Binary Data Analysis for Source Code Leakage Assessment

Adrien Facon[1,3], Sylvain Guilley[1,2,3], Matthieu Lec'hvien[1], Damien Marion[1,2], and Thomas Perianin[1]

[1] Secure-IC S.A.S, Rennes, France,
firstname.lastname@secure-ic.com
[2] Telecom ParisTech, Institut Mines-Télécom, Paris, France
[3] École Normale Supérieure, Paris, France

Abstract. Side Channel Analysis (SCA) is known to be a serious threat for cryptographic algorithms since twenty years. Recently, the explosion of the Internet of Things (IoT) has increased the number of devices that can be targeted by these attacks, making this threat more relevant than ever. Furthermore, the evaluations of cryptographic algorithms regarding SCA are usually performed at the very end of a product design cycle, impacting considerably the time-to-market in case of security flaws. Hence, early simulations of embedded software and methodologies have been developed to assess vulnerabilities with respect to SCA for specific hardware architectures. Aiming to provide an agnostic evaluation method, we propose in this paper a new methodology of data collection and analysis to reveal leakage of sensitive information from any software implementation. As an illustration our solution is used interestingly to break a White Box Cryptography (WBC) implementation, challenging existing simulation-based attacks.

Keywords: Software analysis, GNU Debugger (GDB), Differential Computation Analysis (DCA), Correlation Power Analysis (CPA), binary analysis, realignment algorithm, Catalyzr^{TM} tool.

1 Introduction

1.1 Previous Work

Measuring Electromagnetic (EM) or power traces from embedded devices to identify potential leakage of information is a time consuming and challenging process. First, it requires equipment (oscilloscope, probes, signal amplifiers...) that demand prior knowledge to the technician. Secondly, the code to evaluate needs to be embedded on the final product or on an evaluation board, it means that the development has to be finished at the evaluation time. This justifies the interest of the security evaluation community for simulation, both in the industrial and academic world. Aiming to speed-up, facilitate the evaluation, and allowing an evaluation during the development, several works have proposed Side Channel (SC) trace simulators. The principle is to simulate the leakage that

might happen during the code execution. Numerous simulators are available to simulate SC (power or EM) leakage. All present a similar general construction flow illustrated in *Fig. 1*. First, they take a description of the implementation to evaluate as input. For example, the inputs of SILK [13] are tagged C++ source codes. More often, inputs are architecture dependant compiled binaries: Elmo [9] uses binaries for the ARM Cortex-M0 and OSCAR [11] uses binaries for the 8-bit Atmel AVR microcontroller. The Dynamic Binary Analyzer (DBA) framework [2] supports more architectures: ARM, x86, MIPS, SPARC and SH4. In the simulation step, the SC-simulators execute the code to record data. For example, Elmo [9] uses the emulator Thumbulator, while in [3] the authors use the Valgrind debugger. The choice of the data provider is influenced by the fact that the simulator is specific to an architecture. Another important choice of the simulation step is the selection of the data to record. The authors of [3] proposed to record the memory accesses. The DBA framework [2] records the stack, the heap, the CPU-registers and the executed instructions. Elmo [9] is focused on the values of the operands, the bit-flips of the operations and the operations. The last step is the trace generation, or how to transform the recorded data to obtained traces as similar as possible to real and effective physical leakage. Elmo [9] provides one of the most realistic and complex model computed with linear regressions (also used in [5]) and F-test on real leakage traces recorded on a ARM Cortex-M0. Otherwise, the most commonly used models are the Hamming weight and the Hamming distance, which are simplifications of effective physical leakage. Those models have been preferred by the authors of [2, 11].

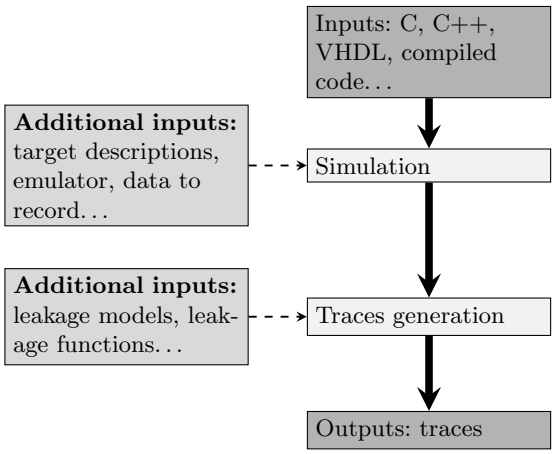

Fig. 1: General flow commonly used by the SC-simulators.

1.2 Contributions

In this context, we propose a new methodology of data collection and analysis to identify potential leakages from any software implementation. Our solution takes inspiration from the work proposed by Bos *et al.* in [3] . An improvement of the analysis step has been proposed in [1], realizing on three intermediate computations instead of one initially. As a first difference, we propose a new methodology of data collection. We record all bit-modifications that happen during a code execution (registers, memory content, flags, Program Counter (PC)), while in [3], authors record only the read, written and executed addresses. Furthermore, we do not apply any model to the recorded data in order to generate traces. All the collected data is exhaustively analyzed to avoid information loss. To do so, we introduce a binary analysis to extract the leaking points from the data. Then, we leverage on PC to keep track of the execution context to map the identified leakage to the source code. Indeed, mapping leakages to source code is nowadays an important need for continuous improvement of products; this is particularly true for software implementations, where traces are long and code is complex. To succeed, we need to overcome two main difficulties: misalignment and multiplicity of leaking resources. We show how to characterize and then exploit the PC to realign all the data using a simple accumulative algorithm, and we introduce a methodology of data selection that significantly reduce the size of the data to analyze. Finally, we show how our solution can identify leakage in the source code applying our methodology to WBC implementation.

1.3 Outline

The paper is organized as follows. In the *Sec. 2* we describe each step of our solution. In the *Sec. 3* we present an evaluation performed with our solution on a WBC implementation. Finally, we summarize the presented work in *Sec. 4*.

2 Solution Presentation

We present here-after how the Catalyzr^{TM} tool collects, pre-processes, and analyses simulation traces.

2.1 Notation and Recording Step

Aiming to record all the data manipulated by a given binary, the debugger GDB is used, but alternative software could be used to collect data. The analysis that we propose is independent of the data provider. To identify all potential leakages, the recording process is as exhaustive as possible. Each time the PC changes, all the internal data are saved (PC, registers, flags...). For example, in the case of an x86 architecture in 64-bit execution (properties of the system used for all the results given in the current paper), the internal data are:

– the sixteen 64-bit registers: `rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8, r9, r10, r11, r12, r13, r14, r15`,
– the six 16-bit registers: `cs, ss, ds, es, fs, gs`,
– the 64-bit `eflags` (with the bit 1, 5, 15, 22-63 reserved).
– the PC (Program Counter, also named `rip`)

In the whole document, the matrix notations are used. The recorded data of an execution is stored in a matrix noted $X^{D,R} \in (\mathbb{Z}/2\mathbb{Z})^{D,R}$, with D the number of times the PC changes and R the number of bit needed to store all the internal data (except the PC that is stored independently, and used in resynchronization process deepened in *Subsec. 2.2*). For a given dataset $X^{D,R}$, the associated list of successive PC-values is stored in a matrix $\mathrm{Pc}^D \in (\mathbb{Z}/2^{64}\mathbb{Z})^D$. An illustration of a recorded trace is provided in *Fig. 2*, the black color corresponds to one and the white to zero, the x-axis describes the internal data and the y-axis the index of the PC. The illustrated trace follows from the execution of the WBC algorithm freely provided at the Challenge CHES-2016[4]. The illustration of the recorded trace provided in *Fig. 2* shows that only a little part of the registers seems to be used during the execution. The reduction of the size of manipulated data is detailed in *Subsec. 2.3*. A set of Q executions is noted as $\{X_q^{D_q,R}, \mathrm{Pc}_q^{D_q}\}_{q<Q}$,

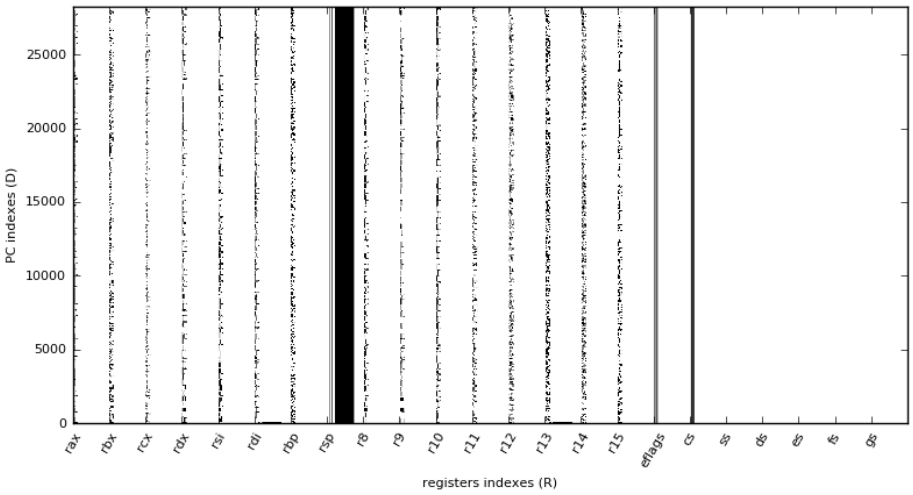

Fig. 2: Representation of a recorded trace of a WBC algorithm using GDB, black points correspond to one values, and white to zero values.

an element of $X^{D,R}$ is noted $X_{d,r}$ and X_d^R (resp. X_r^D) refers to a row matrix (resp. a column matrix). In the context of the SCA, the traces are commonly compared with distributions of intermediate values manipulated by the target

[4] `http://ctf.newae.com/`

and dependent of a secret variable (generally a cryptographic key). We store those distributions in the matrix $Y^{S,Q,K,B}$, with S the dimension of the leakage model, B the number of bytes of the secret and K the number of possible values for each secret bytes (256 if no values are forbidden). For example, if the target algorithm is the Advanced Encryption Standard (AES)-128, and if the focused sensitive intermediate values are the output of the Substitution-box function (SBOX) at the first round, the distribution for a *bit-level* model is expressed as in the following *Formula 1*:

$$Y^{S,Q,K,B} = \left\{ \left(Sbox(P_{q,b} \oplus k) \ \& \ 2^s \right) >> s \right\}_{\substack{s<8, k<256 \\ q<Q, b<16}} \tag{1}$$

The choice of the *bit-level* model is motivated by the bit representation of the recorded data $X^{D,R} \in (\mathbb{Z}/2\mathbb{Z})^{D,R}$. To lighten the notations, the dimension B is not always precised.

2.2 Realignment Algorithm

The first problematic met in the proposed study is the misalignment of the data. Indeed, misalignment could be due to the randomization of the execution, or more generally, by the presence of conditional branches. A vertical alignment is a prerequisite point to realize vertical analysis. Most of the vertical analysis techniques, as CPA [4] or Linear Regression Analysis (LRA) [7] need the data $X_d^{R,Q}$ manipulated at the sample $d < D$ to come from the same operation. The resynchronization is a well known and a well studied problem in the SCA domain [6,8,10,12]. All the proposed algorithms of resynchronization are based on the leaking values distribution in the temporal or in the frequency domain. In our case we have access to additional information thanks to the PC values. In fact, the PC values can be viewed as an identifier. For example $\forall q < Q, \forall d_q < D_q$, $\mathrm{Pc}_{d_q,q}$ is an identifier for the data $X^R_{d_q,q}$. Furthermore, if for $d_0 < D_0, d_1 < D_1$, $\mathrm{Pc}_{d_0,0} = \mathrm{Pc}_{d_1,1}$ it means that the two datasets $X^R_{d_0,0}$ and $X^R_{d_1,1}$ are the result of the same operation in the code (at the assembly level). However the presence of a loop in the source code could imply repetitions of PC value. Hence evince, if the two datasets $X^R_{d_0,0}, X^R_{d_1,1}$ result form the same operation they may come from distinct iterations. Moreover, conditional branching in the code produce misalignment. The goal of the proposed realignment algorithm is to transform the raw dataset $\{X_q^{D_q,R}, \mathrm{Pc}_q^{D_q}\}_{q<Q}$ into the dataset $\{X^{D,R,Q}, \mathrm{Pc}^D\}$ where $\forall\ d < D$, $\forall\ q_0 < Q$, $\forall q_1 < Q$, X^R_{d,q_0} and X^R_{d,q_1} result in the same operation, at the same iteration. The main constraints are the execution time and memory required. We proposed here a single-pass realignment algorithm detailed in *Schedule 1*. The fact that we only need the PC values to resynchronize the data significantly reduces the computational time and the needed memory. Indeed, our algorithm of resynchronization only have to read one time the $D_{q_{q<Q}}$ 64-bit PC values instead of the $\{X^{D_q,R}\}_{q<Q}$ bit of data in the case of an algorithm based on the entire dataset. To provide details on the last assumption

of the *Eq. 2*, the two *Def. 1* are needed. The whole algorithm described in the *Schedule 1* reveals the fix-points and the pseudo fix-points that are automatically realigned.

Definition 1. *Fix-point and pseudo fix-point*

- ***A fix-point*** *is a* PC *value with a deterministic presence and a deterministic number of occurrence:*

$$\begin{aligned} &\mathrm{Pc}_d \in \{\mathrm{Pc}_q^{D_q}\}_{q<Q} \textit{ is a fix-point} \\ &\iff \exists\ m \in \mathbb{N}, \sum_{d_q=1}^{D_q} \begin{cases} 1 & \textit{if } \mathrm{Pc}_{d_q,q} = \mathrm{Pc}_d \\ 0 & \textit{otherwise} \end{cases} = m\ \forall q < Q. \end{aligned}$$

- ***A pseudo fix-point*** *is a* PC *value with a deterministic number of appearance, in the case it appears (so a fix-point is also a pseudo fix-point):*

$$\begin{aligned} &\mathrm{Pc}_d \in \{\mathrm{Pc}_q^{D_q}\}_{q<Q} \textit{ is a pseudo fix-point} \\ &\iff \exists\ m \in \mathbb{N}, \sum_{d_q=1}^{D_q} \begin{cases} 1 & \textit{if } \mathrm{Pc}_{d_q,q} = \mathrm{Pc}_d \\ 0 & \textit{otherwise} \end{cases} \in \{0, m\}\ \forall q < Q. \end{aligned}$$

To illustrate the proposed realignment algorithm, we first start with an application to a simple example. The *Fig. 3* displays a control flow graph with a conditional branching, a loop and a conditional branching inside. The letters A, B, C, ..., I are the PC values. The probability associated to each conditional branching are p_0 and p_1. The presented results have been obtained with $p_0 = 1/2$ and $p_1 = 1/3$. Three distinct executions of the proposed flow graph gave the following PC successions:

$$\{\mathrm{Pc}^{D_q}\}_{q<3} = \begin{cases} \mathrm{Pc}^{D_0} = A, B, D, E, H, I, E, H, I, E, F, G, I \\ \mathrm{Pc}^{D_1} = A, B, D, E, F, G, I, E, H, I, E, F, G, I \\ \mathrm{Pc}^{D_2} = A, C, D, E, H, I, E, H, I, E, F, G, I \end{cases} \tag{2}$$

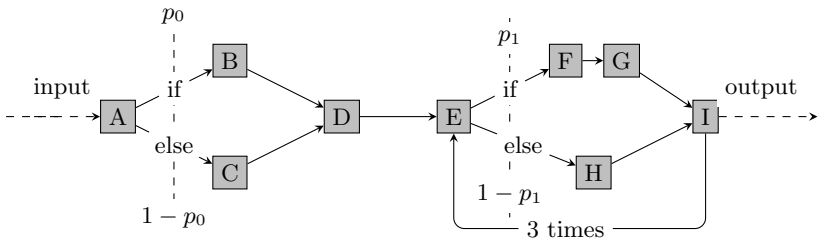

Fig. 3: Control flow of the presented example

1. get the set of all possible PC values:

   $\mathrm{Pc}^{D'} = \bigcup_{q=0}^{Q-1} \mathrm{Pc}_q^{D_q}$, with D' the number of distinct possible values of PC over Q

2. accumulate in the matrix $\#\mathrm{Pc}^{D',Q}$ the numbers of times each PC appears in each trace:

$$\#\mathrm{Pc}^{D',Q} = \left[\#\mathrm{Pc}_{d,q} = \sum_{d_q=0}^{D_q-1} \begin{cases} 1, & \text{if } \mathrm{Pc}_{d_q,q} = \mathrm{Pc}_d \\ 0, & \text{otherwise} \end{cases} \right]_{\substack{d<D' \\ q<Q}}$$

3. the fix-points and the pseudo fix-points are stored in F^D with its associated number of appearance:

$$\mathrm{F}^{D'',2} = \{(\mathrm{Pc}_d, m_d) \in (\mathrm{Pc}^{D'}, \mathbb{N}) | \exists\ m \in \mathbb{N},\ \#\mathrm{Pc}_{d,q} \in \{0, m\},\ \forall\ q < Q, \}$$

4. finally the axis $\mathrm{Pc}^{D(=\sum_{d=1}^{D''} m_d)}$ of PC used for the alignment is created using $\mathrm{F}^{D'',2}$:

$$\begin{aligned} \mathrm{Pc}^D &= \{\overbrace{\mathrm{F}_{0,0}, \ldots, \mathrm{F}_{0,0}}^{\mathrm{F}_{0,1} times}, \ldots, \overbrace{\mathrm{F}_{D''-1,0}, \ldots, \mathrm{F}_{D''-1,0}}^{\mathrm{F}_{D''-1,1} times}\} \\ &= \{\overbrace{\mathrm{Pc}_0, \ldots, \mathrm{Pc}_0}^{m_0 times}, \ldots, \overbrace{\mathrm{Pc}_{D''-1}, \ldots, \mathrm{Pc}_{D''-1}}^{m_{D''-1} times}\} \end{aligned}$$

Schedule 1: step-by-step description of the realignment algorithm

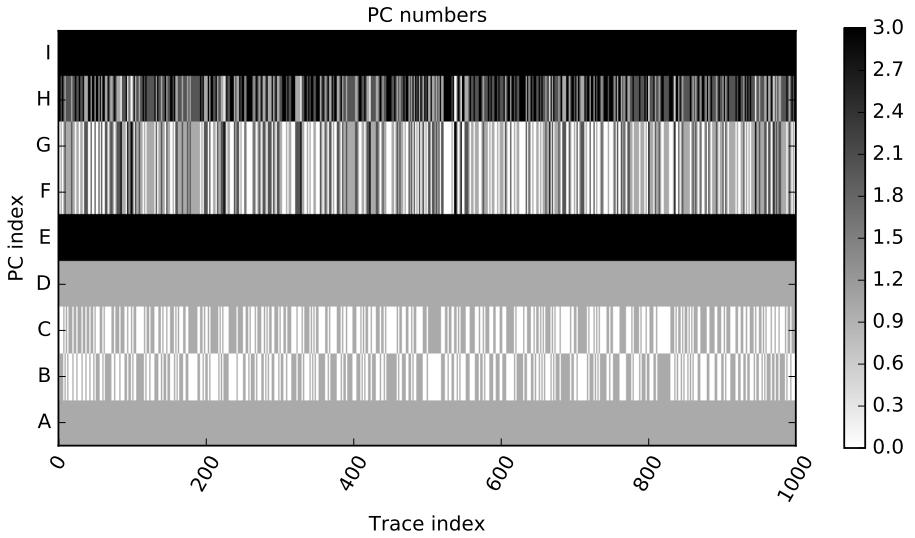

Fig. 4: $\#\mathrm{Pc}^{D',Q}$ for $Q = 1000$ executions of the flow graph described in *Fig. 3*, with $D' = 8$ and $\mathrm{Pc}^{D'} = \{A, B, C, D, \ldots, I\}$. The colors refer to the number of time each PC appear in each execution.

The application of the proposed realignment algorithm, described in the *Schedule 1*, to a thousand execution of the flow graph detailed in *Fig. 3*, gave the matrix $\#\mathrm{Pc}^{D',Q}$ displayed in *Fig. 4*.

The realignment algorithm designates A, D, E and I as fix-points; B, C as pseudo fix-points while the PC F, G and H could not be realigned in the preliminary study. Those PC need more information to be realigned. Finally, the realignment algorithm gives the following output:

$$\mathrm{F}^{D''} = \{(A,1),(B,1),(C,1),(D,1),(E,3),(I,3)\}$$
$$\mathrm{Pc}^{D} = \{A, B, C, D, E, E, E, I, I, I\}.$$

If we go back to the three execution traces given in the *Eq. 2*, the resynchronization is done as written in *Fig. 5*, with in red the elements affected by the realignment.

The algorithm of realignment have been applied to a real-world masked implementation of an AES-128, the results obtained are displayed in *Fig. 7*. It reveals that some PC, around the index 100 seem to be neither fix-points nor pseudo fix-points. This aspect is confirmed by the results of the computation of the mean and the standard deviation of $\#\mathrm{Pc}^{D',Q}$ over Q displayed in *Fig. 7*. In fact, this three figures show that two PC values have non-constant number of apparition in all the executions. In addition to give the PC that causes the misalignment and the resynchronized axis of PC, the algorithm helps to find

$$\{X^{D,q}\}_{q<3} =$$

$$\begin{cases} X^{D,R,0} = \{X^{0,R,0}, X^{1,R,0}, 0, X^{2,R,0}, X^{3,R,0}, X^{6,R,0}, X^{9,R,0}, X^{5,R,0}, X^{8,R,0}, X^{12,R,0}\} \\ X^{D,R,1} = \{X^{0,R,1}, X^{1,R,1}, 0, X^{2,R,1}, X^{3,R,1}, X^{7,R,1}, X^{10,R,1}, X^{6,R,1}, X^{9,R,1}, X^{13,R,1}\} \\ X^{D,R,2} = \{X^{0,R,2}, 0, X^{1,R,2}, X^{2,R,2}, X^{3,R,2}, X^{6,R,2}, X^{9,R,2}, X^{5,R,2}, X^{8,R,2}, X^{12,R,2}\} \end{cases}$$

Fig. 5: Results of the resynchronization

the lines in the source code that provoke the misalignment. This matching from the PC values to the source lines code is made easier by the usage of GDB to record the leakage.Furthermore, the localization of the misalignment origins helps to identify timing leakage. In our example illustrated in *Fig. 7*, the realignment reveals that the misalignment is caused by the function *xtime* transcribed in *Fig. 6a*. This function multiplies the input b by two in GF(2^8), but this implementation contains conditional statement that misalign the data and that could produce time leakage. A possible improvement can be the usage of the constant time implementation of *xtime* provided in *Fig. 6b*. Our realignment algorithm immediately and precisely identify the non-constant time line in the source code. This information is very useful for a developer that want to implement constant time algorithm aim to protect his code against the *timing attacks*.

```
unsigned char xtime (unsigned char b)
{
  unsigned char const tmp = b<<1;
  return (b&0x80)?(tmp^0x1b):tmp;
}
```

(a) implementation with conditional branch

```
unsigned char xtime (unsigned char b)
{
  return (x<<1)^((x>>7)*0x1b);
}
```

(b) constant time implementation

Fig. 6: Implementations of xtime used in the MixColumns function of the AES

2.3 Data Reduction

Once the recorded data are realigned, if necessary, we have now access to the resynchronized data $X^{D,R,Q}$ and the associated PC vector Pc^D. Now that it is possible to analyze vertically the data, two questions arise. Are all the data in $X^{D,R,Q}$ relevant? And is it possible to reduce the dimension of the data to analyze? To answer these questions, we define the notion of activity matrix in *Def. 2* and of transition matrix *Def. 3*.

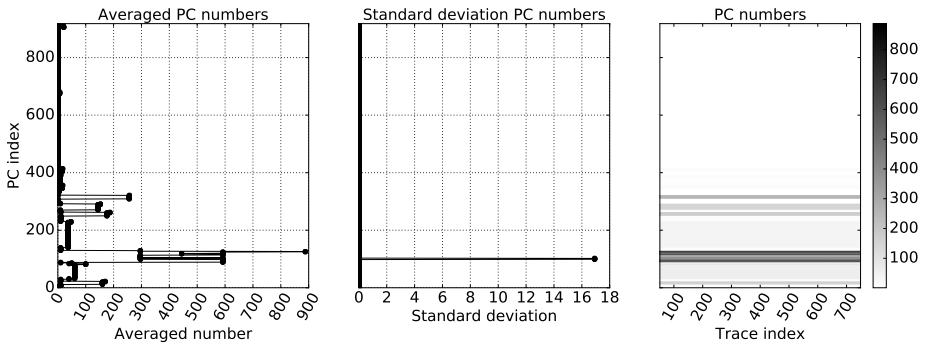

Fig. 7: Results of the application of the realignment algorithm to a masked implementation of an AES-128. The mean (on the left) and the standard deviation (in the middle) over Q of $\#\text{Pc}^{D',Q}$ (in the right) for thousand executions ($Q = 1000$) and around thousand distinct values of PC ($D' \simeq 1000$). The colors refer to the number of time each PC appear in each trace. This results have been obtained by executing a real-world masked implementation of an AES-128.

Definition 2. ***The activity matrix*** $A^{D,R}$ *of a given set* $X^{D,R,Q}$ *is defined as follow:*

$$A^{D,R} = \left[A_{d,r} = \begin{cases} 1, & \text{if } \sum_{q=0}^{Q-1} X_{d,r,q} \notin \{Q, 0\} \\ 0, & \text{otherwise} \end{cases} \right]_{\substack{d<D, \\ r<R}}$$

The activity matrix of a given dataset identifies the points ($(d, r) \in D \times R$) with a non-zero variance over Q.

Definition 3. ***The transition matrix*** $T^{D,R}$ *of a given set* $X^{D,R,Q}$ *is defined as follow:*

$$T^{D,R} = \left[T_{d,r} = \begin{cases} 1, & \text{if } d = 0 \\ \vee_{q=1}^{Q} X_{d-1,r} \oplus X_{d,r} & \text{otherwise} \end{cases} \right]_{\substack{d<D, \\ r<R}}$$

The transition matrix identifies the points ($(d, r) \in D \times R$) that change at least one time between the PC d and $d+1$ over all the traces. Both matrix $T^{D,R}$ and $A^{D,Q}$ could be computed in-line accumulating each trace. Then, we identify the points that could leak information as $L = \{(d, r) \in (D \times R) | T_{d,r} \wedge A_{d,r} = 1\}$. In *Fig. 8*, the matrices $A^{D,R}$, $T^{D,R}$ and $A^{D,R} \wedge T^{D,R}$ obtain analyzing a data set of 250 traces of execution of an AES WBC implementation. We observe that our algorithm permits to identify the 0.29% from the entire samples that could leak information. Thus we conserve only 20190 PC values over 28277 and 120 bit register over 1472. This data reduction speeds up the analysis and

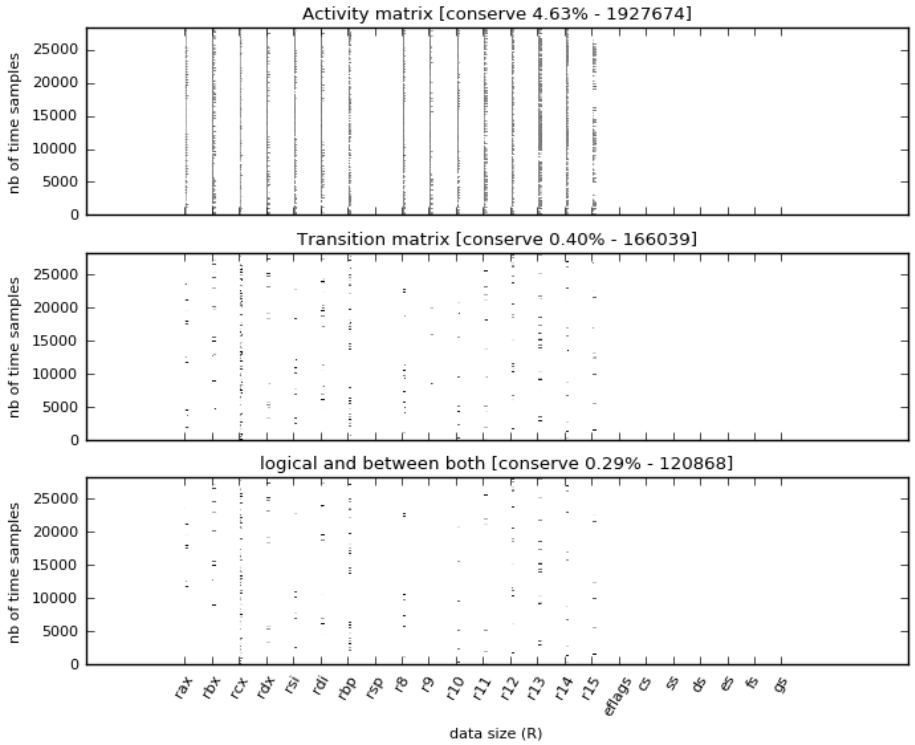

Fig. 8: Illustration of the matrices $A^{D,R}$ (on the top),$T^{D,R}$ (in the middle) and $A^{D,R} \wedge T^{D,R}$ (on the bottom). In white are displayed the 0 and the 1 in black.

reduce the memory footprint analyzing only the data $X^{L,Q}$. Furthermore we take advantage of the very low density of the sparse matrices $\{X_q^{D,R}\}_{q<Q}$ to reduce the storage required for the traces. The storage of sparse matrices is a well study problematic in computer science and a lot of solutions are freely provided. The following *Tab. 1* summarizes the gain in storage that we obtain using a method called *Compressed Sparse Column matrix (CSC)* present in *scipy*[5]. Thus, the needed memory to store the traces $X^{D,R,Q}$ decrease from $9,7$Go to 132Mo using the CSC compression on the matrix $T^{D,R,Q}$ defined in *Eq. 3*.

2.4 Distinguisher: CPA

The potential leakage points have been identified and stored in the dataset $X^{L,Q}$. To know if some of those points leak sensitive information, we use the CPA proposed by Brier *et al.* in 2004 [4]. In our case, the guessed intermediate values are stored in $Y^{S,Q,K,B}$ as explained in the *Eq. 1*. The CPA is based on the computation of the *Pearson coefficient* between $X^{L,Q}$ and $Y^{S,Q,K,B}$ to discriminate

[5] `https://www.scipy.org/`

	$\{(X_q^{D,R}\}_{q<Q}$ 8-bit matrix	$\{(X_q^{D,R}\}_{q<Q}$ CSC 8-bit matrix	$\{(T_q^{D,R}\}_{q<Q}$ CSC 8-bit matrix
size	9, 7Go	2, 6Go	132Mo

with

$$\{(T_q^{D,R}\}_{q<Q} = \left[\begin{cases} X_{d,r,q}, & \text{if } d = 0 \\ X_{d-1,r,q} \oplus X_{d,r,q} & \text{otherwise} \end{cases} \right]_{\substack{q<Q, r<R \\ d<D}} \tag{3}$$

Tab 1: Benchmark of the needed memory to store the traces $X^{D,R,Q}$ for $Q = 250$ using distinct formats.

the right key $\{k_b^\star\}_{b<B}$ as defined in the following *Eq. 4*.

$$\mathcal{D}(X^{L,Q}, Y^{S,Q,K,B}) = \left\{ \frac{\text{cov}(X^{L,Q}, Y_{s,k,b}^Q)}{\sigma(X^{L,Q})\sigma(Y_{s,k,b}^Q)} \right\}_{\substack{k<256, S<8 \\ b<16}}, \tag{4}$$

where σ is the variance and cov the covariance both over Q.

Then we identify the leaking samples using the *Absolute distinguishing margin* (*AbsMarg*) proposed by Whitnall *et al.* in [14] and recalled in the following *Eq. 5*. The usage of the *AbsMarg* metric is motivated by the presence of ghost peaks in the results of CPA.

$$\begin{aligned} &AbsMarg\ (\mathcal{D}(X^{L,Q}, Y_{s,b}^Q)) \\ &= \frac{\mathcal{D}(X^{L,Q}, Y_{s,k^\star,b}^Q) - \max_{\substack{k_b<256, \\ k\neq k_b^\star}} \{\mathcal{D}(X^{L,Q}, Y_{s,k,b}^Q)\}}{\underbrace{\cancel{\mathcal{D}(Y_{s,k^\star,b}^Q, Y_{s,k^\star,b}^Q)}}_{=1} - \max_{\substack{k_b<256, \\ k\neq k_b^\star}} \{\mathcal{D}(Y_{s,k^\star,b}^Q, Y_{s,k,b}^Q)\}} \end{aligned} \tag{5}$$

We take advantage that we analyze binary dataset to simplify the accumulative formula of the *Pearson coefficient* given in *Eq. 6*. This simplification speeds up the analysis and reduces the memory footprint.

$$\begin{aligned} &\mathcal{D}(X^{L,Q}, Y_{s,k,b}^Q) \\ &= \frac{\text{acc\_xy} - \frac{1}{Q}(\text{acc\_x.acc\_y})}{\sqrt{\text{acc\_xx} - \frac{1}{Q}(\text{acc\_x.acc\_x})}\sqrt{\text{acc\_yy} - \frac{1}{Q}(\text{acc\_y.acc\_y})}} \\ &= \frac{\text{acc\_xy} - \frac{1}{Q}(\text{acc\_x.acc\_y})}{\sqrt{\text{acc\_x} - \frac{1}{Q}(\text{acc\_x.acc\_x})}\sqrt{\text{acc\_y} - \frac{1}{Q}(\text{acc\_y.acc\_y})}} \end{aligned} \tag{6}$$

where we define the acc_$\star$ as follow,

$$\text{acc\_xy} = \sum_{q=1}^{Q} X_q^L . Y_{s,k,b,q}$$

$$\text{acc\_x} = \sum_{q=1}^{Q} X_q^L = \sum_{q=1}^{Q} (X_q^L)^2 = \text{acc\_xx}$$

$$\text{acc\_y} = \sum_{q=1}^{Q} Y_{s,k,b,q} = \sum_{q=1}^{Q} (Y_{s,k,b,q})^2 = \text{acc\_yy}$$

3 Results

In the following analyses, we use the known-plaintext attack model, which means that an attacker only requires access to the random inputs. Additionally, the attack could be performed with just the compiled binary. The source code access is only mandatory to map the leakages to the source code.

3.1 WBC Analysis

As presented in the previous *Subsec. 2.4* we use the CPA to reveal leakage from $X^{L,Q}$ and discriminate the secret key $\{k_b^\star\}_{b<16}$ from the guesses $\{k_b | k_b \neq k_b^\star\}_{b<16}$. As proposed in [1], we extend our leakage model Y to take into account the two products computed during the MixColumn execution. In [1] authors proposed to compute three distinct 8-bit Differential Power Analysis (DPA) while the two products only add ten new bit-distributions to the initial model. Indeed the model extension makes growing the model size S from 8 to 18, and not to 24, because 6 bit-distributions are redundant as resumed in the following *Proposition 1.*

Proposition 1. $\forall x \in \mathbb{N}$, *with* $x < 256$ *the products by two and three computed in the MixColumn function respect the following properties:*

- *the bit* 2 *of* $2.x$ *is equal to the bit* 1 *of* x
- *the bit* 5 *of* $2.x$ *is equal to the bit* 4 *of* x
- *the bit* 6 *of* $2.x$ *is equal to the bit* 5 *of* x
- *the bit* 7 *of* $2.x$ *is equal to the bit* 6 *of* x
- *the bit* 0 *of* $2.x$ *is equal to the bit* 7 *of* x
- *the bit* 0 *of* $3.x$ *is equal to the bit* 1 *of* $2.x$

The *Proposition 1* permits to construct an extended model expressed in the following *formula 7.*

The *Fig. 9* illustrates the results obtained by applying the CPA on $Q = 250$ recorded traces from the WBC implementation provided at the Capture The Flag (CTF) challenge of CHES-2016. The CPA has been computed for all samples L (120k) but to facilitate the visualization we focus on the first 20k samples

$$Y^{S,Q,K,B} =$$

$$\left[\begin{cases} Sbox\ (P_{q,b} \oplus k)\ \&\ 2^{s}, & \text{if } s < 8 \\ (2.Sbox\ (P_{q,b} \oplus k))\ \&\ 2^{v[s-8]}, & \text{if } 7 < s < 11, \text{ with } v = \{1,3,4\} \\ (3.Sbox\ (P_{q,b} \oplus k))\ \&\ 2^{s-10}, & \text{if } 10 < s \end{cases} \right]_{\substack{s<18, k<256, \\ q<Q, b<16}} \tag{7}$$

where the leaking ones are localized. In grey we display the results obtained with the bad guesses: $\left\{ \max_{k_b<256, k_b \neq k_b^\star} (\mathcal{D}(X^{L,Q}, Y^{Q}_{s,k,b})) \right\}_{b<16, s<18}$ and in color the result obtained with the right key: $\left\{ \mathcal{D}(X^{L,Q}, Y^{Q}_{s,k^\star,b}) \right\}_{b<16, s<18}$. As first result we notice that all key byte are recovered. It is a another powerful advantage of our analysis that lead us to easily and quickly recover the secret of a WBC implementation. To improve the discrimination of the leaking samples from the

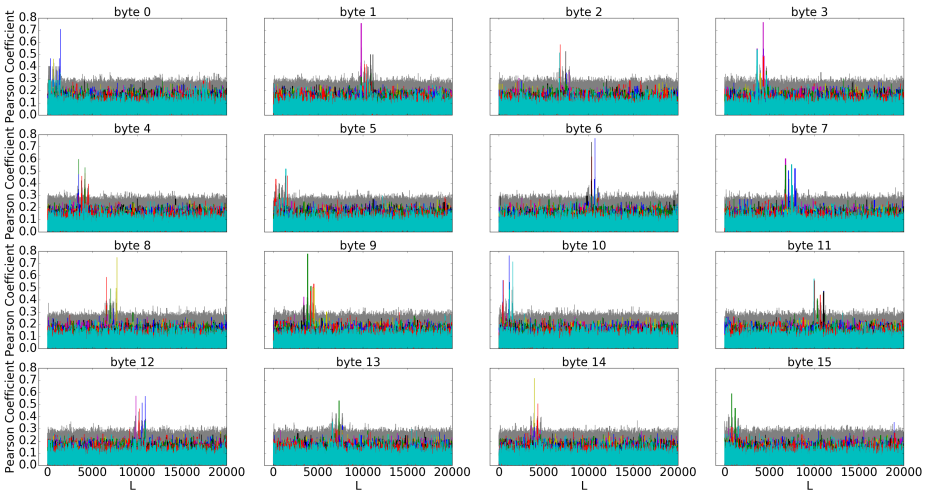

Fig. 9: Results of the CPA on recorded data from a WBC implementation.

computed CPA we use the *AbsMarg* displayed in *Fig 10*, in which we apply a threshold at 0.25. Then, using the PC values we map the identified leaking samples to the source code. We summarize the obtained results in the two *Tab 1* provided in the appendix in which we link each leakage to a line code, a leaking bit model, and bit register.

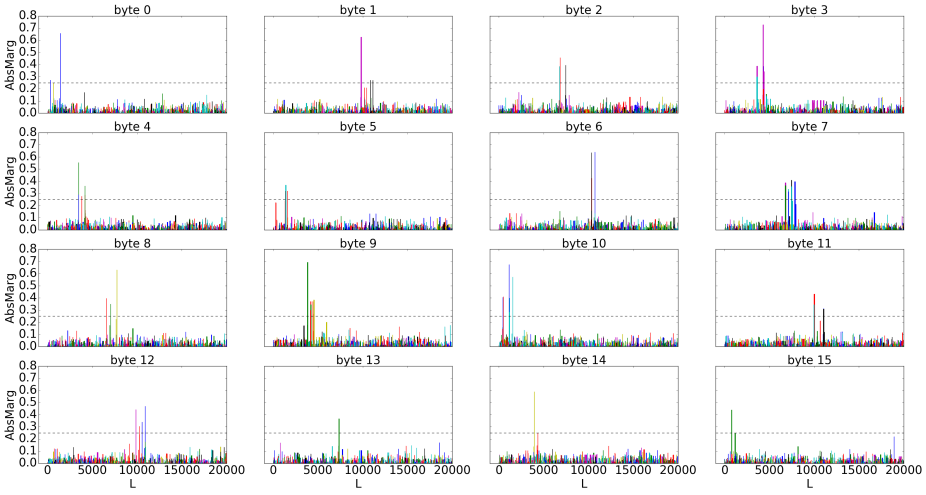

Fig. 10: AbsMarg of the CPA's results. Only the samples with a positive *AbsMarg* samples are plotted.

4 Conclusion

We present in this paper a new methodology of SC evaluation for software implementation. We improve the state of the art in that field by providing a practical and effective methodology to extract all the data that will be manipulated by a software implementation during its execution. All the recorded data are analyzed independently, at bit level, without any arbitrary leakage model applied to generate traces as in all the SC-simulators presented in the state of the art (as far as we know). These features give to our solution exhaustive properties and suppress the noise that a leakage model could generate. Our exhaustive approach makes it agnostic for the target hardware by focusing the analysis on the manipulated data and not on hardware characteristics. Furthermore, our methodology allows to map, in the time and in the space, the leakage of sensitive information to the source code, and this can be of significant help for an evaluator or a developer. Advantageously, we provide two additional new methods to support and improve the SCA assessment. First, we describe an efficient resynchronization algorithm based on the control flow values. Second, we give a methodology of samples selection to significantly decrease the number of samples to analyze without any loss of information. Both features are crucial when analyzing complex and/or massive software implementations.

Acknowledgments

This work was partly supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT)

(No. 2016-0-00399, Study on secure key hiding technology for IoT devices [KeyHAS Project]) and other project(s).

Acknowledgments

This work was partly supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2016-0-00399, Study on secure key hiding technology for IoT devices [KeyHAS Project]).

References

1. Hyunjin Ahn and Dong-Guk Han. Multilateral white-box cryptanalysis: Case study on WB-AES of CHES challenge 2016. *IACR Cryptology ePrint Archive*, 2016:807, 2016.
2. Julien Allibert, Benoit Feix, Georges Gagnerot, Ismael Kane, Hugues Thiebeauld, and Tiana Razafindralambo. Chicken or the egg - computational data attacks or physical attacks. *IACR Cryptology ePrint Archive*, 2015:1086, 2015.
3. Joppe W. Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. Differential computation analysis: Hiding your white-box designs is not enough. In *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 215–236, 2016.
4. Eric Brier, Christophe Clavier, and Francis 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*, pages 16–29, 2004.
5. Nicolas Debande, Maël Berthier, Yves Bocktaels, and Thanh-Ha Le. Profiled model based power simulator for side channel evaluation. *IACR Cryptology ePrint Archive*, 2012:703, 2012.
6. Nicolas Debande, Youssef Souissi, Maxime Nassar, Sylvain Guilley, Thanh-Ha Le, and Jean-Luc Danger. "re-synchronization by moments": An efficient solution to align side-channel traces. In *2011 IEEE International Workshop on Information Forensics and Security, WIFS 2011, Iguacu Falls, Brazil, November 29 - December 2, 2011*, pages 1–6, 2011.
7. Julien Doget, Emmanuel Prouff, Matthieu Rivain, and François-Xavier Standaert. Univariate side channel attacks and leakage modeling. *J. Cryptographic Engineering*, 1(2):123–144, 2011.
8. Sylvain Guilley, Karim Khalfallah, Victor Lomné, and Jean-Luc Danger. Formal framework for the evaluation of waveform resynchronization algorithms. In *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication - 5th IFIP WG 11.2 International Workshop, WISTP 2011, Heraklion, Crete, Greece, June 1-3, 2011. Proceedings*, pages 100–115, 2011.
9. David McCann, Carolyn Whitnall, and Elisabeth Oswald. ELMO: emulating leaks for the ARM cortex-m0 without access to a side channel lab. *IACR Cryptology ePrint Archive*, 2016:517, 2016.
10. Hugues Thiebeauld, Georges Gagnerot, Antoine Wurcker, and Christophe Clavier. Scatter : A new dimension in side-channel. Cryptology ePrint Archive, Report 2017/706, 2017. `https://eprint.iacr.org/2017/706`.

11. Céline Thuillet, Philippe Andouard, and Olivier Ly. A smart card power analysis simulator. In *Proceedings of the 12th IEEE International Conference on Computational Science and Engineering, CSE 2009, Vancouver, BC, Canada, August 29-31, 2009*, pages 847–852, 2009.
12. Jasper G. J. van Woudenberg, Marc F. Witteman, and Bram Bakker. Improving differential power analysis by elastic alignment. In *Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*, pages 104–119, 2011.
13. Nikita Veshchikov. SILK: high level of abstraction leakage simulator for side channel analysis. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC 2014, New Orleans, LA, USA, December 9, 2014*, pages 3:1–3:11, 2014.
14. Carolyn Whitnall and Elisabeth Oswald. A fair evaluation framework for comparing side-channel distinguishers. *J. Cryptographic Engineering*, 1(2):145–160, 2011.

Appendix

line source code	reg. name	bit reg.	leaking value	leaking bit	key byte
l.4086: v16 = lookup_nibble2(table_4436, v16, v18, 0);	rsi	1	x	7	0
l.4128: v22 = lookup_nibble2(table_4499, v22, v23, 0);	rbp	0	3.x	4	0
l.4420: v18 = lookup_nibble(table_13890, v4);	r14	3	x	4	1
l.4417: v2 = lookup_nibble2(table_4934, v16, v2, 0);	r15	1	3.x	1	1
l.4417: v2 = lookup_nibble2(table_4934, v16, v2, 0);	r13	3	x	4	1
l.4421: v19 = lookup_nibble(table_13891, v4);	r13	2	x	4	1
l.4422: v16 = lookup_nibble2(table_4940, v16, v18, 0);	rcx	2	x	4	1
l.4421: v19 = lookup_nibble(table_13891, v4);	r13	1	3.x	1	1
l.4421: v19 = lookup_nibble(table_13891, v4);	r13	0	3.x	1	1
l.4418: v16 = lookup_nibble(table_13888, v3);	rdx	0	3.x	1	1
l.4463: v4 = lookup_nibble(table_13915, v4);	r9	0	3.x	3	1
l.4462: v35 = lookup_nibble(table_13914, v4);	rdi	0	3.x	3	1
l.4464: v34 = lookup_nibble2(table_5003, v34, v35, 0);	rcx	2	3.x	3	1
l.4315: v21 = lookup_nibble(table_13815, v13);	rbx	0	3.x	7	2
l.4316: v18 = lookup_nibble2(table_4779, v18, v20, 0);	r13	1	3.x	6	2
l.4343: v33 = lookup_nibble(table_13831, v13);	r12	2	3.x	3	2
l.4343: v33 = lookup_nibble(table_13831, v13);	r12	3	3.x	3	2
l.4199: v17 = lookup_nibble2(table_4605, v17, v19, 0);	r10	0	3.x	7	3
l.4199: v17 = lookup_nibble2(table_4605, v17, v19, 0);	r10	1	x	4	3
l.4199: v17 = lookup_nibble2(table_4605, v17, v19, 0);	r10	3	3.x	7	3
l.4200: v18 = lookup_nibble(table_13732, v11);	rbx	0	3.x	7	3
l.4200: v18 = lookup_nibble(table_13732, v11);	rbx	1	x	4	3
l.4200: v18 = lookup_nibble(table_13732, v11);	rbx	3	3.x	7	3
l.4204: v18 = lookup_nibble2(table_4611, v18, v20, 0);	rbx	0	x	4	3
l.4204: v18 = lookup_nibble2(table_4611, v18, v20, 0);	rbx	2	3.x	7	3

l.4204: v18 = lookup_nibble2(table_4611, v18, v20, 0);	rcx	0	x	4	3
l.4204: v18 = lookup_nibble2(table_4611, v18, v20, 0);	rcx	2	3.x	7	3
l.4204: v18 = lookup_nibble2(table_4611, v18, v20, 0);	rcx	2	3.x	7	3
l.4230: v28 = lookup_nibble(table_13750, v12);	rbp	0	x	4	3
l.4227: v21 = lookup_nibble2(table_4647, v21, v27, 0);	rbx	2	x	0	3
l.4231: v29 = lookup_nibble(table_13751, v12);	rcx	0	x	4	3
l.4232: v26 = lookup_nibble2(table_4653, v26, v28, 0);	rcx	2	x	4	3
l.4232: v26 = lookup_nibble2(table_4653, v26, v28, 0);	rbp	2	x	0	3
l.4232: v26 = lookup_nibble2(table_4653, v26, v28, 0);	rbp	1	x	0	3
l.4228: v26 = lookup_nibble(table_13748, v11);	r12	4	x	4	3
l.4228: v26 = lookup_nibble(table_13748, v11);	r12	5	x	4	3
l.4228: v26 = lookup_nibble(table_13748, v11);	r12	6	x	4	3
l.4228: v26 = lookup_nibble(table_13748, v11);	r12	7	x	4	3
l.4232: v26 = lookup_nibble2(table_4653, v26, v28, 0);	rbx	6	x	4	3
l.4233: v27 = lookup_nibble2(table_4654, v27, v29, 0);	rcx	3	x	4	3
l.4233: v27 = lookup_nibble2(table_4654, v27, v29, 0);	rcx	4	x	4	3
l.4233: v27 = lookup_nibble2(table_4654, v27, v29, 0);	rcx	5	x	4	3
l.4233: v27 = lookup_nibble2(table_4654, v27, v29, 0);	rcx	6	x	4	3
l.4198: v16 = lookup_nibble2(table_4604, v16, v18, 0);	rax	1	x	1	4
l.4198: v16 = lookup_nibble2(table_4604, v16, v18, 0);	rax	1	3.x	4	4
l.4198: v16 = lookup_nibble2(table_4604, v16, v18, 0);	rax	3	x	1	4
l.4211: v21 = lookup_nibble(table_13739, v6);	rbp	3	3.x	6	4
l.4225: v27 = lookup_nibble(table_13747, v6);	rbp	2	3.x	5	4
l.4225: v27 = lookup_nibble(table_13747, v6);	rbp	3	2.x	3	4
l.4127: v5 = lookup_nibble(table_13675, v5);	r9	0	2.x	3	5
l.4126: v23 = lookup_nibble(table_13674, v5);	rdi	0	2.x	3	5
l.4123: v21 = lookup_nibble2(table_4493, v21, v23, 0);	r8	2	3.x	7	5
l.4127: v5 = lookup_nibble(table_13675, v5);	r14	2	3.x	7	5
l.4127: v5 = lookup_nibble(table_13675, v5);	r14	1	3.x	7	5
l.4128: v22 = lookup_nibble2(table_4499, v22, v23, 0);	rcx	1	3.x	7	5
l.4127: v5 = lookup_nibble(table_13675, v5);	rcx	2	2.x	3	5
l.4441: v35 = lookup_nibble(table_13903, v14);	r12	2	3.x	3	6
l.4442: v20 = lookup_nibble2(table_4968, v20, v34, 0);	r15	1	2.x	3	6
l.4442: v20 = lookup_nibble2(table_4968, v20, v34, 0);	r15	3	2.x	3	6
l.4456: v34 = lookup_nibble2(table_4989, v34, v36, 0);	r15	3	3.x	4	6
l.4311: v17 = lookup_nibble2(table_4773, v17, v19, 0);	r10	1	3.x	5	7
l.4311: v17 = lookup_nibble2(table_4773, v17, v19, 0);	r10	2	3.x	1	7
l.4312: v18 = lookup_nibble(table_13812, v8);	rbx	1	3.x	5	7
l.4312: v18 = lookup_nibble(table_13812, v8);	rbx	2	3.x	1	7
l.4316: v18 = lookup_nibble2(table_4779, v18, v20, 0);	rbx	0	3.x	5	7
l.4316: v18 = lookup_nibble2(table_4779, v18, v20, 0);	rbx	1	3.x	1	7
l.4316: v18 = lookup_nibble2(table_4779, v18, v20, 0);	rcx	0	3.x	5	7
l.4316: v18 = lookup_nibble2(table_4779, v18, v20, 0);	rcx	1	3.x	1	7
l.4328: v30 = lookup_nibble(table_13822, v13);	rbp	3	x	3	7
l.4328: v30 = lookup_nibble(table_13822, v13);	rbp	3	x	7	7

l.4329: v31 = lookup_nibble(table_13823, v13);	rcx	3	x	3	7
l.4329: v31 = lookup_nibble(table_13823, v13);	rcx	3	x	7	7
l.4329: v31 = lookup_nibble(table_13823, v13);	rcx	2	x	3	7
l.4329: v31 = lookup_nibble(table_13823, v13);	rcx	2	x	7	7
l.4342: v32 = lookup_nibble(table_13830, v13);	rbp	0	3.x	7	7
l.4342: v32 = lookup_nibble(table_13830, v13);	rbp	1	3.x	3	7
l.4343: v33 = lookup_nibble(table_13831, v13);	rcx	0	3.x	7	7
l.4343: v33 = lookup_nibble(table_13831, v13);	rcx	1	3.x	3	7
l.4343: v33 = lookup_nibble(table_13831, v13);	rcx	0	3.x	3	7
l.4344: v30 = lookup_nibble2(table_4821, v30, v32, 0);	rcx	2	3.x	7	7
l.4357: v13 = lookup_nibble(table_13839, v13);	r14	0	3.x	4	7
l.4358: v7 = lookup_nibble2(table_4842, v7, v31, 0);	r13	0	3.x	4	7
l.4359: v8 = lookup_nibble2(table_4843, v8, v13, 0);	rcx	2	3.x	4	7
l.4309: v19 = lookup_nibble(table_13811, v7);	rdx	0	x	2	8
l.4324: v18 = lookup_nibble2(table_4793, v18, v20, 0);	rbx	0	x	1	8
l.4352: v30 = lookup_nibble2(table_4835, v30, v31, 0);	rbp	0	3.x	2	8
l.4352: v30 = lookup_nibble2(table_4835, v30, v31, 0);	rbp	3	3.x	2	8
l.4207: v17 = lookup_nibble2(table_4619, v17, v19, 0);	r10	2	3.x	5	9
l.4208: v18 = lookup_nibble(table_13736, v1);	rbp	2	3.x	5	9
l.4208: v18 = lookup_nibble(table_13736, v1);	rbp	1	3.x	5	9
l.4212: v18 = lookup_nibble2(table_4625, v18, v20, 0);	rcx	1	3.x	5	9
l.4225: v27 = lookup_nibble(table_13747, v6);	r11	2	3.x	6	9
l.4224: v26 = lookup_nibble(table_13746, v6);	r14	2	3.x	6	9
l.4221: v19 = lookup_nibble2(table_4640, v19, v21, 0);	r10	0	2.x	3	9
l.4221: v19 = lookup_nibble2(table_4640, v19, v21, 0);	r10	2	3.x	5	9
l.4221: v19 = lookup_nibble2(table_4640, v19, v21, 0);	r14	1	3.x	6	9
l.4225: v27 = lookup_nibble(table_13747, v6);	rcx	1	3.x	6	9
l.4222: v20 = lookup_nibble(table_13744, v1);	rbp	0	2.x	3	9
l.4222: v20 = lookup_nibble(table_13744, v1);	rbp	2	3.x	5	9
l.4225: v27 = lookup_nibble(table_13747, v6);	rbp	1	3.x	5	9
l.4226: v20 = lookup_nibble2(table_4646, v20, v26, 0);	rcx	1	3.x	5	9
l.4234: v20 = lookup_nibble2(table_4660, v20, v26, 0);	rcx	2	2.x	3	9
l.4238: v27 = lookup_nibble(table_13754, v6);	r9	3	2.x	3	9
l.4239: v6 = lookup_nibble(table_13755, v6);	r8	2	3.x	2	9
l.4235: v21 = lookup_nibble2(table_4661, v21, v27, 0);	rdi	3	2.x	3	9
l.4240: v26 = lookup_nibble2(table_4667, v26, v27, 0);	rdi	2	2.x	3	9
l.4239: v6 = lookup_nibble(table_13755, v6);	r15	2	3.x	2	9
l.4239: v6 = lookup_nibble(table_13755, v6);	rcx	2	2.x	3	9
l.4239: v6 = lookup_nibble(table_13755, v6);	r15	1	3.x	2	9
l.4241: v1 = lookup_nibble2(table_4668, v1, v6, 0);	rcx	1	3.x	2	9
l.4091: v21 = lookup_nibble(table_13655, v15);	rbx	1	x	7	10
l.4092: v18 = lookup_nibble2(table_4443, v18, v20, 0);	r13	1	3.x	6	10
l.4119: v25 = lookup_nibble(table_13671, v15);	r12	0	x	3	10
l.4119: v25 = lookup_nibble(table_13671, v15);	r12	1	x	0	10
l.4120: v22 = lookup_nibble2(table_4485, v22, v24, 0);	r15	0	2.x	4	10

l.4134: v5 = lookup_nibble2(table_4506, v5, v23, 0);	rbp	1	x	3	10
l.4426: v20 = lookup_nibble(table_13894, v14);	r11	3	3.x	7	11
l.4423: v17 = lookup_nibble2(table_4941, v17, v19, 0);	r10	2	2.x	3	11
l.4427: v21 = lookup_nibble(table_13895, v14);	rcx	3	3.x	7	11
l.4427: v21 = lookup_nibble(table_13895, v14);	rcx	2	3.x	7	11
l.4424: v18 = lookup_nibble(table_13892, v9);	rbx	2	2.x	3	11
l.4428: v18 = lookup_nibble2(table_4947, v18, v20, 0);	rbx	1	2.x	3	11
l.4428: v18 = lookup_nibble2(table_4947, v18, v20, 0);	rcx	1	2.x	3	11
l.4465: v3 = lookup_nibble2(table_5004, v3, v4, 0);	r14	3	x	0	11
l.4465: v3 = lookup_nibble2(table_5004, v3, v4, 0);	r14	3	3.x	3	11
l.4470: v4 = lookup_nibble2(table_5010, v4, v35, 0);	rsi	3	x	0	11
l.4470: v4 = lookup_nibble2(table_5010, v4, v35, 0);	rsi	3	3.x	3	11
l.4470: v4 = lookup_nibble2(table_5010, v4, v35, 0);	rsi	2	x	0	11
l.4470: v4 = lookup_nibble2(table_5010, v4, v35, 0);	rsi	2	3.x	3	11
l.4470: v4 = lookup_nibble2(table_5010, v4, v35, 0);	rax	2	x	0	11
l.4470: v4 = lookup_nibble2(table_5010, v4, v35, 0);	rax	2	3.x	3	11
l.4425: v19 = lookup_nibble(table_13893, v9);	rax	1	3.x	1	12
l.4436: v18 = lookup_nibble2(table_4961, v18, v20, 0);	rbx	0	x	2	12
l.4450: v20 = lookup_nibble2(table_4982, v20, v34, 0);	rbp	3	x	7	12
l.4463: v4 = lookup_nibble(table_13915, v4);	r10	0	3.x	4	12
l.4463: v4 = lookup_nibble(table_13915, v4);	r10	1	x	7	12
l.4337: v31 = lookup_nibble(table_13827, v7);	r11	2	2.x	1	13
l.4336: v30 = lookup_nibble(table_13826, v7);	r14	2	2.x	1	13
l.4333: v19 = lookup_nibble2(table_4808, v19, v21, 0);	r14	1	2.x	1	13
l.4337: v31 = lookup_nibble(table_13827, v7);	rcx	1	2.x	1	13
l.4217: v27 = lookup_nibble(table_13743, v12);	r12	1	x	5	14
l.4104: v22 = lookup_nibble(table_13662, v15);	rbp	1	3.x	5	15
l.4105: v23 = lookup_nibble(table_13663, v15);	rcx	1	3.x	5	15
l.4105: v23 = lookup_nibble(table_13663, v15);	rcx	0	3.x	5	15

Table 1: Leakage characterization and mapping to the source code