Generic SCARE: reverse engineering without knowing the algorithm nor the machine

Ronan Lashermes[1] and Hélène Le Bouder[2]

[1] Inria/SED&LHS
[2] IMT-Atlantique, IRISA, OCIF team

Keywords: Side channels · reverse engineering · SCARE · RISC

Abstract. We introduce a novel side-channel-based reverse engineering technique capable of reconstructing a procedure solely from inputs, outputs, and traces of execution. Beyond generic restrictions, we do not assume any prior knowledge of the procedure or the chip it operates on. These restrictions confine our analysis to 8-bit RISC constant-time software implementations.
Specifically we demonstrate with simulated traces the theoretical feasibility of reconstructing a symmetric cryptographic cipher, even in scenarios where traces are sampled with information loss and noise, such as when measuring the power consumption of the chip.

1 Introduction

Side-channel analysis (SCA) is a widely used technique for recovering secrets from a running chip. Typically, the attacker records a leaking signal, called a trace (e.g. computation timing, power consumption, electromagnetic emissions, etc.), in conjunction with the inputs or outputs. Then by using a predetermined model (CPA-like attacks), or by learning the model (template attacks, deep learning attacks), the attacker can tie the leakage to the coveted secret and recover it.

Drawing from the same principles, side-channel analysis for reverse engineering (SCARE) techniques leverage side-channel leakages to reconstruct a portion of an algorithm rather than a secret key [15]. These techniques invariably commence with specific information about the target and typically aim to recover a fragment of an otherwise well-understood algorithm.

Threat model In this paper, we propose an attack that capitalizes on these cryptanalysis techniques to reverse engineer an entire procedure using only its inputs, outputs, and execution traces. Notably, the traces can be lossy and noisy, such as the power consumption of a chip. The attacker is not privy to any other information about the procedure or the platform it executes on, barring some generic restrictions that are detailed in section 3. Fundamentally, our technique is confined to RISC software implementations, 8-bit algorithms, and constant-time procedures.

Applications While the technique is not specifically aimed at symmetric cryptographic primitives, it is particularly useful in these scenarios for an attacker. Generic SCARE enables the replication of a cipher that is observed during execution. Several applications are possible:

Forging encrypted data: If a cryptographic protocol is poorly designed, lacking proper authentication, forging encrypted data can allow an adversary to inject their own information into the system. A man-in-the-middle attack could be launched using Generic SCARE by observing the sender's encryption process and then replacing the encrypted data with the forged ones.

Decrypting data with specific modes of operation: Some modes of operation, especially counter-based ones such as CTR, use the same cipher for encryption and decryption. Therefore, depending on the specific protocol, it may be possible to use the cipher learnt with Generic SCARE to decrypt data that have been encrypted with the same cipher and key.

Creating Message Authentication Codes: A message authentication code (MAC) is a procedure used to verify the integrity of a message for trusted recipients. As both the signing and verifying parties share the same secret key, it provides a weak form of authentication for the message's origin. In certain circumstances, it may be possible to clone a MAC function using Generic SCARE. For example, in the case of the Pelican MAC function [7], if the attacker is able to observe the execution of the Pelican MAC multiple times for messages of the same size, it becomes possible to forge a MAC for a new message.

Obtaining decryption from the observation of encryptions: Similarly to how CTR modes are particularly vulnerable, certain cipher structures may enhance the potency of this technique. In particular, the decryption procedures of Feistel ciphers could be inferred by observing the encryption procedure. However, initial experiments indicated that our method would need to be modified to properly deal with reordering.

Motivations This paper aims to probe the boundaries of SCA: what information can be extracted from traces? Under certain conditions, we demonstrate that an unknown algorithm, even when executed on an unidentified machine, can be comprehensively reverse-engineered using noisy and lossy execution traces such as power consumption. This underscores the importance of understanding what can be inferred from seemingly innocuous data leakages.

The conditions required for the efficacy of our technique are critical: by conscientiously implementing algorithms in a manner that avoids satisfying these conditions, greater resistance can be achieved. Nonetheless, our results will probably be improved with time by relaxing these conditions and points in one direction: security-related functionalities must not be executed on an insecure substrate of computation. There is an imperative need for generic processors with robust protections against SCA.

Reiterating the significance of Kerckhoffs's principles, we furnish yet another argument illustrating that security through obscurity can be overcome.

Contributions Our SCARE technique only relies on the leakage of the data in the targeted procedure, depending on intermediate values, in addition to input and output data. Our main contributions are: we introduce the information flow hypergraph (IFH) as a tool to measure and analyse information flows in a trace. We show that it is possible to reconstruct information lost to sampling by following information propagation in the IFH. We demonstrate that the technique works in presence of Gaussian noise.

The source code demonstrating our analysis is available at https://gitlab.inria.fr/bbscare.

Our results are primarily theoretical and would necessitate significant adaptation to be applied to real chips. However, with this paper, we extend the boundaries of what is considered achievable through side channels.

Organization section 2 presents a concise state of the art to provide context. Definitions of trace and machine requirements are outlined in section 3.

We delineate the methodology for reverse engineering of procedures under three distinct settings: **Perfect** (section 4): the traces contain all intermediate values of the procedure, without any noise. This idealized scenario serves as a foundation for introducing our methodology and notation. **Lossy** (section 5): the intermediate values are sampled with a loss of information but without noise. In our examples, we utilize the Hamming weights of the data as opposed to the data itself. **Noisy and lossy** (section 6): the intermediate values are sampled with both information loss (again, Hamming weights in our examples) and Gaussian noise.

Our proposal is theoretical, and we discuss in section 7 the challenges an attacker might encounter when attempting to apply Generic SCARE to a real-world device.

To validate our techniques, we implement them to reverse engineer simulations of AES-128 and Twine symmetric cryptographic ciphers. Throughout this paper, we reference these experiments to complement the abstract formalization. Results specifically pertinent to our examples are distinctly highlighted in a double-framed box, as illustrated by this text.

Subsequently, we suggest countermeasures in section 8. The paper concludes with a summary of key findings in section 9.

2 Related Works

Blind side channels have been previously explored [5,12,13,14], wherein neither inputs nor outputs are utilized to recover a cryptographic key; rather, only the traces and knowledge of the algorithm are employed. Blind side channels operate by identifying correlations across different time points within the traces, which are parameterized by a segment of the key. In this regard, there are notable similarities between Generic SCARE and blind side channels.

SCARE is a technique that employs side channels with the knowledge of inputs and outputs to acquire insights into the inner workings of a procedure, typically a symmetric cipher. The technique was first introduced by Novak [15] and later refined by Clavier [3], who employed side channels to extract a concealed portion of the A3/A8 algorithm. Utilizing this technique, it becomes feasible to uncover confidential components of an algorithm, assuming the rest is known, such as permutation tables for DES [8] or AES S-Boxes [11].

Prior research has demonstrated the viability of applying SCARE to extract segments of generic cryptographic structures. Feistel networks [16], substitution-permutation networks (SPNs) [17], and stream ciphers [10] have all been shown to be susceptible to this technique.

In a noteworthy study by Clavier *et al.* [4], the authors successfully demonstrate the feasibility of employing SCARE to recover an entire AES-like algorithm.

Our work represents a generalization of these earlier analyses, as we target any software constant-time procedures without knowledge of the internal algorithmic structure.

Another avenue within SCARE involves side-channel-based disassemblers that aim to recover executed instructions. By analysing traces, these approaches seek to identify the instructions and data processed by the chip. Eisenbarth *et al.* [9] constructed a leakage model for all instructions and compared the actual leakage to these models. Utilizing this method, they were able to recover portions of the procedure's software.

In a study by Cristiani *et al.* [6], up to 95% recovery of the full 14-bit instruction was demonstrated by exploiting the leakage of all bits individually.

While this line of research excels in recovering control flow, our focus is on the data flow. Our goal is to construct a new procedure that retains the same functionality albeit not necessarily with an identical control flow. Solely knowing the control flow is insufficient for reconstructing an unknown procedure.

Table 1. Classification of related presented techniques.

Technique	Knowledge			Attack target
	I/O	Traces	Procedure	
Blind SCA	no	yes	known	cryptographic key
SCA disassembler	no	yes	ISA known	instruction opcodes
SCARE	yes	yes	partial knowledge	part of procedure
Generic SCARE	yes	yes	generic properties	complete procedure

Our Generic SCARE technique does not focus on a specific algorithmic structure but instead operates under the assumption of a generic one. As elaborated in section 3: the software is executed on a RISC core and consists exclusively of elementary functions with either one or two inputs. As a result, our approach is distinctly different from other SCARE techniques and does not lend itself to

direct success rates comparisons, as we are targeting different aspects for reverse engineering as illustrated on Table 1.

3 Traces and machines

Our technique requires traces of execution, in conjunction with inputs and outputs, as the primary data for information recovery. This section elaborates on the characteristics that define a valid trace, as well as the methods employed to obtain them.

3.1 Traces requirements

In this paper, a trace is defined as a byte array that captures information pertaining to the procedure an attacker seeks to analyse. A trace may contain all register values at each point in time during the procedure's execution, or it may consist of sampled measurements of the chip's power consumption while the procedure executes. For Generic SCARE to operate effectively, the traces must satisfy certain constraints.

8-bit RISC software procedure The attacker can only deal with 8-bit RISC software implementations. It means that only 1-to-1 or 2-to-1 elementary operations are allowed with one input-one output or two inputs-one output, operating on 8-bit data. Equivalently, one bit of data is totally determined by at most 16 previous bits of data. This property is a property of the procedure, not necessarily of the hardware it is executed on: in our experiments, we are using a 8-bit software implementation on a standard 32-bit RISC-V hardware.

Various symmetric ciphers have been designed targeting low-cost 8-bit microcontrollers (e.g., AES and Twine), with functions operating on 8-bit variables. These ciphers often remain vulnerable to our technique, even when executed on 32-bit targets.

Temporal causality In one trace, the information is present in chronological order. A byte value at time t_0 can be determined by byte values at times $t < t_0$.

Synchronicity For the analysis to be carried out, the procedure under test must be executed repeatedly. Traces must be synchronous, meaning that at the same time index within the trace, the byte value must correspond to the same sub-operation of the underlying procedure. There should be no jitter, and traces must not be shuffled. This constraint necessitates a perfectly constant-time implementation of the procedure. In scenarios with noise, this constraint can be somewhat relaxed: jitter can be modelled as additional noise, in addition to usage of trace resynchronization techniques.

Completeness The traces must not miss any data on the underlying procedure: all elementary 1-to-1 and 2-to-1 operations must be captured by the trace. Since power consumption can usually be measured with a sampling rate higher than the clock frequency, this requirement is easily fulfilled.

Information content The traces must contain enough information to recover the procedure. If there are not enough traces, or if the sampling function loses too much information with respect to the specific implementation, it may not be possible to succeed. A more precise analysis of this condition is proposed in subsection 5.2, after having introduced the necessary notions.

Noise model We consider in section 6 that the noise follows a Gaussian distribution and is independent and identically distributed (iid), as usually assumed in SCA noise models since [2]. The model shortcomings when it comes to real-world applicability are discussed in section 7.

3.2 Targeted Processor

Based on the constraints imposed on the traces, we can infer the characteristics of the machine from which these traces originate. Bytes are processed sequentially, and only 1-to-1 and 2-to-1 operations are permissible, which is typical for an 8-bit RISC processor or an 8-bit procedure running on any RISC processor.

Cryptographic hardware coprocessors generally process multiple bytes in parallel; our technique is not applicable to them if byte-by-byte leakage extraction is not possible. For the same reason, implementations using single instruction, multiple data (SIMD) are outside the scope of our technique.

Traces can be generated through probing, such as by reading byte values in the write-back stage of the pipeline. Alternatively, they can be the measured power consumption of a chip or obtained through an EM probe.

3.3 Setup

The reverse engineering is done on a test machine with a laptops Intel i7-12700H CPU, 32GB of RAM and 1TB of disk space. Data is generated by a RISC-V (RV32I) emulator that records all destination register values. Two symmetric encryption algorithms have been tested: AES-128 and Twine, in an 8-bit software constant-time implementation realized in assembly. Traces are generated by storing all the destination register values for each execution. This register holds the result of the instruction if there is one, simulating the interception of the output register during write-back. For example, for the instruction `add x1, x2, x3` that adds the values in `x2` and `x3` and puts the resulting value in `x1`, we only save this last value into the trace. The lossy and noisy scenarios are obtained by transforming these value: using the Hamming weight of the values instead of the value for the lossy scenario, using the Hamming weight and adding a Gaussian noise for the noisy scenario.

4 Perfect sampling: noiseless, lossless

In this section, the attacker learns a procedure in perfect conditions : the traces are complete, sampling is lossless and noiseless.

> In our experiments, the traces contain all the destination register values for each instruction that writes to one.

4.1 Generalities and notation

Data generation The aim is to construct a function $\mathcal{F}$ that accurately reproduces the output of the targeted unknown procedure $\mathcal{P}$ for any given input. The attacker accomplishes this by using traces: she systematically records trace and output data across multiple executions of the same procedure $\mathcal{P}$, while varying input data. In the data generation phase, the procedure is executed E times, with inputs of size S_I bytes, outputs of size S_O bytes, and trace lengths of S_T bytes. All measured data are sampled, yielding only values as unsigned bytes: $\mathbb{B} = [\![0, 255]\!]$.
These experiments give us three matrices: I is the input matrix, where $(I_{e,i}) \in \mathbb{B}^{E \times S_I}$. O is the output matrix, where $(O_{e,o}) \in \mathbb{B}^{E \times S_O}$. T is the trace matrix, where $(T_{e,t}) \in \mathbb{B}^{E \times S_T}$.

One row of a matrix corresponds to one execution. For the trace matrix, one column corresponds to all values sampled at the same time t, for all executions. In the rest, it is supposed that inputs and outputs are present in the traces directly, i.e. there is a 1-to-1 function linking an input byte to one trace time etc. A trick to make that assumption hold in all cases is to rewrite T as the concatenation of matrices I, T and O.

How to choose inputs In the data generating experiments, inputs have to be provided to the procedure the attacker is trying to learn. Our function $\mathcal{F}$ is only as good as the inputs fed. The attacker must aim at triggering all the datapath inside the procedure, or at least all the ones necessary to use $\mathcal{F}$ afterward. In this sense, generating inputs for the analysis is similar to generating inputs for a fuzzing analysis: if the attacker has any knowledge about the procedure to learn, she can craft better inputs and thus lower the required number of executions to successfully build $\mathcal{F}$. In some cases though, the attacker does not control the inputs and she just needs to record enough of them, as discussed in subsection 5.2.

> In our examples, we consider that we reverse engineer an unknown symmetric cipher procedure, and random inputs are used.

Vectorized functions and other notations Since the traces are synchronous, then $\exists\ t_1, t_2, t_3, f$ such that $\forall e, T_{e,t_3} = f\left(T_{e,t_1}, T_{e,t_2}\right)$; there is a function f that links times t_1 and t_2 to t_3. $\boldsymbol{f}$ denotes the vectorized function that applies f element-wise to its inputs. Therefore, the previous property can be written $\exists\ t_1, t_2, t_3, f$ such that $T_{*,t_3} = \boldsymbol{f}\left(T_{*,t_1}, T_{*,t_2}\right)$. The $*$ denotes that the whole column is selected in the T matrix.

To simplify our explanations, we abuse this notation by omitting the reference to the trace matrix: $\tilde{t}$ is a shorthand to designate the vector $T_{*,t}$ formed by all values in the trace matrix at time t. Note that we can apply the tilde operator to a set of values: $\tilde{\{t_1, t_2\}} = T_{*,\{t_1,t_2\}}$, the submatrix with all the rows and the two selected columns. The new notation is therefore $\exists\ t_1, t_2, t_3, f$ such that $\tilde{t_3} = \boldsymbol{f}(\tilde{t_1}, \tilde{t_2})$ as illustrated on Figure 1.

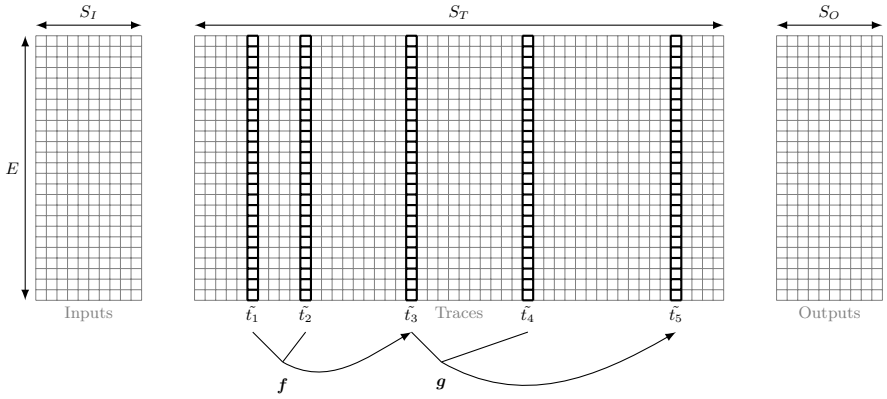

Fig. 1. Illustrative recapitulation of our notation. Each square represents a byte. The procedure is executed E times. Inputs, trace of execution and outputs are recorded and correspond to a row in the matrices. A time is the index of the column in the traces matrix T. In the illustration, we have identified relations f and g allowing deducing a column from two others. $\tilde{t_3} = \boldsymbol{f}(\tilde{t_1}, \tilde{t_2})$ and $\tilde{t_5} = \boldsymbol{g}(\tilde{t_3}, \tilde{t_4})$.

Fractional added information At numerous points, the attacker needs a measure to detect 1-to-1 and 2-to-1 functions by measuring how much information is in the output, knowing the inputs. The measure that is devised is called **fractional added information** and denoted fai, it is the estimated normalized conditional entropy.

$$fai(\tilde{b}|\tilde{a}) = \frac{H(\tilde{b}|\tilde{a})}{H(\tilde{b})}. \tag{1}$$

where H is the Shannon entropy, $\tilde{b}$ the output and $\tilde{a}$ the input of the 1-to-1 or 2-to-1 function ($\tilde{a}$ can be a vector or a two-column matrix). The normalization

is useful to obtain an easier way to detect when the output is totally determined by the inputs, or evaluate in what proportion this dependence is, independently from the entropy of the output $\tilde{b}$. A fai measure of 1 means that $\tilde{b}$ is independent from $\tilde{a}$. If fai is 0, then $\tilde{b}$ is totally determined by the knowledge of $\tilde{a}$.

In our case the input $\tilde{a}$ can be a vector (a column in the trace matrix) of bytes (1-to-1 function), or a vector of pairs of bytes (2-to-1 function).

fai should be based on entropy measures as opposed to correlation measures for its generality: we are not trying to compare an intermediate value with its leakage but two related leakages. The relation can be non-linear, such as an AES S-Box: entropy is required.

> Computing H reliably can be tricky, in particular in our case where we want to minimize the number of traces needed. The estimation of Shannon entropy is an active area of research, but as far as we know, no solution has been proposed when we have a number of samples far lower than the size of the input sets. In our case, computing fai requires to compute the entropy of an array of triplets (3-tuple) of bytes, but we have far less than 2^{24} executions. The entropy of, for example, 2^{16} triplets of bytes is necessarily lower than 16, even if they are drawn from a distribution with entropy greater than 16.
> To overcome this issue, we build an ad hoc entropy estimator. We precompute a table of entropy values given a fixed number of triplets, with both the computed Shannon entropy and the entropy of the distribution (corresponding to an infinite amount of data). We then use this table to find the estimated entropy from the computed entropy by interpolating between the closest values in the estimator table.

Strategy Whatever the sampling quality of our experiments (lossy, noisy or not), our technique follows a similar strategy.

First, the attacker retrieves the structure of the information flow inside the traces that we called the information flow hypergraph (IFH), and in relations with inputs and outputs. This hypergraph contains only hyperedges with one destination node, and one or two source nodes, for 1-to-1 and 2-to-1 functions respectively. Then she transforms the T matrix into **logic traces** L, a new matrix representing the "ideal" traces that can be extracted from the procedure (noiseless, lossless and with only 2-to-1 functions). From these logic traces, she determines the actual functions linking values between them in the same execution.

4.2 Simplifying traces

The first pass to apply to the raw traces is to remove all useless data. First, if the trace values at a given time are constant for all executions, there is no information content in it: if $\exists c \in \mathbb{B}$ such that $\tilde{t} = c \cdot \mathbf{1}$ then the attacker removes $\tilde{t}$. If two columns are identical, the attack keeps only the first (to comply with

the temporal causality principle): if $\exists t_1, t_2$ with $t_1 < t_2$ such that $\tilde{t_1} = \tilde{t_2}$ then $\tilde{t_2}$ is removed.

In our Twine traces, this simplification allows us to keep only 1 301 columns out of 92 724 initially in the raw traces. A reduction by a factor ≈ 70. Removing only constant columns leave us with 31 429 columns, a factor ≈ 3 reduction. Since this simplification is not possible when there is noise, it also means that we need to handle ≈ 70 more data in the noisy case.

4.3 Logic values and logic traces

According to our prerequirements, only 1-to-1 or 2-to-1 operations are present in the traces. Therefore, the values at each time are entirely determined by the values at one or two previous times. We can represent our computation as only composed of 2-to-1 operations, simplifying 1-to-1 operations. The traces that contain only (noiseless, lossless) 2-to-1 operations are called **logic traces**. They can easily be computed from the raw traces in the noiseless, lossless sampling case.

Raw information flow hypergraph To build the logic traces from the raw traces, we first need to extract the raw IFH, where nodes represent the raw trace times and directed hyperedges have one or two sources and one destination. These hyperedges denote a detected information flow between these sources towards the destination, identified using the fai measure. This process requires the identification of 1-to-1 and 2-to-1 functions, as the presence of a hyperedge with 1 or 2 sources in the IFH indicates the existence of a 1-to-1 or a 2-to-1 function connecting these times. Each hyperedge corresponds to a function linking times together, and they may be weighted by the corresponding fai measure. You can see an illustration on Figure 6 of the IFH corresponding to Figure 1.

Identifying 1-to-1 functions To identify a 1-to-1 function, the $fai(\tilde{t_2}|\tilde{t_1})$ measure for all pairs of time (t_1, t_2) where $0 \leq t_1 < t_2 < S_T$, is computed as illustrated on Figure 2. If $fai(\tilde{t_2}|\tilde{t_1}) = 0$, then $\exists f | \tilde{t_2} = \boldsymbol{f}(\tilde{t_1})$.

Identifying 2-to-1 functions To identify 2-to-1 functions, the attacker examines fai measures for each (t_1, t_2, t_3) triplets with $0 \leq t_1 < t_2 < t_3 < S_T$. If $fai(\tilde{t_3}|\tilde{t_1}, \tilde{t_2}) = 0$, then $\exists\ f$ such that $\tilde{t_3} = \boldsymbol{f}(\tilde{t_1}, \tilde{t_2})$.

At this stage the attacker has a complete **raw IFH** that should link inputs to outputs. The raw to logic transformation consists in the rewriting of the hypergraph in a simpler one called the **logic IFH**, while applying corresponding transformations on the raw traces to obtain the logic traces.

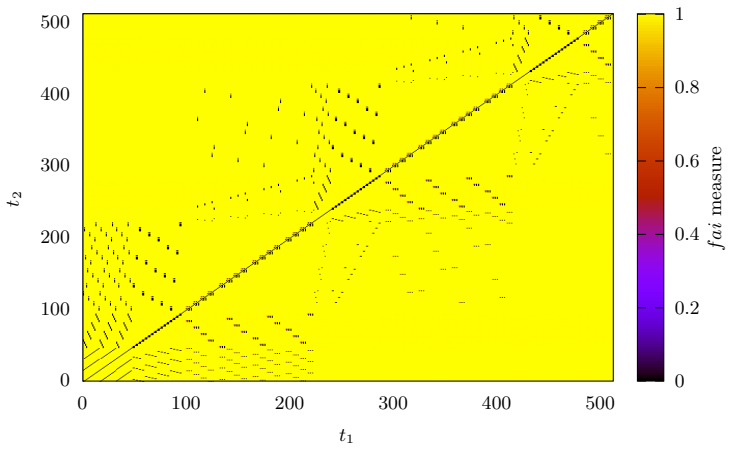

Fig. 2. The $fai(\tilde{t_2}|\tilde{t_1})$ values for the first 512 times (columns) of the traces, ranging a bit more than two AES rounds and allowing identifying 1-to-1 functions. This image visually illustrates the flow of information between times in the traces.

Raw to logic transformation in the perfect information case In the perfect information case, the raw to logic transformation is really simple: the attacker replaces chains of 1-to-1 functions by the head of the chain as illustrated on Figure 3.

Let $f_i : \mathbb{B} \to \mathbb{B}, i \in [\![0, n-1]\!]$ be a valid chain of 1-to-1 operations. Meaning that in the procedure, the input of f_i for a given $i \neq 0$ is the output of f_{i-1}. At the end of the chain, the output of f_{n-1} is used as an input of a 2-to-1 operation $g : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$. For each such 1-to-1 chain, the raw to logic traces transformation requires to remove all columns in the traces corresponding to the outputs of operations $f_i, \in [\![0, n-1]\!]$. Now the 2-to-1 operation g can be observed as a new operation $g'(a, b) = g(f_{n-1} \circ \cdots \circ f_0(a), b)$, for $a, b \in \mathbb{B}$.

In other words, a new 2-to-1 operation consuming the chain input replaces the initial 2-to-1 operation consuming the chain output, while removing the rest of the 1-to-1 chain.

Logic values In the case of a chain of 1-to-1 operations $f_i : \mathbb{B} \to \mathbb{B}, i \in [\![0, n-1]\!]$, $\forall a \in \mathbb{B}$ the vectors $[a, f_0(a), f_1 \circ f_0(a), \cdots, f_{n-1} \circ \cdots \circ f_0(a)]$ are called the **logic values**. In the perfect case, logic values can simply be projected to their first dimension without loss of information, but the concept is useful in more complex cases (cf sections 5 and 6).

Extracting 2-to-1 functions With the 2-to-1 functions identified, the attacker can now extract the functions themselves. For each 2-to-1 function g, with $\tilde{t_3} = \boldsymbol{g}(\tilde{t_1}, \tilde{t_2})$, or equivalently for each hyperedge in the logic information flow hypergraph, she builds a 256×256 table, where the first dimension is the

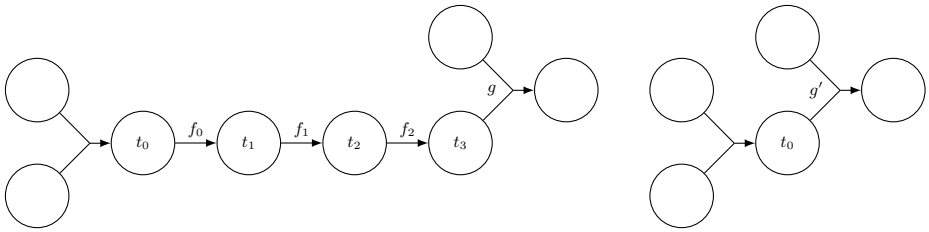

Fig. 3. Examples of partial information flow hypergraphs (IFHs). In case of a 1-to-1 chain, here with nodes from t_0 to t_3, all following nodes can be completely determined from the chain head t_0 in the perfect scenario. This allows rewriting the left hypergraph as the right one, keeping only 2-to-1 functions.

value in $\tilde{t_1}$, the second dimension is the value in $\tilde{t_2}$. The table value at indices (i, j) is $g(i, j)$ as observed in $\tilde{t_3}$.

In case of incoherence, two different outputs for the same inputs, she deduces that a 2-to-1 function has been misidentified.

If the table is not completely filled at the end of the process, it can mean two things: either some input combinations are not possible or the attack needs more traces to reverse engineer the procedure.

4.4 Executing the reconstructed function $\mathcal{F}$

Linking inputs and outputs to logic traces To finish, functions are needed to link procedure inputs and outputs to their corresponding appearance time in the logic traces. This is simply done, for each input/output byte, by finding then extracting the corresponding 1-to-1 function.

Computing the hidden procedure At this point the function $\mathcal{F}$ is built. To execute it, the attacker has to generate a logic trace corresponding to the provided fresh inputs.

1. From the provided inputs, compute the corresponding logic trace values from the input 1-to-1 functions.
2. She evaluates all 2-to-1 functions recursively in the logic IFH order: when the two inputs have been determined, she looks in the corresponding table the value of the output.
3. Now that the logic trace is completely filled, she can deduce the output bytes from the output generating 1-to-1 functions.

5 Lossy sampling

This section considers the case of lossy sampling, which occurs when the tracing method is not perfect, such as when only some bits of a register's complete

value can be extracted. In this section, traces are noiseless: for a given input, the measured traces always remain the same.

This scenario corresponds to a probing attacker: they can connect to specific wires of the targeted chip and register their values without noise.

$s : \mathbb{B} \to Im_s$ is called the **sampling (lossy) function**, a surjective mapping to Im_s the set of images by s.

In our practical experiments, our sampling function s is the Hamming weight (HW): the lossy traces contain the Hamming weights of each byte of the perfect traces seen in section 4. $\mathbb{B} = [\![0, 255]\!]$ and $Im_s = [\![0, 8]\!]$. The loss of information associated with this sampling function can be measured with fai: for a uniformly random byte vector r of size 100 000, $fai(r|\boldsymbol{HW}(r)) \approx 0.68$.

5.1 Building poor logic traces

Information flow hypergraphs (IFHs) Building the raw IFH is done similarly as in the perfect case, in subsection 4.3, from fai measures (1); but in this case the attacker cannot only keep fai values equal to 0. Let $\boldsymbol{f}$ be a 1-to-1 function linking times t_1 to t_2: $\boldsymbol{f}(\tilde{t_1}) = \tilde{t_2}$. The observed fai value is:

$$fai(\boldsymbol{s}(\tilde{t_2})|\boldsymbol{s}(\tilde{t_1})) > 0,$$

even if $fai\left(\tilde{t_2}|\tilde{t_1}\right) = 0$.

Instead, the attacker chooses an information loss threshold L_T, with $0 < L_T < 1$; and an edge in the raw IFH is created if:

$$fai(\boldsymbol{s}(\tilde{t_2})|\boldsymbol{s}(\tilde{t_1})) < 1 - L_T.$$

L_T can be determined as a multiple of the standard deviation of fai measures.

We have selected an information loss threshold of $L_T = 0.05$ for our experiments, which corresponds to 4σ. This threshold allows to accurately recover the IFH with 100 000 traces. We want to stress out that this value means that we detect that information flows if the fai measure is lower than $1 - L_T = 0.95$ instead of equal to 0. Even if the sampling function destroy a lot of information, the IFH can still be recovered quite easily.

Some edges are redundant and imprecise to model the information flow, they are called **phantom edges**. The resulting hypergraph is polluted with numerous phantom edges. $src \to t$ denotes an hyperedge with sources the set src and destination the time t. In practice, there are two cases: $src = \{t_i\}$ or $src = \{t_i, t_j\}$. For example, $\{t_1\} \to t_2$ is the hyperedge with singleton source $\{t_1\}$ and destination t_2.

Let f_1, f_2 form a chain of two 1-to-1 functions such that $\tilde{t_2} = \boldsymbol{f_1}(\tilde{t_1})$ and $\tilde{t_3} = \boldsymbol{f_2}(\tilde{t_2})$. Then, the following hyperedges are found in the hypergraph: $\{t_1\} \to t_2$, $\{t_2\} \to t_3$, $\{t_1\} \to t_3$ and $\{t_1, t_2\} \to t_3$. The lowest fai measure is most probably obtained for the latter edge since the attacker has more information in the source nodes (two nodes instead of one). In this example, she would like to only keep $\{t_1\} \to t_2$ and $\{t_2\} \to t_3$. The edges $\{t_1\} \to t_3$ and $\{t_1, t_2\} \to t_3$ are phantom edges: they incorrectly describe the flow of information due to f_1 and f_2.

Cleaning the hypergraph To clean the polluted hypergraph of its phantom edges, the following operations are done as preprocessing.

- For two given hyperedges $src_1 \to t$ and $src_2 \to t$, src_1 and src_2 being two sets of input times, if $src_1 \subset src_2$ and $fai(\tilde{t}|s\tilde{r}c_1) < fai(\tilde{t}|s\tilde{r}c_2) + L_T$, then remove edge $src_2 \to t$, otherwise remove $src_1 \to t$. In plain words, if there is a 1-to-1 function that gives as much information on t, with margin of error L_T, as a 2-to-1 function, the attacker only keeps the edge corresponding to the 1-to-1 function as illustrated on Figure 4. Indeed, it means that the second input does not bring significant additional information.
- Keeping only the best sources: for all hyperedges $src_i \to t$, all hyperedges with the same destination t, keep only the ones with

$$fai(\tilde{t}|s\tilde{r}c_i) < \left(min_{\forall i} fai(\tilde{t}|s\tilde{r}c_i)\right) + L_T.$$

  There are lots of “transitive" edges in the polluted hypergraph: if there is an edge $\{t_1\} \to t_2$ and $\{t_2\} \to t_3$, there is a very high probability to also get a $\{t_1\} \to t_3$ edge that does not bring any useful information and which is likely removed here.
- Any edge directed at a procedure input is removed in case there are dependencies between input bytes.

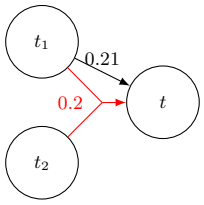

Fig. 4. If two hyperedges with the same fai measure, up to L_T, are connected to the same node: keep in priority 1-to-1 hyperedges. Here $src_1 = \{t_1\} \subset src_2 = \{t_1, t_2\}$. In the figure, the fai is displayed next to the hyperedge. The red 2-to-1 hyperedge has to be removed since $0.21 - L_T < 0.2$.

Ancestor sets From the clean raw hypergraph, the attacker tries to find clusters of nodes corresponding to the same logic value, *i.e.* set of times that are part of a chain of 1-to-1 function. Intuitively, the goal is to find the minimal pair of times, the minimal ancestor set, that completely determines any given time t. If two times t_1 and t_2 share a common minimal ancestor set, they are part of the same cluster, as shown on Figure 5 for times t_3, t_4, t_5 and t_6.

More formally, a function ANCESTORSETS is defined that accepts as inputs the IFH, a set of times src_{init} and a reduction function *reduct*. *reduct*, in our clustering algorithm, maps to the head, the minimal element, for all elements of a cluster. Without explicit precision, *reduct* starts as the identity. ANCESTORSETS returns the set of ancestors for a given set of times, i.e. all the sets that can generate src_{init} according to hg.

From the function ANCESTORSETS, it is possible to define a partial order noted $\leq$: $src_1 \leq src_2$ if $src_1 \in$ ANCESTORSETS(hg, src_2). Reflexivity is trivially demonstrated from the initialization of the *explored* mapping. Transitivity and antisymmetry are a consequence of a directed acyclic hypergraph that allows defining a partial order on nodes.

Finally, the MINANCESTORSETS function is defined: from a directed acyclic hypergraph, a time t and an optional *reduct* function, compute the set of sets ANCESTORSETS$(hg, t, reduct)$. Then order the resulting sets and filter them: if $src_1 \leq src_2$ and $src_1 \neq src_2$, do not keep src_2.

It is possible to have several minimal ancestor sets since the sorting relation is a partial order. In this case, the minimal minimal (twice minimal) ancestor set is selected: the set in the minimal ancestor sets that minimize the fai measure with respect to the destination time.

Clustering times by logic values The GET1TO1CLUSTERS function, which relies on minimal ancestors sets, is used to efficiently cluster logic values. At this stage, times are clustered based on a common pair of source nodes. However, having common sources does not necessarily mean that two times correspond to the same logic value. The attacker post-processes these clusters by dividing them into subclusters based on destinations, using a heuristic. Times t_1 and t_2 are considered to be in the same cluster if the ratio $\frac{\#dest(t_1) \cap dest(t_2)}{\#dest(t_1) \cup dest(t_2)} > 0.5$, where $dest(t)$ is the set of destination times for which t is a source. The idea is that t_1 and t_2 should belong to the same subcluster if they have similar descendants. If they don't, it is a clear sign that they belong to different logic values.

Poor logic values If logic traces are built as in the perfect case (lossless sampling), a problem quickly arises: projecting logic values to their first dimension would lose information. For a given 1-to-1 chain $f_0, f_1, \cdots, f_{n-1}$ of length a given n, the corresponding logic values are now $[s(a), s \circ f_0(a), s \circ f_1 \circ f_0(a), \cdots, s \circ f_{n-1} \circ \cdots \circ f_0(a)], \forall a \in \mathbb{B}$.

Since there is no noise, there are at most 256 (the size of $\mathbb{B}$) different vectors, 1 per $a \in \mathbb{B}$. As such the conversion to logic traces is not a projection to the first

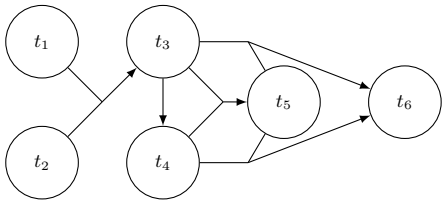

Fig. 5. The GET1TO1CLUSTERS algorithm, with postprocessing, correctly determines that all nodes t_3, t_4, t_5 and t_6 belong to the same logic value since $clusters(\{t_1, t_2\}) = \{t_3, t_4, t_5, t_6\}$

dimension, as in the perfect case, but a projection from logic values (vectors) Im_s^{n+1} to $\mathbb{B}$.

This projection is built by enumerating all logic values and arbitrary assigning a value in $\mathbb{B}$ to each. As in the perfect case (cf section 4), the attacker can now generate the logic traces, by projecting logic values to $\mathbb{B}$. She only has 2-to-1 functions in the resulting traces.

A problem arises when there is no 1-to-1 chain between two 2-to-1 functions, or if the chain is too small with respect to the sampling function: there is not enough information in the logic value to properly reconstruct the initial value (before sampling). We call these **poor logic values**.

> Let us imagine a 1-to-1 function f_0 that feeds a 2-to-1 function. If we look at our logic values $[HW(a), HW(f_0(a))]$, we have at most 81 different vectors, even if all 256 byte values are taken by a.

Consequently, we refer to these logic traces as **poor logic traces** due to their poor information content. However, **it is typically possible to recreate complete, information rich logic traces** using dedicated techniques: this is the "enrichment" process described below.

5.2 Enriching poor logic traces

The objective now is to enrich our poor logic traces, which contain only 2-to-1 functions, to recover the information lost due to the lossy sampling process. The underlying principle behind our solution is straightforward: although the actual intermediate value is lost to lossy sampling, the impact of this value on the rest of the trace can still be discerned.

Poor logic values are lossy values, and rich logic values are lossless (as in section 4). Enrichment is the process of transforming poor logic traces, containing only poor logic values to rich logic traces, containing only rich logic values.

Enrichment by second order side effects Enrichment is done one time, one column in the trace matrix, at a time. The attacker considers the pattern of two elementary operations illustrated on Figure 6.

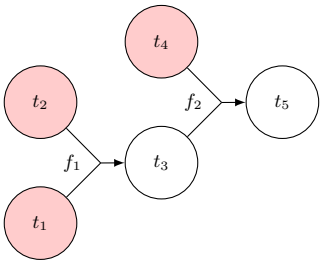

Fig. 6. Wealth pattern: subhypergraph with specific information rich (red background) and poor (white background) nodes.

Notations In this pattern, the attacker considers 5 times (or columns) named t_1, t_2, t_3, t_4, t_5, where $\exists f_1, f_2 | \tilde{t_3} = \boldsymbol{f_1}(\tilde{t_1}, \tilde{t_2})$ and $\tilde{t_5} = \boldsymbol{f_2}(\tilde{t_3}, \tilde{t_4})$. She observes $\tilde{t_1}, \tilde{t_2}, \boldsymbol{s}(\tilde{t_3}), \tilde{t_4}$ and $\boldsymbol{s}(\tilde{t_5})$ and she wants to reconstruct $\tilde{t_3}$. It is important to note that if she observes $\boldsymbol{s}(\tilde{t_4})$ and $\tilde{t_5}$ instead (t_4 is poor, t_5 is rich), she gets the same results by swapping the columns for the enrichment procedure.

A given pair $(\alpha, \beta) \in \mathbb{B}^2$ determines the value $\gamma = f_1(\alpha, \beta)$. Therefore, it can be observed that $\boldsymbol{s}(\tilde{t_3}[\tilde{t_1}{=}\alpha, \tilde{t_2}{=}\beta]) = s(\gamma) \cdot \mathbf{1}$.

$[\tilde{t_1}{=}\alpha, \tilde{t_2}{=}\beta]$ is a "selector" notation to say that rows e in $T_{e,t}$ (the trace matrix) are selected such that $T_{e,t_1} = \alpha$ and $T_{e,t_2} = \beta$. With respect to Figure 1, we are selecting a submatrix by selecting both a column and some specific rows corresponding to the specified condition.

In plain words, if rows are selected such that the pair value (α, β) at times t_1 and t_2 is constant, a constant vector is observed at time t_3 for these same rows. But $\tilde{t_4}[\tilde{t_1}{=}\alpha, \tilde{t_2}{=}\beta]$ is supposedly not a constant vector. The attacker can try to recover γ from its influence on t_5 through f_2. We invoke currying to note $f_2(\gamma) : \mathbb{B} \mapsto \mathbb{B}$ the function created from f_2 by fixing its first inputs to γ. The goal is now to cluster pairs of value for times t_1 and t_2 depending on the resulting $f_2(\gamma)$ function.

Given two pairs $(\alpha_1, \beta_1), (\alpha_2, \beta_2)$, both pairs map to the same value $\gamma_1 = \gamma_2$ through f_1 if the two functions $f_2(\gamma_1)$ and $f_2(\gamma_2)$ are indistinguishable.

Since the number of possible functions from a given set S_1 to a given set S_2 is $|S_2|^{|S_1|}$, an attacker can recover the value γ even if the sampling s loses a lot of information.

Sieve In the lossy case, we can determine whether $f_2(\gamma_1)$ and $f_2(\gamma_2)$ are identical by applying a sieve. Specifically, for a given set of inputs, we check if the outputs of the two functions match. For all pairs (α, β), the vector $\boldsymbol{v}_{\alpha,\beta} = [s \circ f_2(\gamma)(0), s \circ f_2(\gamma)(1), \cdots, s \circ f_2(\gamma)(255), s(\gamma)]$ is calculated where $\gamma = f_1(\alpha, \beta)$. Here, $s \circ f_2(\gamma)(\delta)$ refers to the value observed in vector $\tilde{t_5}$ for a given triplet α, β, δ at times t_1, t_2, t_4 respectively. If these three values are fixed, then only one value in $\tilde{t_5}$ can be observed, which is reported in $\boldsymbol{v}_{\alpha,\beta}$.

If a value is not observed in vector $\tilde{t_5}$ (if the value δ is not seen in vector $\tilde{t_4}$, it is not possible to observe $s \circ f_2(\gamma)(\delta)$ in $\tilde{t_5}$), the corresponding coordinate in the vector is replaced by a special "don't care" value, denoted as $*$.

The $sieve : \mathbb{B}^{256+1} \times \mathbb{B}^{256+1} \mapsto \{\text{true}, \text{false}\}$ function assesses whether two vectors $\boldsymbol{v}_{\alpha_1,\beta_1}$ and $\boldsymbol{v}_{\alpha_2,\beta_2}$ correspond to the same value in vector $\tilde{t_3}$. Here, $sieve(\boldsymbol{v}_{\alpha_1,\beta_1}, \boldsymbol{v}_{\alpha_2,\beta_2}) = \text{true}$ if and only if $f_1(\alpha_1, \beta_1) = \gamma_1 = \gamma_2 = f_1(\alpha_2, \beta_2)$ with enough data.

The *sieve* function performs a pairwise comparison of vector coordinates and returns true only if they are all equal, with $*$ being considered equal to all values in $\mathbb{B}$.

Given that there are at most 256 different values for γ, there can be at most 256 sets of pairs (α, β) that are transitively equal according to the *sieve* function. The final step involves numbering these sets and using these values to generate the new column $\tilde{t_3}$.

Conditions for enrichment

When is enrichment possible ? The critical question is: at what conditions is enrichment possible ? The answer may prevent our technique to work and point to efficient countermeasures. Several conditions that may prevent enrichment have been identified:

1. The structure of the graph of 2-to-1 functions. Is the wealth pattern in Figure 6 always present? If at any point in the enrichment process, the pattern cannot be found, enrichment stops and the secret procedure cannot be reversed. A countermeasure to Generic SCARE would then be to implement the procedure such that this wealth pattern cannot be found. This is harder than simply ensuring that $t_4 > t_3$ in Figure 6. Indeed times can be swapped if the operation respect the partial order defined by the IFH.

   > In our experiments, and to our surprise, this wealth pattern was enough to fully recover both AES and Twine. We conjecture that it is so because symmetric encryption algorithms, by being invertible, imply a specific structure of the IFH.

   Other wealth patterns, sub-hypergraphs with specific rich and poor nodes, can be used to recover information, not just the one presented on Figure 6.
2. If the sampling function loss of information has a structure that propagates through a 2-to-1 function, it may prevent enrichment. E.g. if the sampling function is “parity” but the 2-to-1 function preserves parity, there is no information propagation on the rest of the trace.
3. The number of observed traces E must be high enough (detailed in next paragraph).

How many traces are needed ? This depends on several factors : the sampling function s, the number e of edges in the logic IFH (equal to the number of 2-to-1 functions) and the desired probability threshold t of success.

For the enrichement process We need enough traces to fill the vectors $\boldsymbol{v}_{\alpha,\beta}$: if not, they are filled with don't care values * and enrichment is not possible.

For example, if the attacker chooses a success probability of $t = 0.95$, with $e = 468$ relations as in our AES use case, we need o traces to cluster the vectors $\boldsymbol{v}_{\alpha,\beta}$ into at most 256 sets with probability $t^{1/e} = 0.95^{1/468} \approx 0.9999$. We estimated the required number o of traces to distinguish 256 vectors of size 257, as in our use cases, on Figure 7 for various sampling functions. From theses cumulative probabilities, we can estimate the required number of traces o to reach a cumulative probability of 0.9999 for our specific sampling function s. From Figure 7, with the Hamming weight sampling function, we require 92 traces to distinguish 256 vectors with probability 0.9999.

Since we need to get traces for all pairs (α, β), we can deduce a lower bound for the required number of traces as $o \cdot 2^{8+8}$.

In conclusion, if we want a success rate $> 95\%$, with 468 edges in the logic IFH, intermediate values uniformly distributed, we need $92 \cdot 2^{16} \approx 6\,000\,000$ traces.

Experimentally, we observed that the probability of success is dramatically reduced under 6 millions traces. However, the experiment is too resource-intensive to establish success rate statistics.

From Figure 7, the worst sampling function is "Uniform 2", where bytes are split in two equal classes. The parity function would have the same cumulative probabilities. In this case, the minimum number of traces is $126 * 2^{16} \approx 8\,200\,000$.

For 2-to-1 functions extraction How many traces are required to extract 2-to-1 function (cf subsection 4.3)? To fill a function table, using the parameters for our problem instance: 2^{16} the number of possible inputs for a 2-to-1 functions, applied to the coupons collector problem, the expectation for the number E of executions to observe all procedure inputs is:

$$2^{16} \cdot \sum_{k=1}^{2^{16}} \frac{1}{k} \approx 765\,000 \text{ executions.}$$

If the perfect case, the number to extract functions dominates and around 765 000 traces are needed. In the lossy case, the enrichment is the part requiring more traces: 6 000 000 are needed.

5.3 Procedure recovery from rich logic traces

To finalize the procedure recovery from the rich logic traces, the traces resulting from the enrichment process on poor logic traces, it is possible to fall back to the perfect case as presented in sections 4.3 and 4.4.

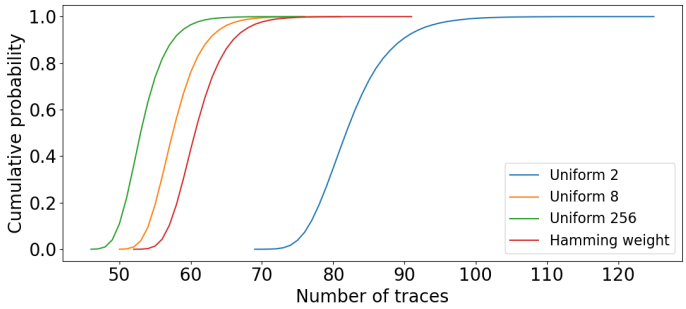

Fig. 7. Cumulative probabilities to distinguish among 256 vectors of size 257, for different sampling functions. Uniform n is a sampling function that assigns byte values into n bins uniformly.

6 Noisy lossy sampling

This section considers a scenario in which the sampling method is noisy, meaning that sampling the same data twice may yield different results. Consequently, the attacker must now work with probabilities.

Our analysis is conducted under the assumption of Gaussian noise. In this context, the noisy sampling function $\mathcal{N}(a, \sigma, r)$ is defined as $s(a) + \sigma * \sqrt{2} * erf^{-1}(2 \cdot r - 1)$, where a is the data to be sampled, σ is the noise standard deviation, and $r \in [0, 1]$ is a random value drawn from a uniform distribution (with a different value being drawn for each sample).

To simulate a measurement with an acquisition device such as an oscilloscope, the resulting values are digitized as byte values: they are scaled (the maximum value is 255 and the minimum value is 0), and then binned to the closest integer value.

In our experiments, we choose the very low standard deviation $\sigma = 0.1$. Indeed, the number of traces needed for a successful recovery of the procedure grows quickly with σ. This suggests that a real-life scenario would probably need other techniques to build logic values as discussed in section 7.

6.1 Building poor logic traces

Information flow hypergraphs Recovering the IFH is exactly the same as in the lossy case (cf subsection 5.1), the only difference is that the information loss threshold L_T must now account for Gaussian noise in addition to sampling loss.

Poor logic values With the presence of noise, the logic values vectors may have more values than elements in $\mathbb{B}$. Since the noise is iid, the attacker can use a

clustering method on these vectors. For a given clustering method and a cluster count between 1 and 256, the attacker evaluate the maximum distance between a value and its associated cluster centre. She chooses the cluster count that has the lowest maximum distance as the correct count. The clustering method can now be used to generate the poor logic values.

We use a hierarchical clustering technique, which worked well with $\sigma = 0.1$.

6.2 Enriching poor logic traces

In the noisy case, since our sampling is lossy, the attacker also needs to enrich our poor logic traces. The technique is similar: she identifies a value $\gamma = f_1(\alpha, \beta)$ in column $\tilde{t_3}$ by observing its impact on function f_2.

But in the lossy case, f_2 was characterized by the vector with 257 coordinates (1 coordinate per δ value, plus one for γ). In the noisy case, for a given γ and δ, the attacker may observe different values $f_2(\gamma)(\delta)$ because of the noise. As a consequence, the attacker characterizes f_2 with a matrix in this case.

$$m_{\alpha,\beta} = \begin{bmatrix} p_0(f_2(\gamma)(0) = 0) & p_1(f_2(\gamma)(1) = 0) & \cdots & p_{255}(f_2(\gamma)(255) = 0) \\ p_0(f_2(\gamma)(0) = 1) & p_1(f_2(\gamma)(1) = 1) & \cdots & p_{255}(f_2(\gamma)(255) = 1) \\ \vdots & \vdots & \ddots & \vdots \\ p_0(f_2(\gamma)(0) = 255) & p_1(f_2(\gamma)(1) = 255) & \cdots & p_{255}(f_2(\gamma)(255) = 255) \end{bmatrix}.$$

In $m_{\alpha,\beta}$, each column δ is the probability distribution of observing $f_2(\gamma)(\delta)$ in the traces. The attacker hopes that even if she has noise on the δ and ϵ values ($\epsilon = f_2(\gamma, \delta)$), $m_{\alpha,\beta}$ is still characteristic of the γ value.

Now, instead of a sieve, she uses a distance function to compare two matrices. In this case she sums the Euclidean distances of the columns two by two.

$$dist(m_{\alpha_1,\beta_1}, m_{\alpha_2,\beta_2}) = \sum_{0 \le c < 256} \left(\sqrt{\sum_{0 \le r < 256} (m_{\alpha_1,\beta_1})^2_{r,c} - (m_{\alpha_2,\beta_2})^2_{r,c}} \right).$$

The attacker uses the distance *dist* to cluster the pairs (α_1, β_1) and (α_2, β_2): if they are close, she assumes that they induce the same γ value.

At this step, the attacker has obtained rich logic traces, and she can finish the procedure recovery as in the other cases.

7 Applying Generic SCARE in practice

Our technique is only a theoretical one for now: numerous challenges obstruct its direct implementation on a real device.

We want to stress that these challenges are not specific to this technique but concern numerous other works. For example, all blind side channel analyses (e.g., [5,12,13,14]) suffer from the same issues and can only be considered theoretical attacks.

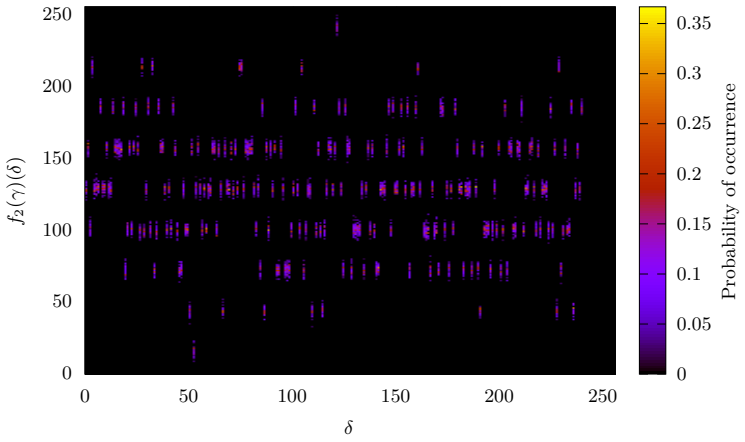

Fig. 8. Visual representation of one particular matrix $m_{\alpha,\beta}$. The nine horizontal bands represent the nine Hamming weight values with noise. For a fixed value δ, the values $f_2(\gamma)(\delta)$ represents a Gaussian distribution centred on one Hamming weight value.

Constant time Generic SCARE necessitates synchronous traces, as resulting from constant time procedures. However, constant time is not an absolute concept: significant efforts are made to employ instructions with a fixed execution duration, but as of now, it is not feasible to guarantee that a sequence of instructions maintains constant time at the scale of a cryptographic cipher. Notably, microarchitectural aspects introduce variability, sometimes dependent on intermediate values. At another level, variability in the clock signal (the clock period is never perfectly nominal) can be discernible at the scale of a cipher.

Hence, the concept of constant time is somewhat nebulous in practice, leading to a preference for stating that execution time is independent of intermediate values, as this provides a more accurate description of the desired property.

Timing variations may be modelled as additional noise and thus might be manageable within the noisy scenario. However, this depends on several experimental factors such as our ability to resynchronize traces, the importance of timing variability, etc.

Non-independence of leakages An assumption inherent to our model is that leakages are independent: at each point in time, only one intermediate value leaks. This assumption, however, does not hold up in practice. First, due to the electrical properties of circuits where capacitances store energy, and thus the leaked information, over a certain period of time. Second, because of the microarchitectural pipelining where a data point is processed over multiple clock cycles, in parallel to other data points.

This discrepancy implies that our method of constructing the IFH may not be directly applicable in real-world situations. In our model, each node corresponds to one timing in the raw IFH, prior to its transformation to the logical IFH.

However, in practice, it is not possible to construct the raw IFH, as the relation between intermediate value and timing is not isomorphic.

Therefore, the logical IFH must be constructed directly, bypassing the raw IFH stage. Determining the optimal approach to achieve this remains an open problem.

Relating leakage from multiple points-of-interest The challenges multiply when attempting to relate leakages from different times in the traces, a scenario common in blind side channels or when attacking masked implementations.

In the classical correlation power analysis (CPA) scenario, one can manipulate the model applied to the known value to establish a precise relationship between the model and the leakage. This allows for the use of an efficient distinguisher. For instance, in CPA, the model is constructed in such a way that the relationship is linear, making Pearson's correlation a suitable tool.

However, this isn't an option in our context. The relationship between two leakages could be arbitrary, making our fai distinguisher relatively inefficient due to its generality. Thus, in the presence of noise, detecting a flow of information between different leakages becomes a formidable task, with no solution that we are aware of in the literature.

Preliminary experiments carried out by measuring the power consumption of a Cortex-M3 microcontroller with a ChipWhisperer has shown that while detecting 1-to-1 relations was straightforward, identifying 2-to-1 relations proved impossible due to false positives. Overcoming this hurdle will necessitate an efficient preprocessing method applied to traces with appropriate timing and sampling resolutions.

Leakage profiling is difficult to reproduce Detecting 1-to-1 and 2-to-1 relations is one aspect, but extracting the actual function from a known relation is an entirely different challenge. This requires a profiled analysis on the leakage, with a high enough success rate to continue the analysis. Perfect success is not necessary, but the required threshold is yet to be determined.

Our preliminary experiments on real traces have shown varying success rates, ranging from as low as 4% (still better than random guessing) to as high as 100% depending on the targeted function, achieved using linear discriminant analysis (LDA). This is possibly resulting from the non-independence of leakages.

In theory, the performance should improve with the use of deep learning models. However, our attempts to utilize models from the literature have been unsuccessful. These models need to be heavily customized for our specific use cases, a task which is not straightforward. Nevertheless, this topic is a dynamic research area, and recent works [1] suggest possible improvements here.

8 Countermeasures

The point to the attacks presented above is to understand what weakens an implementation with respect to reverse engineering. Therefore, to counter the

attack, it is enough to force the implementation not to meet the attack requirements.

Desynchronization A possible countermeasure is to make the procedure complete in a variable amount of time, maximizing variations in elementary operations. It is important not to introduce new vulnerabilities, the timing variation must not depend on secret data. The desynchronization often results in a jitter following a normal distribution. In this case, desynchronization is similar to introduce Gaussian noise on the values. It is not an absolute countermeasure, but it can make the attacker requires an enormous amount of data for the attack to succeed. It is probable that resynchronization techniques limit the effectiveness of this countermeasure.

3-to-1 functions and wider data width A key point of our attack is the tractability to enumerate all triplets of values describing a function, by limiting ourselves to 2-to-1 functions on bytes. The attack will become a lot harder in the presence of 3-to-1 functions (or more) or in case of a wider data width. 64-bit 2-to-1 functions require to enumerate 2^{128} possibilities for the input of the function, an amount not tractable today. But this wider data width must be algorithmic, without possible reduction. For example, a 64-bit xor operation can also be seen as 8, 8-bit, xor operations.

Short 1-to-1 function chains An algorithmic defence against our attack would be to limit the length of the 1-to-1 function chains. Indeed a long chain allows the conversion to logic values to recover a lot of the information lost by sampling. In particular, S-Boxes applied to data with length lower or equal to 8 bits helps the reverse engineering.

Masking Masking can be a countermeasure if random masks cannot be observed by the attacker, nor deduced through 1-to-1 chains. Since these values have not been generated from previous values, it is not possible to enrich them and the attack is not applicable.

Yet these random masks, by construction, do not propagate over long distances in the IFH, which is a distinctive feature of masks with respect to other intermediate values. Therefore, it may be possible for the attacker to bypass this countermeasure.

9 Conclusion

Generic SCARE demonstrates a counter-intuitive possibility: it is possible to completely reverse engineer a given procedure from inputs, outputs and traces of execution even in the presence of lossy sampling and noise. The power consumption of a chip, knowing associated inputs and outputs, is enough to reconstruct the procedure. Even if only a theoretical attack for now, we show that there are no fundamental blockers to reverse engineer a procedure such as a symmetric

cipher by observing the power consumption of the chip executing it. For that, we rely on the observable consequences of intermediate values when propagating along the information flow hypergraph.

This work is limited in its capacity to deal with real life scenarios, and this should be improved. The purely stochastic approach should be enhanced with a deep learning approach for the different reasons mentioned in section 7.

In these scenarios, the attacker likely has some knowledge of the targeted system: they may know the chip model or a part of the targeted algorithm. All this information may be used to make the attack more efficient. However, how to incorporate partial knowledge into our technique has yet to be determined.

Declarations

Ethical approval Not applicable.

Competing interests The authors declare that they have no competing interests. This work was done on the research time of the authors, in a tenure-equivalent position.

Authors' contributions This is a joint work between R.L. and H.LB. They both devised the theory, wrote and reviewed the paper. R.L. is the main author of the proof of concept source code.

Funding This work was not supported by any grant.

Availability of data and materials The source code for the proof of concept, used in the paper to reports our results will be available upon acceptance. There is currently no way to reliably share the code anonymously while respecting our institutions' IP policies.

References

1. Bursztein, E., Invernizzi, L., Král, K., Moghimi, D., Picod, J.M., Zhang, M.: Generic attacks against cryptographic hardware through long-range deep learning. arXiv preprint arXiv:2306.07249 (2023)
2. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M.J. (ed.) Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1666, pp. 398–412. Springer (1999). https://doi.org/10.1007/3-540-48405-1_26, https://doi.org/10.1007/3-540-48405-1_26
3. Clavier, C.: An improved SCARE cryptanalysis against a secret A3/A8 GSM algorithm. In: McDaniel, P.D., Gupta, S.K. (eds.) Information Systems Security, Third International Conference, ICISS. vol. 4812, pp. 143–155. Springer (2007)
4. Clavier, C., Isorez, Q., Marion, D., Wurcker, A.: Complete reverse-engineering of aes-like block ciphers by SCARE and FIRE attacks. Cryptogr. Commun. **7**(1), 121–162 (2015)

5. Clavier, C., Reynaud, L.: Improved blind side-channel analysis by exploitation of joint distributions of leakages. In: Fischer, W., Homma, N. (eds.) Cryptographic Hardware and Embedded Systems - CHES. vol. 10529, pp. 24–44. Springer (2017)
6. Cristiani, V., Lecomte, M., Hiscock, T.: A bit-level approach to side channel based disassembling. In: Belaïd, S., Güneysu, T. (eds.) Smart Card Research and Advanced Applications - 18th International Conference, CARDIS 2019. vol. 11833, pp. 143–158. Springer (2019)
7. Daemen, J., Rijmen, V.: The pelican MAC function. IACR Cryptol. ePrint Arch. p. 88 (2005), http://eprint.iacr.org/2005/088
8. Daudigny, R., Ledig, H., Muller, F., Valette, F.: Scare of the des. In: Ioannidis, J., Keromytis, A., Yung, M. (eds.) Applied Cryptography and Network Security. pp. 393–406. Springer (2005)
9. Eisenbarth, T., Paar, C., Weghenkel, B.: Building a side channel based disassembler. Trans. Comput. Sci. **10**, 78–99 (2010)
10. Guilley, S., Sauvage, L., Micolod, J., Réal, D., Valette, F.: Defeating any secret cryptography with SCARE attacks. In: Abdalla, M., Barreto, P.S.L.M. (eds.) Progress in Cryptology - LATINCRYPT. vol. 6212, pp. 273–293. Springer (2010)
11. Jap, D., Bhasin, S.: Practical reverse engineering of secret sboxes by side-channel analysis. In: IEEE International Symposium on Circuits and Systems, ISCAS. pp. 1–5. IEEE (2020)
12. Le Bouder, H., Lashermes, R., Linge, Y., Thomas, G., Zie, J.: A multi-round side channel attack on AES using belief propagation. In: Cuppens, F., Wang, L., Cuppens-Boulahia, N., Tawbi, N., García-Alfaro, J. (eds.) Foundations and Practice of Security - 9th International Symposium, FPS. vol. 10128, pp. 199–213. Springer (2016)
13. Linge, Y., Dumas, C., Lambert-Lacroix, S.: Using the joint distributions of a cryptographic function in side channel analysis. In: Prouff, E. (ed.) Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014. vol. 8622, pp. 199–213. Springer (2014)
14. Meraneh, A.H., Clavier, C., Le Bouder, H., Maillard, J., Thomas, G.: Blind side channel on the elephant LFSR. In: SECRYPT (2022)
15. Novak, R.: Side-channel attack on substitution blocks. In: Zhou, J., Yung, M., Han, Y. (eds.) Applied Cryptography and Network Security, First International Conference,ACNS. vol. 2846, pp. 307–318. Springer (2003)
16. Réal, D., Dubois, V., Guilloux, A., Valette, F., Drissi, M.: SCARE of an unknown hardware feistel implementation. In: Grimaud, G., Standaert, F. (eds.) Smart Card Research and Advanced Applications, 8th IFIP WG 8.8/11.2 International Conference, CARDIS. vol. 5189, pp. 218–227. Springer (2008)
17. Rivain, M., Roche, T.: SCARE of secret ciphers with SPN structures. In: Sako, K., Sarkar, P. (eds.) Advances in Cryptology - ASIACRYPT. vol. 8269, pp. 526–544. Springer (2013)