compared with traces generated with GDB as explained above (III-B).

A. Real traces.

First, we use the real traces provided by the challenge[3]. We use the version with a known fixed key and random plaintexts. Fig. 2 illustrates some of these real traces.

Ten curves are superimposed in grey. The repetition of a pair of two distinct patterns is clearly visible eight times. In order to highlight the structure of the AES algorithm, we represent the mean of all the traces in blue. This removes the data dependency, while keeping the control flow information. This last curve is superimposed on the others at the beginning of the first pattern, but it quickly decays over time because of a lack of synchronisation that piles up. The standard deviation represented in green color highlights the same fact. Both statistics reveal clearly eight AES rounds.

Secondly, we attack the first AES round without synchronising it. Please note, however, that the physical behaviour of a device is to leak the sum of the bits, because each register leaks one value and the power consumption is additive. Now, the leakage model for the CPA must correspond to the

[1]RIP is also called Program Counter register (PC). Contrary to RIP, this denomination does not depend on the working architecture.

[2]This implementation can be found as "Stagegate #1", at ctf.newae.com/flags.

[3]They have been automatically captured on a ChipWhisperer-Lite—which is a capture hardware dedicated to side-channel analysis—using an Atmel XMEGA firmware.

$$\begin{tikzcd}
\boxed{\text{Source code}} \arrow[r, "\text{Compiler}"] & \boxed{\text{Binary file}} \arrow[r, "\text{Debugger}"] & \boxed{\text{Data}} \arrow[r, "\text{Synchronisation}"] & \boxed{\text{Traces}} \\
\texttt{a.c} \arrow[r, mapsto, "\texttt{gcc}"] & \texttt{a.out} \arrow[r, mapsto, "\texttt{gdb}", "n \text{ times}"'] & \texttt{a.gdblog} \arrow[r, mapsto, "\text{Parsing}"] & \texttt{a.gdbtra}
\end{tikzcd}$$

Figure 1. Traces generation using GDB.

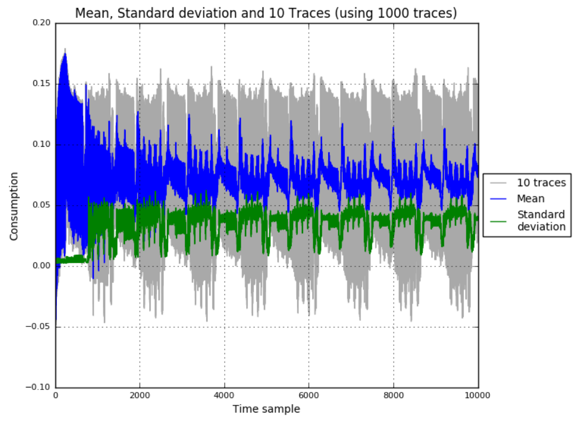

Figure 2. Some curves, along with some statistics, amongst the 1,000 recorded of an unprotected AES-128 implementation.

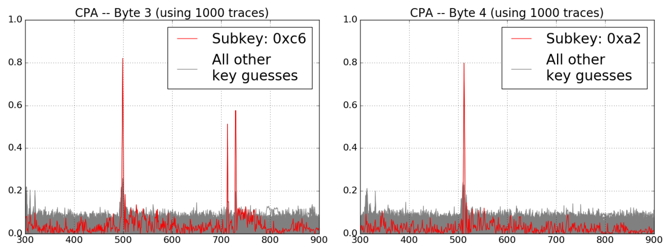

Figure 3. CPAs performed with L_1 on 1,000 recorded traces. X axis represents the time, and Y axis the correlation coefficient. All the subkey hypotheses are denoted in grey, and the correct subkeys are denoted in red.

leakages, hence we use the Hamming weight as leakage model. Therefore we use the leakage function

$$L_1(p[i]) = \texttt{HW}(\texttt{SBox}[p[i] \oplus k[i]]) \tag{1}$$

where $p[i]$ and $k[i]$ denote the plaintext and the key bytes i ($1 \leq i \leq 16$) respectively. The analysis is done for all the sixteen key bytes; nevertheless, for the sake of clarity, results for only the bytes 3 and 4 are shown in Fig. 3. The grey curves represent the correlation coefficients obtained for all the 256 subkey hypotheses. The correct subkeys are depicted in red and present high peaks of correlation up to 0.8. Even if the number of peaks is not the same for all subkeys, the great difference between the red curve and the grey curve maxima could mean that any attacker would be able to find the correct key easily, i.e. with only few traces. As expected, all the subkeys are broken so this implementation is not secure against such side-channel attacks.

B. *Comparison with our target agnostic GDB method.*

Now, we use the GDB method to generate traces. Thanks to the PCs stored in parallel to the RAX values, we are able to find the start and the end of any AES operation for each trace; thus we cut the extra part before and after the first AES round, and thereby aligning all the traces on its first sample.

On such GDB traces, state-of-the-art SCA trace resynchronisation methods [10], [11], [12] are inappropriate because no external noise affects our traces nor low-pass filtering stage allowing to average instant variations over time leaving more evidence in the traces for the scheduling of the algorithm. That is why we have been pushed to devise a novel resynchronisation technique based on a bi-variate side-channel measurement: we extract the pair made up of the accumulator and the PC. The former contains the leakage about the data whereas the latter contains the information about the position in the code, i.e., the schedule of the algorithm. In order to perform the synchronisation, first, one constructs PC's timeline according to reference instructions, then, one uses a local instruction insertion or drop.

In this case, the desynchronisation comes from the xtime function and the conditional branch defined by the source instructions `if (x & 0x80) {...}` and `else {...}` which introduce the respective assembly instructions `jns` (for "Jump No Sign", a conditional jump which makes a jump if $\mathsf{MSB}(x) = 0$) and `jmp`. When the synchronisation is done, all the AES functions can be exactly delimited according to the PCs, as illustrated on traces obtained by our GDB method in Fig. 4. This definitely allows to reverse-engineer the structure of the algorithm under analysis, as already noted in [13].

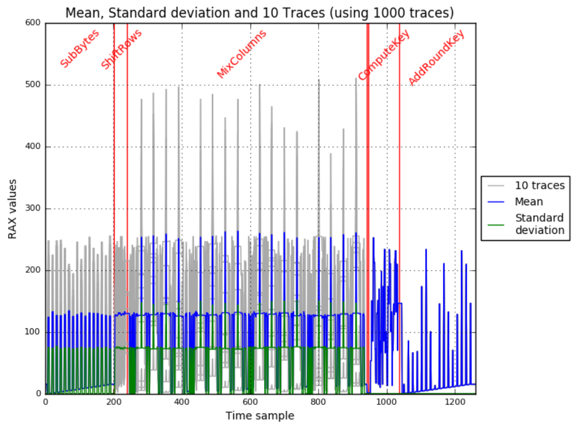

Figure 4. Some curves, along with some statistics, amongst the 1,000 generated with GDB. These traces have been synchronised and the functions are delimited (the xtime function is intentionally not delimited on this figure). The sample is the whole AES first round.

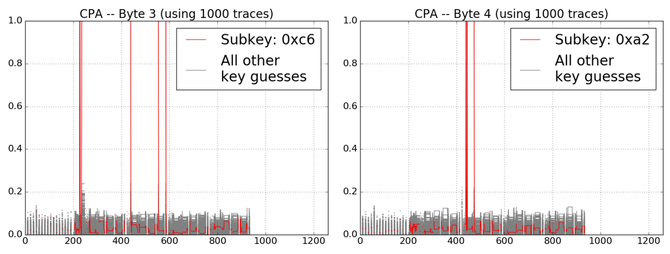

Figure 5. CPAs performed with L_2 on 1,000 GDB traces after synchronisation. X axis represents the instruction count, and Y axis the correlation coefficient. All the subkey hypotheses are denoted in grey, and the correct subkeys are denoted in red.

We perform a new CPA on this first AES round without using any leakage model, since RAX values can be directly measured. So the leakage function becomes:

$$L_2(p[i]) = \mathtt{SBox}[p[i] \oplus k[i]] \tag{2}$$

Compared to (1), we can get rid of the non-injective HW function, which will definitely make former analyses more accurate. As previously, we illustrate in Fig. 5 the results for only the bytes 3 and 4. For each byte, the grey correlation coefficients approximately reach 0.2, whereas the red curves have several distinct correlation peaks, all equal to 1. There are three correlation peaks for each byte, as well as many others for the byte 3. All these peaks allow an attacker to break all the secret key bytes with few traces, which confirms that this AES-128 implementation is not secure against first-order side-channel attacks. Furthermore, they are all equal to 1, i.e. the correlation peaks exactly match the leaking assembly instructions, so we store their index numbers.

To further investigate the leakage instructions after synchronisation, we illustrate the analysis on subkey 4. The first source of leakage is data movement instructions (`mov` and `movzbl`) occuring during bytes copy of the MixColumns operation. Another source of leakage comes from a Xor operation between some of these bytes. Finally, the last ones come from a Xor and a data movement instructions using the result of the xtime function. Each of these leakages appears with the handling of one part of RAX.

C. Discussion.

It appears that the number of successive peaks for each byte with GDB traces (Fig. 5) matches the number of peaks series found for the same bytes with real traces (Fig. 3). Yet the analysis conditions are not the same: the real traces have been obtained using an Atmel AVR XMEGA microcontroller, i.e. using 8/16-bit CPU with a pipeline, whereas the GDB traces have been generated with a 64-bit x86 architecture with cache. There is nothing in common between both methods. This demonstrates that one can highlight the leakages of a cryptographic code in spite of the architecture. Leakage is thus intrinsic to the C source code; hence the robustness of the present methodology because it allows one to perform side-channel evaluation of a software without any device.

Moreover, the correlation obtained with (2) is bijective with the secret key, i.e. no information is lost, whereas (1) is non-injective. So the correlation reaches 1.0 in Fig. 5, contrary to Fig. 3.

IV. Conclusion

The GDB method is a powerful tool for side-channel evaluation. It allows one to anticipate the use of a cryptosystem in a software and its weakness against such physical attacks. Indeed, this methodology is effective for finding real side-channel leakages considering only numerical values, without the need of taking into account the final device. Besides, it enables really precise evaluations, especially the delimitation of the operations and the extraction of the leaking assembly instructions. Ultimately, because this first (and difficult) step in writing secure software against first-order attacks is to know where the device leaks, the last benefit of this method is substantial: it enables to pinpoint flaws in the protections.

References

[1] A. Moradi, S. Guilley, and A. Heuser, "Detecting hidden leakages," in *Applied Cryptography and Network Security - 12th International Conference, ACNS 2014, Lausanne, Switzerland, June 10-13, 2014. Proceedings*, 2014, pp. 324–342.

[2] J. den Hartog, J. Verschuren, E. P. de Vink, J. de Vos, and W. Wiersma, "PINPAS: A tool for power analysis of smartcards," in *Security and Privacy in the Age of Uncertainty, IFIP TC11 18th International Conference on Information Security (SEC2003), May 26-28, 2003, Athens, Greece*, 2003, pp. 453–457.

[3] C. Thuillet, P. Andouard, and O. Ly, "A Smart Card Power Analysis Simulator," in *CSE (2)*, 2009, pp. 847–852.

[4] G. Barthe, S. Belaïd, F. Dupressoir, P. Fouque, B. Grégoire, and P. Strub, "Verified Proofs of Higher-Order Masking," in *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, 2015, pp. 457–485.

[5] R. Stallman, R. Pesch, and S. Shebs, "Debugging with gdb: The gnu Source-Level Debugger Ninth Edition, for gdb version 20040217," 2012.

[6] S. Bhasin, J.-L. Danger, S. Guilley, and Z. Najm, "NICV: Normalized Inter-Class Variance for Detection of Side-Channel Leakage," in *International Symposium on Electromagnetic Compatibility (EMC '14 / Tokyo)*. IEEE, May 12-16 2014.

[7] É. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, 2004, pp. 16–29.

[8] B. Preneel and B. Wyseur, "White-box cryptography," 2008.

[9] NIST FIPS, "197: Advanced encryption standard (AES)," *Federal Information Processing Standards Publication*, vol. 197, no. 441, p. 0311, 2001.

[10] N. Homma, S. Nagashima, Y. Imai, T. Aoki, and A. Satoh, "High-Resolution Side-Channel Attack Using Phase-Based Waveform Matching," in *CHES*, ser. LNCS, vol. 4249. Springer, October 10-13 2006, pp. 187–200, Yokohama, Japan.

[11] S. Guilley, K. Khalfallah, V. Lomne, and J.-L. Danger, "Formal Framework for the Evaluation of Waveform Resynchronization Algorithms," in *WISTP: Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing*, ser. LNCS, LNCS, Ed., vol. 6633. Springer, June 1-3 2011, pp. 100–115, Heraklion, Greece.

[12] J. G. J. van Woudenberg, M. F. Witteman, and B. Bakker, "Improving Differential Power Analysis by Elastic Alignment," in *CT-RSA*, 2011, pp. 104–119.

[13] J. W. Bos, C. Hubain, W. Michiels, and P. Teuwen, "Differential computation analysis: Hiding your white-box designs is not enough," in *Cryptographic Hardware and Embedded Systems – CHES 2016*, ser. Lecture Notes in Computer Science, B. Gierlichs and A. Y. Poschmann, Eds., vol. 9813. Springer Berlin Heidelberg, 2016, pp. 215–236.