Secure Message Authentication in the Presence of Leakage and Faults (Long Paper)

Francesco Berti[2], Chun Guo[3,4], Thomas Peters[1], Yaobin Shen[1], François-Xavier Standaert[1]

[1] UCLouvain, ICTEAM/ELEN/Crypto Group
[2] TU Darmstadt, Germany, CAC - Applied Cryptography
[3] School of Cyber Science and Tech., Shandong University, Qingdao, Shandong, 266237, China
[4] Shandong Research Institute of Industrial Technology, Jinan, Shandong, 250102, China

Abstract. Security against side-channels and faults is a must for the deployment of embedded cryptography. A wide body of research has investigated solutions to secure implementations against these attacks at different abstraction levels. Yet, to a large extent, current solutions focus on one or the other threat. In this paper, we initiate a mode-level study of cryptographic primitives that can ensure security in a (new and practically-motivated) adversarial model combining leakage and faults. Our goal is to identify constructions that do not require a uniform protections of all their operations against both attack vectors. For this purpose, we first introduce a versatile and intuitive model to capture leakage and faults. We then show that a MAC introduced at Asiacrypt 2021 natively enables a leveled implementation where only its underlying tweakable block cipher must be protected, as long as only its tag verification can be faulted. We finally describe two approaches to amplify security in the case where also the tag generation can be faulted. One is based on iteration and requires the adversary to inject increasingly large faults to succeed. The other is based on randomness and allows provable security against differential faults.

Keywords: side-channels · leakage · faults · combine attacks · mode-level protections

1 Introduction

The security of cryptographic implementations against leakage has been a topic of intense attention over the last two decades. Due to the physical nature of side-channel attacks, solutions to prevent have been shown to benefit highly from a cross-layer approach: at the implementation level, countermeasures like masking or shuffling can amplify the leakage noise [MOP07]; at the primitive level, the design of (tweakable) block ciphers or permutations can be optimized for the implementation of these countermeasures (see, e.g., [GLSV14]); at the protocol level, modes of operation can be developed in order to enable so-called "leveled implementations", where different parts of the mode need different levels of security against leakage, surveyed in [BBC+20]. Overall, and despite a unified analysis of this cross-layer approach remains a challenge (e.g., due to the difficulty to model leakage in a way that is at the same time practically relevant and theoretically sound), this mix of theoretical and practical advances has lead to the possibility to ensure high security against side-channel attacks at affordable implementation cost.

This situation is echoed when considering faults. It is even amplified due to the even larger versatility of the attack vectors [BCN+06], making the progresses towards a cross-

layer approach as developed against leakage more intricate. Here as well, implementation-level countermeasures (e.g., taking advantage of redundant computations and error correction) were first investigated [JT12]. But recent progresses have shown that working at the primitive level can be beneficial (see, e.g., the design of FRIET [SBD+20] or the DEFAULT layer in [BBB+21]). And a similar observation holds for investigations at high-level abstractions, where so-called atomic models of computation capture attacks where adversaries can induce faults between atomic operations of varying granularities [FG20, AOTZ20].

Based on this state-of-the-art, an important question is whether it is possible to combine these solutions towards security against side-channels and faults? Such a question is motivated by the risk of so-called combined attacks [RLK11]. It is also known to be non-trivial when considering countermeasures at the implementation level, where side-channel and fault resistance can lead to somewhat contradictory requirements [REB+08]. As a result, and as a natural starting point, we consider the question whether the concept of leveled implementation can be generalized to security against leakage and faults. In other words, is it possible to implement basic cryptographic functionalities without uniformly protecting all their operations against leakage and faults with countermeasures?

In this paper, we answer this question positively for the case of Message Authentication Codes (MACs). Our contributions in this respect are twofold.

First, we show that the recent LR-MAC1 leakage-resilient MAC proposed as Asiacrypt 2021 natively offers good features for this purpose [BGPS21]. Precisely, its tag verification can be implemented such that only the Tweakable Block Cipher (TBC) that manipulates its long-term secret require security against leakage and faults. By contrast, all the other operations can leak in an unbounded manner and can be faulted arbitrarily. Less positively, we also show that this security guarantee does not extent to its tag generation.

Second, we show that the security of a tag generation can be improved against leakage and faults in two different directions. On the one hand, and assuming that inserting faults on multiple and large intermediate computations becomes increasingly difficult for the adversary, we show that iterative constructions can leverage the amount of faults to inject as a security parameter. We prove the security of a new construction, coined LR-MACd, in order to illustrate this claim. On the other hand, the attacks against the Asiacrypt 2021 MAC are possible under a (strong) stuck-at fault model. Assuming that, for some technologies, only differential faults are possible, we show that randomizing the tag generation can lead to strong (differential fault) security with leakage. We prove the security of a third construction, coined LR-MACr, in order to illustrate this claim. It confirms the intuition that as long as the TBC is unpredictable under leakage, faults do not help adversaries since LR-MACr takes a random sequence together with a message as input and the random sequence randomizes the computation of the tag.

The leakage models we use for this purpose are the (standard) ones proposed in [BGPS21]: we mix unbounded leakage for the non-sensitive computations and require unpredictability for the TBCs manipulating long-term secrets. As for the fault models, we use a simplified version of the ones proposed in [FG20, AOTZ20] that easily translates into interpretable leveled implementation guidelines. Quantitatively, we consider unbounded faults (which can hit any number of intermediate values of the implementation per query) and bounded faults (which can only hit a number of them). Qualitatively, we consider stuck-at and differential faults. Besides, our results are obtained without idealized assumptions for the cryptographic primitives (that are in general questionable with leakage or faults).

Besides, we clarify a few generalities regarding security against side-channel and fault attacks at the mode level. Namely, we first discuss the additional requirements needed to turn fault-resilience (where security can vanish when a fault hits the verification but is restored afterwards) into fault-resistance (where security is always guaranteed). We then

Table 1: Summary of our constructions. The column 'Faults Vrfy' says whether the scheme is secure when the adversary can only inject faults in tag verification. The column 'Faults Mac' says whether the scheme is secure when the adversary can also inject faults in tag generation. The column 'Fault types' describes the type of faults where SaF stands for Stuck-at-Faults, D stands for Differential Faults, and U / B stand for unbounded / bounded number of faults. The column '#TBC' is the number of protected TBC calls.

	Faults Vrfy	Faults Mac	Fault types	# of protected TBCs
LR-MAC1	✓	×	(SaF&DF), U	1
LR-MACd	✓	✓	(SaF&DF), B	2
LR-MACr	✓	✓	DF, U	1

describe how the (quite coarse-grain) model of computation we consider can be made finer-grain for the non-keyed operations. We finally show that the stuck-at and differential fault models are equivalent for deterministic operations when unbounded leakage is available: this observation provides a separation between an implementation that separately provides security against leakage and faults and an implementation that provides security against their combination (and therefore confirms that our unified model is necessary).

We summarize our constructions in Table 1.

Related works. There is a wide body of research on security against leakage. Theoretical approaches have been recently surveyed in [KR19]. The most relevant constructions for our investigation of leveled (symmetric) designs are the follow ups of [PSV15]. Practical approaches considering the secure implementation of countermeasures like masking or shuffling are orthogonal to our concerns but are important for the secure TBC implementation we need (i.e., to fulfill our assumptions). We refer the interested reader to [GR17, CGLS21] for recent examples of masking in software and hardware, respectively. Theoretical attempts to model faults are a bit scarcer. We refer to [GLM+04] for an early result in this direction and to [LL12] for a first treatment of combined attacks. The most relevant models for our investigations are the one of Fischlin and Günther [FG20] and the one of Aranha et al. [AOTZ20] (both contain a comprehensive list of references with other possible abstractions). We slightly simplify them in order to make their interpretation more intuitive and (most importantly) extend them to leakage. Practical approaches considering the secure implementation of countermeasures against faults are orthogonal to our concerns as well. We refer the interested reader to [BBKN12] for an overview and to [IPSW06] and [DN20] for formal attempts to analyze some of them. Eventually, we mention the recent work of Dobraunig and Mennink [DMP20]: their goals are similar to ours but their analysis, based on the quantification of the entropy loss due to side-channels and faults, is finer-grain than ours and so far specialized to ideal permutations. It is an interesting question whether such a finer-grain model can be used to improve or refine the analysis of our constructions and lead to more efficient leveled implementations.

2 Background

Notations. Let $\{0,1\}^n$ be the set of all n-bit strings and by $\{0,1\}^*$ the set of all finite strings. We denote by $x \xleftarrow{\$} \mathcal{X}$ the fact that x is picked up uniformly at random from the set $\mathcal{X}$. We denote by $[c]$ the set of $\{0,1,2,\ldots,c\}$ for some integer c. A $(q_1,...,q_o,t)$-adversary A is a probabilistic algorithm allowed to make q_i queries to oracle O_i and runs in time at most t. By $\mathsf{A}^{\mathsf{O}_1,\ldots,\mathsf{O}_o}(x) \to y$, we denote that the adversary A, on input x and having access to oracles $\mathsf{O}_1,...,\mathsf{O}_o$, outputs y.

2.1 Cryptographic Primitives

Our schemes use two cryptographic primitives, namely Tweakable Block Ciphers (TBCs) and hash functions, which are defined below.

Definition 1. A *tweakable block ciphers* (TBC) $\mathsf{F} : \mathcal{K} \times \mathcal{TW} \times \{0,1\}^n \to \{0,1\}^n$ is a family of permutations, where $\mathsf{F}_k^{tw}(\cdot) = \mathsf{F}(k, tw, \cdot)$ is a permutation over $\{0,1\}^n$. Here the key $k \in \mathcal{K}$ is secret and used to provide the security, and the tweak $tw \in \mathcal{TW}$ is public and used to provide variability.

Definition 2. A hash function $\mathsf{H} : \mathcal{HK} \times \{0,1\}^* \to \{0,1\}^n$ is said to be $(t, \epsilon_{\mathsf{CR}})$-*collision resistant*, if for any t-adversary A, we have

$$\Pr[s \xleftarrow{\$} \mathcal{HK}, (m_0, m_1) \leftarrow \mathsf{A}(s) \mid m_0 \neq m_1, \mathsf{H}_s(m_0) = \mathsf{H}_s(m_1)] \leq \epsilon_{\mathsf{CR}}.$$

2.2 Message Authentication Code

A Message Authentication Code (MAC) is a symmetric-key scheme aimed to provide data authenticity. We next provide is definition.

Definition 3. A MAC scheme Π is a triplet of algorithms $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ where:

- Gen. The key-generation algorithm Gen generates k which is usually picked uniformly at random over the key space $\mathcal{K}$;
- Mac. The tag-generation algorithm Mac takes as input a key $k \in \mathcal{K}$ and a message $m \in \{0,1\}^*$, and outputs a tag $\tau \in \mathcal{TAG}$.
- Vrfy. The verification algorithm Vrfy takes as input a key $k \in \mathcal{K}$, a message $m \in \{0,1\}^*$ and a tag $\tau \in \mathcal{TAG}$, and outputs either 1 *(accept)* or 0 *(reject)*.

We require *correctness*: $\forall (k, m) \in \mathcal{K} \times \{0,1\}^*,\ \mathsf{Vrfy}(k, m, \mathsf{Mac}(k, m)) = 1$.

2.3 Security in the Presence of Leakage

When an adversary has access not only to the outputs of an oracle but also to its leakage, we denote her as $\mathsf{A}^{\mathsf{L_O}}$. In this case, on input x, the leaking oracle $\mathsf{L_O}$ returns $y = \mathsf{O}(x)$ and the leakage $\mathsf{l_o} := \mathsf{L_O}(x)$. If the oracle has a key k, then we write the leakage function as $\mathsf{L_O}(x; k)$. Adversaries are sometimes allowed to "model" the leakage as in the case of profiled side-channel attacks [CRR02]. Hence, we grant them oracle access to L. This oracle allows the adversary to make queries on inputs x and keys k' of her choice.

Strong Unforgeability with Leakage (SUF−L2). We consider the security of MACs in the presence of leakage. Informally, it should be hard for the adversary to forge a tag even having access to the leakage of the tag-generation and the verification algorithms. That is, to find a fresh and valid pair of message and tag (m, τ) such that $\mathsf{Vrfy}_k(m, \tau) = \top$. We use the SUF−L2 definition by Berti et al. [BGP+19] for this purpose.

Definition 4 (SUF−L2)**.** A $\mathsf{MAC} = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ with tag-generation leakage function $\mathsf{L_{Mac}}$ and verification leakage function $\mathsf{L_{Vrfy}}$ is $(q_L, q_M, q_V, t, \epsilon)$-strongly existentially unforgeable leakage resistant in both tag-generation and verification against chosen-message attacks if for all (q_L, q_M, q_V, t)-adversaries A, for $\mathsf{L} = (\mathsf{L_{Mac}}, \mathsf{L_{Vrfy}})$, we have

$$\Pr\left[\mathsf{FORGE\text{-}L2}^{\mathsf{suf\text{-}vcma}}_{\mathsf{MAC,L,A}} \Rightarrow 1\right] \leq \epsilon,$$

where the $\mathsf{FORGE\text{-}L2}^{\mathsf{suf\text{-}vcma}}_{\mathsf{MAC,L,A}}$ experiment is defined in Table 2.[1]

[1] It is a natural extension of the standard unforgeability definition (without leakage), which is $\mathsf{FORGE\text{-}L2}^{\mathsf{suf\text{-}vcma}}$ where the leakage functions L_M and L_V do not output anything.

Table 2: The FORGE-L2$^{\text{suf-vcma}}$ experiment.

The FORGE-L2$^{\text{suf-vcma}}_{\text{MAC,L,A}}$ experiment	
Initialization: $k \leftarrow \text{Gen}$ $\mathcal{S} \leftarrow \emptyset$ Finalization: $(m, \tau) \leftarrow \text{A}^{\text{L,LMac,LVrfy}}$ If $(m, \tau) \in \mathcal{S}$ return 0 Return $\text{Vrfy}_k(m, \tau)$	Oracle $\text{LMac}(m)$: $(\tau, \text{l}_{\text{mac}}) = \text{LMac}_k(m)$ $\mathcal{S} \leftarrow \mathcal{S} \cup \{(m, \tau)\}$ Return $(\tau, \text{l}_{\text{mac}})$ Oracle $\text{LVrfy}(m, \tau)$: Return $\text{LVrfy}_k(m, \tau)$

For simplicity, we will denote the final output of the adversary as the $(q_V + 1)$th verification query in the rest of the paper.

The Unbounded Leakage Model. In the unbounded leakage model [BKP+18], the leakage function reveals all the internal states produced during the execution of the scheme, except the ones of the strongly protected components used to manipulate long-term secret keys. This model is based on the observation that in order to implement a leakage-resilient cryptographic scheme, it is sometimes possible to let most of its underlying building blocks leak in an unrestricted manner, and to only protect some sensitive computations strongly. More precisely, in the unbounded leakage model, the building blocks are divided in:

- Unprotected building blocks that fully leak their inputs, outputs and keys;
- Strongly protected building blocks that leak their inputs and outputs in full, and only leak their keys in a strongly restricted manner.

For simplicity, the strongly protected component is sometimes modeled as leak-free. In this paper, we rather require the (weaker, non-idealized and falsifiable) assumption that it ensure strong unpredictability with leakage.

Strong unpredictability with leakage (SUP–L2). Unpredictability is among the simplest requirements for TBCs. It is appealing in leakage-resilient cryptography since it can be tested by an evaluation laboratory. We consider the strong unpredictability of TBCs with leakage. Intuitively, it says that it is hard for the adversary to find a fresh and valid triplet (tw, x, y) such that $y = \text{F}_k^{tw}(x)$ even with access to the leakage associated to the implementation of the TBC. We recall the SUP–L2 definition by Berti et al. [BGP+19].

Definition 5 (SUP–L2)**.** A tweakable block cipher $\text{F} : \mathcal{K} \times \mathcal{TW} \times \{0,1\}^n \rightarrow \{0,1\}^n$ with leakage function pair $\text{L} = (\text{L}_{\text{Eval}}, \text{L}_{\text{Inv}})$ is $(q_{\text{L}}, q_E, q_I, t, \epsilon)$-*strongly unpredictable with leakage in evaluation and inversion* (SUP–L2), or $(q_{\text{L}}, q_E, q_I, t, \epsilon)$-SUP–L2, if for any $(q_{\text{L}}, q_E, q_I, t)$-adversary A, we have

$$\Pr[\text{SUP–L2}_{\text{A,F,L}} \Rightarrow 1] \leq \epsilon,$$

where the SUP–L2 experiment is defined in Table 3.

2.4 The LR-MAC1 Construction

Finally, LR-MAC1 is a leakage-resilient MAC with SUF–L2 security in the unbounded leakage model, assuming a collision-resistant hash function and a SUP–L2 secure TBC [BGPS21]. It improves over HBC [BPPS17] by avoiding the difficult interaction between the hash function and the TBC. The code description and figure of LR-MAC1 are illustrated in Algorithm 1 and Figure 1 respectively.

Table 3: Strong unpredictability with leakage in evaluation & inversion.

The $\mathsf{SUP\text{-}L2}_{\mathsf{A,F,L}}$ experiment	
Initialization: $k \xleftarrow{\$} \mathcal{K}$ $\mathcal{L} \leftarrow \emptyset$ Finalization: $(x, tw, z) \leftarrow \mathsf{A}^{\mathsf{L},\mathsf{LEval},\mathsf{LInv}}$ If $(x, tw, z) \in \mathcal{L}$ Return 0 If $z = \mathsf{F}_k(tw, x)$ Return 1 Return 0	Oracle $\mathsf{LEval}(tw, x)$: $z = \mathsf{F}_k(tw, x)$ $\mathsf{l_e} = \mathsf{L}_{\mathsf{Eval}}(tw, x; k)$ $\mathcal{L} \leftarrow \mathcal{L} \cup \{(x, tw, z)\}$ Return $(z, \mathsf{l_e})$ Oracle $\mathsf{LInv}(tw, z)$: $x = \mathsf{F}_k^{-1}(tw, z)$ $\mathsf{l_i} = \mathsf{L}_{\mathsf{Inv}}(tw, z; k)$ $\mathcal{L} \leftarrow \mathcal{L} \cup \{(x, tw, z)\}$ Return $(x, \mathsf{l_i})$

Algorithm 1 The LR-MAC1 algorithm.

It uses a strongly protected TBC $\mathsf{F} : \mathcal{K} \times \mathcal{TW} \times \{0,1\}^n \to \{0,1\}^n$ and a hash function $\mathsf{H} : \mathcal{HK} \times \{0,1\}^* \to \mathcal{TW}$.

Gen:

- $k \xleftarrow{\$} \mathcal{K}$
- $s \xleftarrow{\$} \mathcal{HK}$

$\mathsf{Mac}_k(m)$:

- $h = \mathsf{H}_s(m)$
- $\tau = \mathsf{F}_k^h(0^n)$
- Return τ

$\mathsf{Vrfy}_k(m, \tau)$:

- $h = \mathsf{H}_s(m)$
- $\tilde{x} = \mathsf{F}_k^{h,-1}(\tau)$
- If $\tilde{x} == 0^n$ Return 1
- Else Return 0

3 Modeling Fault and Leakage

We now give a general model to capture Fault-then-Leak (FL) attacks, where the adversary can inject faults on any ephemeral value during the computation and observe these values thanks to leakage (for the same query). Injecting faults is adaptive through the different faulting-and-leaking queries. Our way to capture faults on the (ephemeral) values unifies both persistent memory faults (as if the fault occurs in the memory and overwrites the correct input once and for all) and transient faults (as if the fault only occurs during a chosen computation). We then apply our model to the case of MACs, and give the first experiment formally capturing strong unforgeability against chosen-message attacks with faults-then-leakage in tag generation and verification.

3.1 Faulty Matrix & Atomic Computation Model

Let $(f_1, \ldots, f_m)$ be an implementation of a cryptographic algorithm Algo_k, where k is a key viewed as a parameter encoded in (some of) the functions f_j's.

By this, if we write $\mathsf{Algo}_k(x) = y$ with input $x = (x_1, \ldots, x_n)$, we mean that the

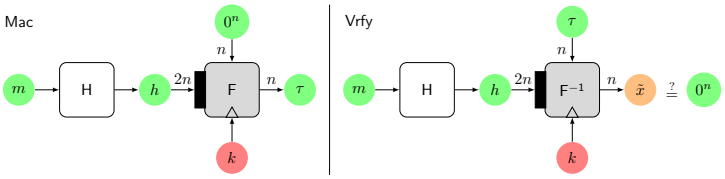

Figure 1: LR-MAC1.

following sequence of computations:

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= y_1, \\ f_2(x_1, x_2, \ldots, x_n, y_1) &= y_2, \\ &\vdots \\ f_m(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_{m-1}) &= y_m, \end{aligned}$$

return $\mathsf{Select}(y_1, \ldots, y_m) = y$, where Select is a deterministic function independent of k which simply selects those of the y_j's (that are the outputs of the f_j's) that are included in y. For simplicity we do not include Select in the sequence $(f_1, \ldots, f_m)$, but formally, Select is part of the implementation.

The definition of an implementation $(f_1, \ldots, f_m)$ is general and covers cases where, for instance, f_1 does not depend on x_3 or $y_3 = k$. It might also be that f_1 and f_2 can be run in parallel if f_2 does not depend on y_2, and so on and so forth. That is, the ordered sequence of the functions f_j's does not strictly force the computation to be sequential and, in that sense, it does not fully capture the time. Nevertheless, we need to capture the dependencies between the functions f_j's and their inputs $(x_1, \ldots, x_n, y_1, \ldots, y_{j-1})$. In other words, we want to know the inputs that are "really used" by the functions.

We capture the dependency on the inputs of the functions f_j's by replacing all the components that are not used by the empty string ε. This way, we can represent the inputs of all the f_j's by the $m \times (n + m - 1)$ matrix below:

$$\begin{pmatrix} \tilde{x}_{11} & \tilde{x}_{12} & \cdots & \tilde{x}_{1n} & \varepsilon & \varepsilon & \cdots & \varepsilon \\ \tilde{x}_{21} & \tilde{x}_{22} & \cdots & \tilde{x}_{2n} & \tilde{y}_{21} & \varepsilon & \cdots & \varepsilon \\ \vdots & & & \vdots & & & \ddots & \vdots \\ \tilde{x}_{m1} & \tilde{x}_{m2} & \cdots & \tilde{x}_{mn} & \tilde{y}_{m1} & \tilde{y}_{m2} & \cdots & \tilde{y}_{m\,m-1} \end{pmatrix}$$

where the j-th line corresponds to f_j's inputs so that $\tilde{x}_{ji} = x_i$ if f_j depends on x_i and $\tilde{x}_{ji} = \varepsilon$ otherwise, and $\tilde{y}_{ji} = y_i$ if f_j depends on the f_i's output y_i and $\tilde{y}_{ji} = \varepsilon$ otherwise. As a result, each column contains at most 2 distinct values which, in the case of the first column, are x_1 and ε. Obviously, we always have $\tilde{y}_{11} = \cdots = \tilde{y}_{1\,m-1} = \varepsilon$ as f_1 depends on none of the $y_1, \ldots, y_{m-1}$. In the rest of the paper, we call this matrix the **dependency matrix** of the implementation $(f_1, \ldots, f_m)$ of Algo_k.

The dependency matrix offers an easy way to model all the ephemeral values that each step of the computation of Algo_k actually requires and those that are useless. It becomes simple to see all the "active" inputs of the different functions f_j's and where an adversary can provide an effect by injecting a fault (i.e., on any entry distinct of ε). For example, if an adversary wants to inject a persistent memory fault on x_1 and replaces this value by x'_1, it corresponds to a faulty matrix containing all the ε's at the same place of the dependency matrix's first column, and filling the remaining places with x'_1. If the adversary also wants (in the same query to Algo_k) to inject a persistent fault on y_2, even if y_2 is unknown (since

the secret k can be encoded in f_2), it suffices to fill the $(n+2)$-th column with the desired y'_2. Besides, if y_2 still appears in at least 2 positions of its column in the dependency matrix, the adversary can inject different transient faults for each occurrence, and we then fill the faulty matrix accordingly. This way, each query with chosen input $x = (x_1, \ldots, x_n)$ also comes with a **faulty matrix** indicating when and where some correct values during the computation of $\mathsf{Algo}_k(x)$ must be replaced and by which precise other chosen values. Places that remain empty (even not containing the empty string) simply indicate that the corresponding values will not be faulted during the computation.

All the possible faults that an adversary can inject during the computation of $\mathsf{Algo}_k(x)$ are thus induced by the dependency matrix, and therefore by the implementation $(f_1, \ldots, f_m)$. Clearly, another implementation $(f'_1, \ldots, f'_{m'})$ of Algo_k leads to an a-priori distinct fault model, as its faulty matrix can represent other type of inputs where the adversary can inject faults. This fact captures the inherent dependency of fault attacks on the implementation. Moreover, given the $(f_1, \ldots, f_m)$, we assume that choosing the faulty matrix is the best that the adversary can do. Therefore, the faulty matrix also models the adversary's ability to inject faults in the sense that we assume that the implementation makes it unfeasible to inject any other kind of faults in the computation.

In other words, $(f_1, \ldots, f_m)$ also models the power of the adversary. For instance, if f_1 represents the implementation of a hash function H_s with public parameter s, it means that the adversary can only introduce a fault on its inputs. Implicitly, it says that even if H_s is computed by iterating a compression function, the implementation must protect these iterations making the adversary unable to introduce a chosen fault in the middle. In that sense, we call our model *atomic* and denote the f_j's as *atoms* (or atomic components) that cannot be split and exploited by the adversary.

This atomic model leaves the opportunity to be finer or coarser grain. As a first step, this paper will consider mode-level security against faults. In this case, we see the cryptographic building blocks as atoms. As will be clear next, this coarse-grain modeling already allows paving the way towards leveled implementations against combined attacks mixing leakage and faults. But as mentioned in introduction, investigating whether a finer-grain modeling (e.g., at the level of the compression of a hash function or a TBC's rounds) would allow improving our results is an interesting direction for further research. The important asset of our model is that it would directly allow such advanced studies.

3.2 Protected Computations & Types of Faults

Let us assume that the atomic implementation (f_1, f_2, f_3) of Algo_k has the following dependency matrix on input $x = (x_1, x_2)$:

$$\begin{pmatrix} x_1 & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & x_2 & \varepsilon & \varepsilon \\ x_1 & \varepsilon & y_1 & y_2 \end{pmatrix},$$

which already says that f_1 and f_2 can be computed in parallel as f_2 does not depend on y_1, and that injecting a faulty x'_2 on the only place where x_2 is involved is useless as it comes to make the query $x' = (x_1, x'_2)$ to Algo_k without fault. An admissible faulty matrix for the query (x_1, x_2) can be given by:

$$\begin{pmatrix} x'_1 & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \cdot & \varepsilon & \varepsilon \\ \cdot & \varepsilon & \cdot & y'_2 \end{pmatrix},$$

meaning that $f_1(x'_1) = y'_1$, $f_2(x_2) = y_2$ and $f_3(x_1, y'_1, y'_2)$ are computed to answer to the query with $y = \mathsf{Select}(y'_1, y_2, y'_3)$. We stress that the computation of f_2 remains honest

and even if the third line only contains a faulty y'_2, f_3 has to take the output of f_1 without further fault (with y'_1 a faulty input due to x'_1).

As in the leakage setting, letting the designer fault all the atoms of an implementation makes it impossible to reach security. Therefore we additionally model protected computations. Concretely, they correspond to requirements to implementers (i.e., they indicate where countermeasures against faults must be deployed). Of course, our goal is to design modes such that not all the atomic computations must be protected, which is the essence of leveled implementations. For instance, we might want to protect y_1 of the dependency matrix. We then simply model this protection by indicating that it is forbidden in the faulty query to inject a fault at that place, by replacing the corresponding occurrence by $\perp$. We stress again that this modeling does not mean that we (arbitrarily) prevent adversaries to try injecting a fault at that place and time of the computation. The symbol $\perp$ rather means that the corresponding protected input should be fault-immune. In other words, it is an assumption on the implementation, not a restriction of the adversary. In the case we want to protect y_1, we then get the following **protected dependency matrix**:

$$\begin{pmatrix} x_1 & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & x_2 & \varepsilon & \varepsilon \\ x_1 & \varepsilon & \perp & y_2 \end{pmatrix}.$$

As a result, the faulty matrix

$$\begin{pmatrix} x'_1 & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \cdot & \varepsilon & \varepsilon \\ \cdot & \varepsilon & \perp & y'_2 \end{pmatrix}$$

is still admissible and equivalent to the previous attack, since the computation of $f_3(x_1, y'_1, y'_2)$ relies on a previous faulty output y'_1 and not on a fresh injection of a fault happening during the computation of f_3 in its y_1-component.

Note that if f_i is a public function, there is no difference in faulting one of the input or the output (see Sec. 4.2); on the other hand when the secret is encoded in f_i, it is different to fault one of the inputs of f_i or the output. Moreover, for the adversary it may be useful to see, via leakage, the output of f_i with inputs of her choice.

The faulty matrix allows indicating where the adversary wants to introduce faults. It also allows determining the type of faults. We consider two models for this purpose. In the first **stuck-at** fault model, the adversary has full control on the values she can inject. Precisely, she can replace any number of (possibly all the) bits of the target intermediate value by bits of her choice.[2] In the second **differential** fault model, the adversary can only inject a difference on the target intermediate value (i.e., XORing it with a chosen value). In our example above, an admissible differential faulty matrix for $x = (x_1, x_2)$ is

$$\begin{pmatrix} z_1 & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \cdot & \varepsilon & \varepsilon \\ \cdot & \varepsilon & \perp & z_2 \end{pmatrix}_{\Delta},$$

meaning that $f_1(x_1 \oplus z_1) = y'_1$, $f_2(x_2) = y_2$ and $f_3(x_1, y'_1, y'_2 \oplus z_2)$ are computed to answer to the query with $y = \mathsf{Select}(y'_1, y_2, y'_3)$, assuming that $\oplus$ is clear from the context (e.g., it can be the XOR for some inputs and another group law for others). The Δ subscript indicates that we consider differential faults (matrices without Δ denote stuck-at faults). Quite naturally, this model can be refined, for example by assuming that certain atoms can be hit by stuck-at faults and others by differential faults (in the latter case, a Δ subscript can be added for all the elements of the faulty matrix that are differentially faulted) or even that this versatility takes place at the bit level (using similar notations).

[2] For simplicity, we assume that when injecting a stuck-at fault x', some of the bits of x' can be set to $\emptyset$ and the corresponding bits of x are therefore not faulted.

3.3 Fault-then-Leak Attacks & (Un)Bounded Injections

Additionally to modeling faults, we also want to capture the information that an adversary can learn from the leakage of a faulty query. Since the leakage generated by the computation of Algo_k also depends on the implementation $(f_1, \ldots, f_m)$, we associate to each function f_j a leakage function L_{f_j}, or simply L_j for short. Given this tuple of leakage functions $\mathsf{L}_{\mathsf{Algo}} = (\mathsf{L}_1, \ldots, \mathsf{L}_m)$, we usually write $\mathsf{LAlgo}_k(x)$ to indicate the computation of $y = \mathsf{Algo}_k(x)$ with the associated leakage trace $\mathsf{l}_{\mathsf{algo}} = \mathsf{L}_{\mathsf{Algo}}(k; x)$ so that $\mathsf{LAlgo}_k(x) = (y, \mathsf{l}_{\mathsf{algo}})$.

We recall that each leaking atomic computation (f_j, L_j) may also depend on k and other (public) parameters of the black-box algorithm that are only implicit here. If f_j contains k as a parameter, so does L_j which takes the same inputs as f_j. In a security proof we will often make this dependence explicit in the notation and we can write $\mathsf{L}_j(k; \vec{x}_j)$, where $\vec{x}_j$ is the j-th input row of f_j in the dependency matrix, if f_j uses k as a parameter. In practice, if the adversary injects a fault during the computation of f_j, the leakage she can observe depends on the same faulty inputs. Therefore, the use of the (differential) faulty matrix remains unambiguous and naturally extends to $\mathsf{L}_{\mathsf{Algo}}(k; \cdot)$.

In summary, given a query x to LAlgo_k with an admissible (possibly differential) faulty matrix, we capture both the computation of Algo_k on a faulty input and the leakage resulting from the computation obtained with the same chosen fault injection. That means that we model the situation where the adversary knows the (stuck-at or differential) faults she wants to inject prior to each leaking query. We denote such a model as the **fault-then-leak** one. It assumes that the adversary does not have the time to first observe the start of a leaking computation on a chosen input and to inject a fault of which the value is adapted on-the-fly depending on this leakage. The latter seems to capture the reality of accurate fault insertions, which require careful setup manipulations that are not instantaneous. A more powerful and possibly unrealistic leak-then-fault model could be considered as a scope for further research. Besides, we note that the queries are adaptive and therefore, the adversary can make a fault-then-leak query that depends on its current view that includes the leakage of all the previous faulty queries.

Eventually, a meaningful way to restrict the fault adversary is to assume that faulting more intermediate computations (hence more bits) per query is increasingly difficult. In other words, given an implementation $(f_1, \ldots, f_m)$, it can be difficult to fill all the possible entries of an admissible faulty matrix in a single query. We thus differentiate the case of **unbounded faults**, where the adversary is able to inject an unbounded number of faulty inputs in any leaking query, and the case of **ℓ-bounded faults** where she can only inject at most ℓ faulty inputs in each query (thus in any faulty matrix). We note that the bound on the amount of faults is defined at the level of intermediate values, as per our atomic model. Yet, it is easy to translate into a maximum number of bits to fault by looking at the size of these intermediate values. Here again, the model could be refined later on, by allowing that at most a fraction of the bits of an intermediate value can be faulted, which we leave as another interesting scope for further research.

3.4 Faulty Leaky Algorithm & Valid Query

Given a leaking implementation $(f_1, \ldots, f_m)$ of Algo_k with leakage function $\mathsf{L}_{\mathsf{Algo}} = (\mathsf{L}_1, \ldots, \mathsf{L}_m)$, we would like to extend the notation of LAlgo_k to deal with faulty inputs. First, let $\mathcal{F}$ be the empty faulty matrix associated to the (protected) dependency matrix. That is, $\mathcal{F}$ represents a faulty matrix with no fault to inject. Now, we observe that there is a canonical representation between any faulty matrix and the tuple of the faulty inputs that fills $\mathcal{F}$ in the reading direction and gives back the given faulty matrix. If z denotes

this faulty tuple, we write $\mathcal{F}(z)$ the corresponding faulty matrix. For instance:

$$\mathcal{F} = \begin{pmatrix} \cdot & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \cdot & \varepsilon & \varepsilon \\ \cdot & \varepsilon & \perp & \cdot \end{pmatrix}, \qquad \mathcal{F}(x'_1, \cdot, \cdot, y'_2) = \begin{pmatrix} x'_1 & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \cdot & \varepsilon & \varepsilon \\ \cdot & \varepsilon & \perp & y'_2 \end{pmatrix},$$

where we extend $\mathcal{F}$ as a function of the faulty tuples to the corresponding faulty matrix. Here, $z = (x'_1, \cdot, \cdot, y'_2)$. As a result, we can extend LAlgo_k into a faulty & leaky algorithm so that $\mathsf{LAlgo}_k(x, z)$ means the leaking computation of $\mathsf{Algo}_k(x)$ with respect to the fault injection represented by the faulty matrix $\mathcal{F}(z)$.

Moreover, we say that the query (x, z) is **valid** if for all x_i of $x = (x_1, \ldots, x_n)$, x_i is not replaced by a persistent faulty x'_i, i.e., the i-th column of $\mathcal{F}(z)$ does not only contain x'_i and possibly ε. Otherwise, the query is equivalent to (x', z'), where $x' = (x_1, \ldots, x_{i-1}, x'_i, x_{i+1}, \ldots, x_n)$ and the i-th column of $\mathcal{F}(z')$ only contains "$\cdot$" and possibly ε. The reason we reject such kind of queries is to facilitate the description of the winning condition in a security experiment since all valid queries do not trivially correspond to other valid queries with distinct x-inputs. We stress that this restriction does not limit the fault capabilities of the adversary, as she can then simply try to make a query for another input x'_i instead of x_i, while the original input x_i is never manipulated during the computation (which does not require faults). Eventually, we simply define $\mathcal{F}(x, z)$ as the predicate telling whether the input (x, z) is valid or not.

3.5 The MAC Case

We give the first security notion for MAC against faults-then-leak attacks. Starting with the usual (strong) unforgeability against chosen message attacks, we augment the adversary's capabilities by turning the tag generation and verification oracles into their faulty and leaking variants FLMac and FLVrfy. Let $\mathsf{MAC} = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ be a MAC scheme with Mac implementation $(f_1, \ldots, f_m)$ with n inputs and Vrfy implementation $(g_1, \ldots, g_{m'})$ with n' inputs. As a result, the leaking function pair $\mathsf{L} = (\mathsf{L}_{\mathsf{Mac}}, \mathsf{L}_{\mathsf{Vrfy}})$ is well defined. Let also $\mathcal{I}_{\mathsf{Mac}}$ and $\mathcal{I}_{\mathsf{Vrfy}}$ be the set of double indexes (j, i) where the MAC implementation requires protections in the dependency matrices of Mac and Vrfy. Hence, the faulty matrix functions $\mathcal{F}_{\mathsf{Mac}}$ and $\mathcal{F}_{\mathsf{Vrfy}}$ (and predicates) are also well defined.

Definition 6 ($\mathsf{SUF\text{–}FL2}$)**.** A $(\mathcal{I}_{\mathsf{Mac}}, \mathcal{I}_{\mathsf{Vrfy}})$-protected message authentication code $\mathsf{MAC} = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ with leaking function pair $\mathsf{L} = (\mathsf{L}_{\mathsf{Mac}}, \mathsf{L}_{\mathsf{Vrfy}})$ and faulty injection pair $\mathcal{F} = (\mathcal{F}_{\mathsf{Mac}}, \mathcal{F}_{\mathsf{Vrfy}})$ is $(q_{FL}, q_M, q_V, t, \epsilon)$-strongly existentially unforgeable against stuck-at (*resp.-1* differential) unbounded (*resp.-2* ℓ-bounded) fault-then-leak attacks in tag-generation and verification if for all (q_{FL}, q_M, q_V, t)-adversaries $\mathsf{A}^{\mathsf{L},\mathcal{F}}$, we have

$$\Pr\left[\mathsf{FORGE\text{-}FL2}^{\mathsf{suf\text{-}vcma}}_{\mathsf{MAC},\mathcal{F},\mathsf{L},\mathsf{A}} \Rightarrow 1\right] \leq \epsilon,$$

where the $\mathsf{FORGE\text{-}FL2}^{\mathsf{suf\text{-}vcma}}_{\mathsf{MAC},\mathcal{F},\mathsf{L},\mathsf{A}}$ experiment is defined in Table 4.

This definition can be weakened by allowing fault injection either in Mac or in Vrfy, in which case we simply say that the MAC is $\mathsf{SUF\text{–}FL1}$ with respect to either the tag generation or the verification. The above security definition could also be extended to cross-type attacks where the adversary can mix stuck-at and differential faults in each query, following our description on Section 3.2.

4 Warming up

Before investigating the security of concrete MAC constructions, we discuss a few generalities that can help interpreting the impact of our results.

Table 4: The FORGE-FL2$^{\text{suf-vcma}}$ experiment. In stuck-at (resp., differential) attacks, $\mathcal{F}_{\mathsf{Mac}}$ and $\mathcal{F}_{\mathsf{Vrfy}}$ represent the stuck-at (resp., differential) faulty matrices functions and predicates. In the ℓ-bounded case, the predicates $\mathcal{F}_{\mathsf{Mac}}$ and $\mathcal{F}_{\mathsf{Vrfy}}$ return 0 if the fault injection tuple z contains more than ℓ components $\neq$ "$\cdot$".

The FORGE-FL2$^{\text{suf-vcma}}_{\mathsf{MAC},\mathcal{F},\mathsf{L},\mathsf{A}}$ experiment	
Initialization: $k \leftarrow \mathsf{Gen}$ $\mathcal{S} \leftarrow \emptyset$ Finalization: $(m,\tau) \leftarrow \mathsf{A}^{\mathsf{L},\mathcal{F},\mathsf{FLMac},\mathsf{FLVrfy}}$ If $(m,\tau) \in \mathcal{S}$, return 0 Return $\mathsf{Vrfy}_k(m,\tau)$	Oracle $\mathsf{FLMac}(m,z)$: If $\mathcal{F}_{\mathsf{Mac}}(m,z) = 0$, return $\perp$ $(\tau, \mathsf{leak}) = \mathsf{LMac}_k(m,z)$ $\mathcal{S} \leftarrow \mathcal{S} \cup \{(m,\tau)\}$ Return (τ, leak) Oracle $\mathsf{FLVrfy}(m,\tau,z)$: If $\mathcal{F}_{\mathsf{Vrfy}}((m,\tau),z) = 0$, return $\perp$ Return $\mathsf{LVrfy}_k(m,\tau,z)$

4.1 Fault-Resilience vs. Fault-Resistance

As in the leakage setting, we denote as resilient an implementation such that security guarantees vanish in the presence of faults but are restored afterwards, and we call resistant an implementation where these guarantees are always maintained. It is easy to see that fault-resistance requires some fault-immune computations since an adversary can always hit a verification with some trivial attacks during the finalization step of the unforgeability experiment. For example in the case of LR-MAC1 of Figure 1, she could replace $\tilde{x}$ by zero or hit the reject symbol 0 to turn it into the accept symbol 1. Since these attacks are generic and independent of the target cryptographic constructions, our focus in the following sections is mostly on fault-resilience (with the admitted cautionary note that fault-resistance requires a fault-immune verification step).

4.2 Sub-atomic Faults for Publicly Computable Functions

Our following investigations are mode-level, meaning that we consider a quite coarse-grain version of the atomic model where atoms are cryptographic primitives. Yet, it is interesting to note that for *publicly computable functions*, the physical security guarantees we obtain hold even if the corresponding atoms are implemented with the finest (gate-level) granularity. By publicly computable, we mean functions that do not encode any secret key and for which no input is random. In this case, for any (even fine-grain) error, it is always possible to simulate it by just observing its impact on the output (which can be done since the function is publicly computable) and reporting it as a coarse-grain error. This for example implies the useful observation that the hash function of LR-MAC1 does not require any protection of its internal computations against faults.

4.3 Interpreting Fault Immunity

The model of Section 3 assumes that the long-term key is always fault-immune (as it is encoded in the functions). Yet, how this requirement translates into implementation guidelines depends on the type of faults that can be inserted (and possibly the type of primitive considered). In our following treatment of MACs, fault-immunity will only be encountered for the TBC keys. In case stuck-at faults are possible, generic attacks that target the key bit by bit are possible. So fault immunity can only translate into a physical assumption. Essentially, we then require that the TBC remains unpredictable even if the key is faulted. In case only differential faults are possible, fault immunity can

also translate into a mathematical assumption, namely that the TBC is secure against related-key attacks. Since these concerns are quite independent of the MAC constructions, we will not re-discuss them systematically and just mention the requirement that the long-term key is fault immune in our theorem statements.

4.4 Model Equivalence for Deterministic Operations

Eventually, we note that for deterministic operations, the combination of a differential fault with unbounded leakage provides the adversary with the possibility to emulate a stuck-at fault x' by first observing the unbounded leakage of the intermediate value x she wants to target and then injecting a differential fault corresponding to $x \oplus x'$. This observation is interesting since it provides a separation between an implementation that separately provides security against leakage and faults (in which case stuck-at and differential faults are always different) and an implementation that provides security against their combination (in which case both models can be identical for deterministic operations).

5 LR-MAC1 against leakage and faults

We now prove that LR-MAC1 (illustrated in Figure 1) is secure against leakage and faults in its tag verification and exhibit attacks against other MACs in this context. We also show that LR-MAC1 cannot resist stuck-at nor differential faults in its tag generation.

5.1 Secure Verification

The security of LR-MAC1 against leakage and faults (in tag verification) is formalized by the following theorem (see the experiment in Table 4).

Theorem 1. *Let* H *be a* $(t_1, \epsilon_{\mathsf{CR}})$*-collision resistant hash function. Let* F *be a* $(q_{FL}, q_M, q_V, t_2, \epsilon_{\mathsf{SUP\text{-}L2}})$-$\mathsf{SUP}$-$\mathsf{L2}$ *tweakable block cipher with fault-immune long-term key. Then we show that for any* (q_{FL}, q_M, q_V, t)*-adversary* $\mathsf{A}^{\mathsf{L},\mathcal{F}}$ *with (unbounded) leaking function pair* $\mathsf{L} = (\mathsf{L}_{\mathsf{Mac}}, \mathsf{L}_{\mathsf{Vrfy}})$ *and mode-level faulty injection function* $\mathcal{F}_{\mathsf{Vrfy}}$ *in tag verification, LR-MAC1 is* $(q_{FL}, q_M, q_V, t, \epsilon)$*-strongly existentially unforgeable against both stuck-at and differential unbounded fault-then-leak attacks in tag verification, with*

$$\epsilon \leq \epsilon_{\mathsf{CR}} + (q_V + 1)\epsilon_{\mathsf{SUP\text{-}L2}},$$

where $t_1 = t + (q_M + q_V + q_{FL} + 1)t_{\mathsf{H}} + (q_M + q_V + q_{\mathsf{L}})(t_{\mathsf{F}} + t_{\mathsf{L}})$ *and* $t_2 = t + (q_M + q_V + q_{FL} + 1)t_{\mathsf{H}}$.

Faulty matrix and leakage function. Before the formal proof, we first specify the mode-level faulty function $\mathcal{F}_{\mathsf{Vrfy}}$ and (unbounded) leaking function pair $\mathsf{L} = (\mathsf{L}_{\mathsf{Mac}}, \mathsf{L}_{\mathsf{Vrfy}})$. For LR-MAC1, we consider faults only in the tag verification algorithms Vrfy_k, and the mode-level atomic implementation is $f_1 = \mathsf{H}_s(\cdot)$ and $f_2(\cdot) = \mathsf{F}_k^{-1}(\cdot, \cdot)$. We recall, in particular, that it means that the adversary is unable to modify the parameter s of the hash function H. For input $(x_1, x_2) = (m, \tau)$, we thus have $x_1 = m \in \{0,1\}^*$, $x_2 = \tau \in \{0,1\}^n$, $y_1 = \mathsf{H}_s(x_1)$ and $y_2 = \mathsf{F}_k^{-1}(y_1, \tau)$. We stress that we normally have to capture the check $0 == y_2$ with a third function defined as $f_3(\cdot) = [0 == \cdot]$. However, as the check is not protected against leakage, we simply give y_2 in the leakage trace of $\mathsf{L}_{\mathsf{Vrfy}}$ as in the unbounded leakage model all mode-level unprotected intermediate values leak. Obviously, knowing y_2 the adversary will learn nothing more by injecting a fault on that value during the check.

The dependency matrix and the empty faulty matrix are then given by

$$\begin{pmatrix} x_1 & \varepsilon & \varepsilon \\ \varepsilon & x_2 & y_1 \end{pmatrix}, \qquad \mathcal{F}_{\mathsf{Vrfy}} = \begin{pmatrix} \cdot & \varepsilon & \varepsilon \\ \varepsilon & \cdot & \cdot \end{pmatrix}.$$

We require no additional protection for Vrfy_k and $\mathcal{I}_{\mathsf{Vrfy}} = \emptyset$ is not included in the theorem statement as it does not restrict $\mathcal{F}_{\mathsf{Vrfy}}$. If $\mathsf{L}_{\mathsf{F}} = (\mathsf{L}_{\mathsf{Eval}}, \mathsf{L}_{\mathsf{Inv}})$ is the leakage function pair of the TBC F, we have $\mathsf{L}_{\mathsf{Mac}} = \mathsf{L}_{\mathsf{Eval}}$ as well as $\mathsf{L}_{\mathsf{Vrfy}} = (\mathsf{L}_{\mathsf{Inv}}, y_2)$, since L_{H} gives no more information than H does. Therefore, a faulty leaky verification query has the form $\mathsf{FLVrfy}_k(m, \tau, (z_1, z_2, z_3))$, where the function corresponding to faulty matrix is

$$\mathcal{F}_{\mathsf{Vrfy}} = \begin{pmatrix} z_1 & \varepsilon, & \varepsilon \\ \varepsilon & z_2 & z_3 \end{pmatrix}.$$

Hence $(m, \tau, (z_1, z_2, z_3))$ is a valid verification query if and only if $z_1 = z_2 = \cdot$ as otherwise it comes to trivially replace m and τ by z_1 and z_2 respectively, which is the same as the query $(z_1, z_2, (\cdot, \cdot, z_3))$. That is, $\mathcal{F}_{\mathsf{Vrfy}}(m, (z_1, z_2, z_3)) = 1$ if and only if $z_1 = z_2 = \cdot$. For simplicity, we assume that there is only a single possible faulty input and write $\mathsf{FLVrfy}_k(m, \tau, z)$ where z represents the fault injected into y_1, namely the hash value h. On the other hand, a leaky tag generation query is in the form of $\mathsf{LMac}_k(m)$ since we only consider leakage here. We stress that for LR-MAC1, fault-and-leak attacks and leak-and-fault attacks are equivalent. This is because the only possible faulty value is the hash value h which can be obtained locally by the adversary before the computation of LR-MAC1.

Discussion and overview of the proof. The proof is based on the observation that in order to find a fresh and valid pair (m, τ) against LR-MAC1, the adversary needs to either find a collision against the hash function H, or to find a fresh and valid tuple (tw, x, y) against the SUP−L2 security of TBC F even with the power of injecting faults in tag verification. In the proof, the adversary is deemed to win the game if any of her $q_V + 1$ verification queries can be associated to a valid predication against the TBC F. This is to capture the power of the adversary on tag verification since now the adversary can inject any fault on the hash value h and thus has the full control of the input to the TBC F, which is essentially different from Berti et al.'s [BGPS21] model where the adversary can only see what h is but cannot modify it. Our analysis implies that the inversion of TBC in tag verification not only helps to improve the security against side-channel attack, but also significantly improves the security against fault attacks.

Proof. We use a sequence of games to proceed the proof. Denote by E_i the event that the adversary wins the ith Game. Game 0 is exactly the SUF−FL2 game where the adversary (q_{FL}, q_M, q_V, t)-adversary $\mathsf{A}^{\mathsf{L},\mathcal{F}}$ aims at producing a forgery against LR-MAC1.

Game 1 is the same as Game 0 except that we abort if there is a collision in the hash function. Clearly Game 0 and Game 1 are identical if there is no collision in the hash function. We construct an adversary B to bound the difference between Game 0 and Game 1. Adversary B plays the collision resistant game against the hash function H (see Definition 2), and simulates adversary A's oracles by using its own oracle. At the start of game, adversary B picks up a key k uniformly at random from $\mathcal{K}$ for the TBC F. By using s the key of hash function and k the key of the TBC, adversary B can correctly simulate A's oracles that are defined in experiment in Table 4. During the simulation, adversary B holds a list $\mathcal{H}$ to record the input-output pairs of hash function H. That is, every time when H is invoked $y = \mathsf{H}_s(x)$, she will put the pair (x, y) into the list $\mathcal{H}$. At the end of game, adversary A outputs a pair (m, τ). Adversary B then computes $h = \mathsf{H}(m)$ and put the pair (m, h) into the list $\mathcal{H}$. She checks whether there is a collision in the list $\mathcal{H}$, namely $x \neq x'$ but $\mathsf{H}_s(x) = \mathsf{H}_s(x')$. If so, she outputs this collision and wins the game. The time complexity of adversary B is $t_1 = t + (q_M + q_V + q_{FL})(t_{\mathsf{F}} + t_{\mathsf{L}} + t_{\mathsf{H}}) + t_{\mathsf{H}}$. Hence, we have

$$|\Pr[E_0] - \Pr[E_1]| \leq \epsilon_{\mathsf{CR}}.$$

Game 2 is the same as Game 1 except that we abort if in some faulty verification query (m_i, τ_i, z_i) made by A, it can be transformed into a valid prediction against the

TBC F. That is, $(z_i, 0^n, \tau_i)$ a valid prediction against F. To analyze the difference between Game 1 and Game 2, we build a sequence of $q_V + 2$ games Game 1^0, ..., Game 1^{q_V+1} as follows. Game 1^j is the same as Game 1 except that we abort if one of the first j faulty verification queries can be associated to a valid prediction against the TBC F. Thus, Game 1^0 is exactly Game 1 while Game 1^{q_V+1} is exactly Game 2. Let E_1^j be the event that the adversary A wins Game 1^j. Clearly, Game 1^j and Game 1^{j+1} are identical if the $(j+1)$th faulty verification query cannot be associated to a valid prediction against the TBC F. We regard adversary A's final output (m, τ) as the $(q_V + 1)$th verification query, where there are no faults otherwise trivial forgery exists.

We then build an adversary C^j to bound the difference between any two sub-games 1^j and 1^{j+1}. Adversary C^j plays the $\mathsf{SUP\text{–}L2}$ game (illustrated in Table 3) against the TBC F, and simulates adversary A's oracles by using its own oracles. At the start of game, adversary C^j picks up a key s uniformly at random from $\mathcal{HK}$ for the hash function H. With the help of s and its own oracles, she can simulate correctly Game 1^j for adversary A. Then when A asks her $(j+1)$th faulty verification query $(m_{j+1}, \tau_{j+1}, z_{j+1})$, adversary C^j computes $h_{j+1} = \mathsf{H}_s(m_{j+1})$ and outputs $(z_{j+1}, 0^n, \tau_{j+1})$ as her prediction against F. Here the value of z_{j+1} depends on different types of fault attacks: (1) in the stuck-at model, z_1 is the value controlled by the adversary A; (2) in the differential fault model, $z_{j+1} = z_{j+1} \oplus h_{j+1}$ where z_{j+1} is the differential value chosen by the adversary A. In any of these faults, adversary C^j can simulate it correctly for adversary A since she has the key s of the hash function and she has the full control of queries to her oracles. Adversary C^j makes at most q_{FL} queries to L, q_M queries to LEval and $j \leq q_V$ queries to LInv. She runs in time at most $t + (q_M + q_{FL} + q_V + 1)t_{\mathsf{H}}$. Thus,

$$|\Pr[E_1^j] - \Pr[E_1^{j+1}]| \leq \epsilon_{\mathsf{SUP\text{-}L2}}.$$

From the hybrid argument,

$$|\Pr[E_1] - \Pr[E_2]| \leq \sum_{j=0}^{q_V} |\Pr[E_1^j] - \Pr[E_1^{j+1}]| \leq (q_V + 1)\epsilon_{\mathsf{SUP\text{-}L2}}.$$

For Game 2, since $(h_{q_V+1}, 0^n, \tau_{q_V+1})$ cannot be a valid predication against the TBC F, we have $\Pr[E_2] = 0$. Finally, wrapping up,

$$\Pr[E_0] \leq \epsilon_{\mathsf{CR}} + (q_V + 1)\epsilon_{\mathsf{SUP\text{-}L2}}$$

and conclude the proof of Theorem 1. □

5.2 Attacks against other MACs

The previous positive result heavily relies on the inverse-based verification of $\mathsf{LR\text{-}MAC1}$. In this subsection, we show that such a positive result is not always obtained by exhibiting attacks against other verification algorithms that recompute the correct tag like analyzed in [DM21]. Those are typically encountered in permutation-based designs like Ascon [DEMS21] or ISAP [DEM+20]. The first attack is a generic bit-level fault one while the second attack is combining faults and leakage.

Bit-level fault attack. Let Vrfy_k be a verification algorithm that recomputes the correct tag τ in verification. The goal of this attack is to recover one by one the bits of the correct tag τ of the message m. For this purpose, the adversary can simply use stuck-at faults where all the bits of the re-computed tag are set to zero but one, and use an all-zero tag candidate in the comparison. If the verification algorithm accepts the re-computed tag, it means the corresponding bit of the tag is zero, otherwise it is one. After performing $|\tau|$ faulted queries on different bits, the adversary has recovered the tag in full.

Attacking a SPA-secure Design with DPA. Consider Figure 1 in [DM21]. The high-level idea of this leakage-resilient tag verification algorithm is that it maintains the message integrity if the inputs S and T (corresponding to the tag) of a permutation are secure against Simple Power Analysis (i.e., single-input attacks, roughly). Yet, in the context of a combined attack, an adversary can easily use a fault in order to modify the value of S while keeping T constant, in such a way that a DPA (i.e., a multi-input attack) against T becomes possible. Applied to ISAP or Ascon, it means that their leveled implementation should additionally protect this permutation with strong side-channel or fault countermeasures or it becomes possible to forge tags without knowledge of the long-term key.

5.3 Insecure Tag Generation

We finally observe that the good properties of LR-MAC1's tag verification do not extend to its tag generation by exhibiting an attack in this context. If an adversary can inject stuck-at faults, she first computes locally $h' = \mathsf{H}_s(m')$. She next queries m and injects a fault to replace the hash value $h = \mathsf{H}_s(m)$ with h', and obtain the tag τ. Then, (m', τ) is a valid forgery for LR-MAC1 since (m', τ) is fresh and it can pass the verification oracle. If an adversary can only inject differential faults, she first computes locally two hash values $h' = \mathsf{H}_s(m')$ and $h = \mathsf{H}_s(m)$. She next obtains the differential value $\Delta = h' \oplus h$. Then, she queries m and injects the differential fault Δ into the hash value h in order to obtain the tag τ so that (m', τ) is again a valid forgery for LR-MAC1. While this attack is in a relatively strong model, it naturally raises the question whether improved security can be reached at the mode level. The next two sections answer this question positively by exhibiting two approaches allowing to improve physical security against side-channel and fault attacks in contexts where also the tag generation can be targeted.

6 LR-MACd: improved security by iteration

In this section, we propose an new MAC algorithm called LR-MACd that has better security against fault attacks in tag generation, under the plausible assumption that inserting faults on multiple and large intermediate computations in one execution is difficult for the adversary. LR-MACd requires one more TBC call and one more hash function call than LR-MAC1, but it is SUF−FL2 secure, while LR-MAC1 is SUF−FL1 secure.

Scheme description. Let $\mathsf{F} : \mathcal{K} \times \mathcal{TW} \times \{0,1\}^n \to \{0,1\}^n$ be a tweakable block cipher (TBC) and $\mathsf{H} : \mathcal{HK} \times \{0,1\}^* \to \mathcal{TW}$ be a hash function. The algorithm LR-MACd is built by invoking two calls of hash function H and two calls of TBC F. See Algorithm 2 and Figure 2 for code description and figure of LR-MACd respectively.

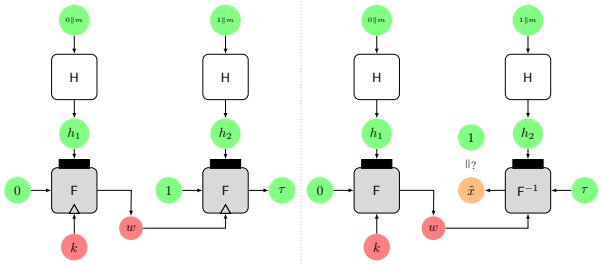

Figure 2: LR-MACd.

Note that the scheme uses two protected TBCs sequentially. So in practice, the same countermeasures should be implemented for both so that even the intermediate value w is protected (which is at the same time natural and required).

Algorithm 2 The LR-MACd algorithm.

It uses a strongly protected TBC $\mathsf{F} : \mathcal{K} \times \mathcal{TW} \times \{0,1\}^n \to \{0,1\}^n$ and a hash function $\mathsf{H} : \mathcal{HK} \times \{0,1\}^* \to \mathcal{TW}$.

Gen:

- $k \xleftarrow{\$} \mathcal{K}$
- $s \xleftarrow{\$} \mathcal{HK}$

$\mathsf{Mac}_k(m)$:

- $h_1 = \mathsf{H}_s(0 \,\|\, m)$
- $w = \mathsf{F}_k^{h_1}(0^n)$
- $h_2 = \mathsf{H}_s(1 \,\|\, m)$
- $\tau = \mathsf{F}_w^{h_2}(0^{n-1}1)$
- Return τ

$\mathsf{Vrfy}_k(m, \tau)$:

- $h_1 = \mathsf{H}_s(0 \,\|\, m)$
- $w = \mathsf{F}_k^{h_1}(0^n)$
- $h_2 = \mathsf{H}_s(1 \,\|\, m)$
- $\tilde{x} = \mathsf{F}_w^{h_2,-1}(\tau)$
- If $\tilde{x} == 0^{n-1}1$ Return 1
- Else Return 0

6.1 Secure Tag Generation and Verification

Self-preserving unpredictability of TBC. Since LR-MACd uses the first TBC to derive the key for the second TBC which is invisible from the adversary, the previous SUP–L2 security definition is not suitable here. Instead, we present an new security definition called self-preserving unpredictability (SPU–L2) to capture the property needed in LR-MACd. Intuitively, it says that the adversary has the oracle access to the TBC F_k under the long-term secret key k. The output y of TBC F_k can be used as the key for another TBC. In this case, the adversary can only receive the leakage and not the output y. The adversary can also query the TBC under the derived key, whose output can also be used as the key for other TBCs. Finally, it should be hard for the adversary to output a valid predication (tw, x, y) against any of these TBCs. Formally, it leads to the following definition:

Definition 7 (SPU–L2)**.** A tweakable block cipher $\mathsf{F} : \mathcal{K} \times \mathcal{TW} \times \{0,1\}^n \to \{0,1\}^n$ with leakage function pair $\mathsf{L} = (\mathsf{L}_{\mathsf{Eval}}, \mathsf{L}_{\mathsf{Inv}})$ is $(q_{\mathsf{L}}, q_E, q_I, q_{Ek}, q_{Ik}, t, \epsilon)$-*self-preserving unpredictable with leakage in evaluation and inversion* (SPU–L2), or $(q_{\mathsf{L}}, q_E, q_I, q_{Ek}, q_{Ik}, t, \epsilon)$-SPU–L2, if for any $(q_{\mathsf{L}}, q_E, q_I, q_{Ek}, q_{Ik}, t)$-adversary A, we have:

$$\Pr[\mathsf{SPU\text{–}L2}_{\mathsf{A},\mathsf{F},\mathsf{L}} \Rightarrow 1] \leq \epsilon,$$

where the SPU–L2 experiment is defined in Table 5.

Remark. In the game of self-preserving unpredictability of F (Table 5), $\mathcal{Q}_1$ is the set of forbidden evaluation queries that are used to derive subkeys. Similarly, $\mathcal{Q}_2$ is the set of forbidden key queries that are used in the evaluation of the TBC.

Security of LR-MACd. The security of LR-MACd against leakage and faults is captured by the following theorem. We consider faults in both tag generation and verification. Recall that the security experiment is illustrated in Table 4.

Theorem 2. *Let* H *be a* $(t_1, \epsilon_{\mathsf{CR}})$*-collision resistant hash function. Let* F *be a* $(q_{FL}, q_M, q_V, q_M + q_V, 0, t_2, \epsilon_{\mathsf{SPU\text{–}L2}})$-SPU–L2 *tweakable block cipher with fault-immune long-term key. Then we show that for any* (q_{FL}, q_M, q_V, t)*-adversary* $\mathsf{A}^{\mathsf{L},\mathcal{F}}$ *with leaking function pair*

Table 5: Self-preserving unpredictability of F with leakage.

The $\mathsf{SPU\text{-}L2}_{\mathsf{A},\mathsf{F},\mathsf{L}}$ experiment.	
Initialization: $k_0 \xleftarrow{\$} \mathcal{K}$ $\mathcal{L}, \mathcal{Q}_1, \mathcal{Q}_2 \leftarrow \emptyset$ $c \leftarrow 0$ Finalization: $(i,t,x,z) \leftarrow \mathsf{A}^{\mathsf{L},\mathsf{LKey},\mathsf{LInk},\mathsf{LEval},\mathsf{LInv}}$ If $(i,t,x,z) \in \mathcal{L}$ or $i \notin [c]$ Return 0 If $z = \mathsf{F}_{k_i}(t,x)$ Return 1 Return 0 Oracle $\mathsf{LKey}(i,t,x)$ If $i \notin [c]$ or $(+,i,t,x) \in \mathcal{Q}_1 \cup \mathcal{Q}_2$ Return $\perp$ $c = c+1$ $k_c = \mathsf{F}_{k_i}(t,x)$ $\mathsf{l_e} = \mathsf{L}_{\mathsf{Eval}}(t,x;k_i)$ $\mathcal{Q}_1 \leftarrow \mathcal{Q}_1 \cup \{(+,i,t,x)\}$ Return $\mathsf{l_e}$	Oracle $\mathsf{LEval}(i,t,x)$: If $i \notin [c]$ or $(+,i,t,x) \in \mathcal{Q}_1$ Return $\perp$ $z = \mathsf{F}_{k_i}(t,x)$ $\mathsf{l_e} = \mathsf{L}_{\mathsf{Eval}}(t,x;k_i)$ $\mathcal{Q}_2 \leftarrow \mathcal{Q}_2 \cup \{(+,i,t,x),(-,i,t,z)\}$ $\mathcal{L} \leftarrow \mathcal{L} \cup \{(i,t,x,z)\}$ Return $(z, \mathsf{l_e})$ Oracle $\mathsf{LInv}(i,t,z)$: If $i \notin [c]$ or $(-,i,t,x) \in \mathcal{Q}_1$ Return $\perp$ $x = \mathsf{F}^{-1}_{k_i}(t,z)$ $\mathsf{l_i} = \mathsf{L}_{\mathsf{Eval}}(t,z;k_i)$ $\mathcal{L} \leftarrow \mathcal{L} \cup \{(i,t,x,z)\}$ $\mathcal{Q}_2 \leftarrow \mathcal{Q}_2 \cup \{(-,i,t,z),(+,i,t,x)\}$ Return $(y, \mathsf{l_i})$ Oracle $\mathsf{LInk}(i,t,z)$ If $i \notin [c]$ or $(-,i,t,x) \in \mathcal{Q}_1 \cup \mathcal{Q}_2$ Return $\perp$ $c = c+1$ $k_c = \mathsf{F}^{-1}_{k_i}(t,z)$ $\mathsf{l_i} = \mathsf{L}_{\mathsf{Inv}}(t,z;k_i)$ $\mathcal{Q}_1 \leftarrow \mathcal{Q}_1 \cup \{(-,i,t,z)\}$ Return $\mathsf{l_i}$

$\mathsf{L} = (\mathsf{L}_{\mathsf{Mac}}, \mathsf{L}_{\mathsf{Vrfy}})$ *and faulty injection pair* $\mathcal{F} = (\mathcal{F}_{\mathsf{Mac}}, \mathcal{F}_{\mathsf{Vrfy}})$, ***LR-MACd*** *is* $(q_{FL}, q_M, q_V, t, \epsilon)$*-strongly existentially unforgeable against both stuck-at and differential 1-bounded fault-then-attacks in tag verification and verification, with*

$$\epsilon \leq \epsilon_{\mathsf{CR}} + (q_V + 1)\epsilon_{\mathsf{SPU\text{-}L2}},$$

where $t_1 = t + 2(q_M + q_V + q_{\mathsf{L}})(t_{\mathsf{H}} + t_{\mathsf{F}} + t_{\mathsf{L}}) + 2t_{\mathsf{H}}$ *and* $t_2 = t + 2(q_M + q_V + q_{\mathsf{L}})(t_{\mathsf{H}} + t_{\mathsf{F}} + t_{\mathsf{L}}) + 2t_{\mathsf{H}}$.

Faulty matrix and leakage function. As before, we first specify the faulty injection pair $\mathcal{F} = (\mathcal{F}_{\mathsf{Mac}}, \mathcal{F}_{\mathsf{Vrfy}})$ and leaking function pair $\mathsf{L} = (\mathsf{L}_{\mathsf{Mac}}, \mathsf{L}_{\mathsf{Vrfy}})$. We begin with the faulty injection function $\mathcal{F}_{\mathsf{Mac}}$ in tag generation. The mode-level atomic implementation for $\mathcal{F}_{\mathsf{Mac}}$ is $f_1 = \mathsf{H}_s(0 \,\|\, \cdot)$, $f_2 = \mathsf{F}_k(\cdot, 0^n)$, $f_3 = \mathsf{H}_s(1 \,\|\, \cdot)$, and $f_4 = \mathsf{F}.(\cdot, 0^{n-1}1)$. This means that the adversary is unable to modify the parameter s, the two one-bit prefixes 0 and 1, the two constant inputs 0^n and $0^{n-1}1$. For input m, we have $x_1 = m$, $y_1 = \mathsf{H}_s(0 \,\|\, x_1)$, $y_2 = \mathsf{F}_k(y_1, 0^n)$, $y_3 = \mathsf{H}_s(1 \,\|\, x_1)$, and $y_4 = \mathsf{F}_{y_2}(y_3, 0^{n-1}1)$. The dependency matrix and the empty faulty matrix are then given by

$$\begin{pmatrix} x_1 & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & y_1 & \varepsilon & \varepsilon \\ x_1 & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & y_2 & y_3 \end{pmatrix}, \qquad \mathcal{F}_{\mathsf{Mac}} = \begin{pmatrix} \cdot & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \cdot & \varepsilon & \varepsilon \\ \cdot & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \cdot & \cdot \end{pmatrix}.$$

We require an additional protection on y_2 which is the key of the second TBC in Mac_k. Hence the protected dependency matrix and empty faulty matrix become

$$\begin{pmatrix} x_1 & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & y_1 & \varepsilon & \varepsilon \\ x_1 & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \perp & y_3 \end{pmatrix}, \qquad \mathcal{F}_{\mathsf{Mac}} = \begin{pmatrix} \cdot & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \cdot & \varepsilon & \varepsilon \\ \cdot & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \perp & \cdot \end{pmatrix}.$$

A faulty leaky tag generation query has the form $\mathsf{FLMac}_k(m, (z_1, z_2, z_3, z_4))$, where the function corresponding to faulty matrix is

$$\mathcal{F}_{\mathsf{Mac}} = \begin{pmatrix} z_1 & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & z_2 & \varepsilon & \varepsilon \\ z_3 & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \perp & z_4 \end{pmatrix}.$$

Note that $(m, (z_1, z_2, z_3, z_4))$ is a valid tag generation query if and only if $z_1 \neq z_3$ as otherwise it is the same as the query $(z_1, (\cdot, z_2, \cdot, z_4))$. That is, $\mathcal{F}_{\mathsf{Mac}}(m, (z_1, z_2, z_3, z_4)) = 1$ if and only if $z_1 \neq z_3$. For simplicity, we write $\mathsf{FLMac}_k(m, z)$ where $z = (z_1, z_2, z_3, z_4)$.

On the other hand, the mode-level atomic implementation for $\mathcal{F}_{\mathsf{Vrfy}}$ in the tag verification is $f_1 = \mathsf{H}_s(0 \parallel \cdot)$, $f_2 = \mathsf{F}_k(0^n, \cdot)$, $f_3 = \mathsf{H}_s(1 \parallel \cdot)$, and $f_4 = \mathsf{F}_{\cdot}^{-1}(\cdot, \cdot)$. For input (m, τ), we thus have $x_1 = m$, $x_2 = \tau$, $y_1 = \mathsf{H}_s(0 \parallel x_1)$, $y_2 = \mathsf{F}_k(0^n, y_1)$, $y_3 = \mathsf{H}_s(1 \parallel x_1)$, and $y_4 = \mathsf{F}_{y_2}^{-1}(y_3, x_2)$. Similarly to the case of LR-MAC1, we reveal the check value y_4 in the leakage trace of $\mathsf{L}_{\mathsf{Vrfy}}$ and thus ignore the function regarding the check operation. The protected dependency matrix and the empty fault matrix are then given by

$$\begin{pmatrix} x_1 & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & y_1 & \varepsilon & \varepsilon \\ x_1 & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & x_2 & \varepsilon & \perp & y_3 \end{pmatrix}, \qquad \mathcal{F}_{\mathsf{Vrfy}} = \begin{pmatrix} \cdot & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \cdot & \varepsilon & \varepsilon \\ \cdot & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \cdot & \varepsilon & \perp & \cdot \end{pmatrix},$$

where the protected set $\mathcal{I}_{\mathsf{Vrfy}} = \{(4, 4)\}$. Hence a faulty leaky tag verification query has the form $\mathsf{FLVrfy}_k(m, \tau, (z_1, z_2, z_3, z_4, z_5))$, where the faulty matrix function is

$$\mathcal{F}_{\mathsf{Vrfy}} = \begin{pmatrix} z_1 & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & z_2 & \varepsilon & \varepsilon \\ z_3 & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & z_4 & \varepsilon & \perp & z_5 \end{pmatrix}.$$

Note that $(m, \tau, (z_1, z_2, z_3, z_4, z_5))$ is a valid tag generation query if and only if $z_1 \neq z_3$ and $z_5 = \cdot$ as otherwise it is trivially the same as the query $(z_1, z_5, (\cdot, z_2, \cdot, z_4, \cdot))$. Hence without loss of generality, we write $\mathsf{FLVrfy}_k(m, \tau, z)$ where $z = (z_1, z_2, z_3, z_4)$. Since we are working on the 1-bounded model, the adversary can select any one of these faulty inputs to inject in each query, either in tag generation or verification.

Finally, for leaking function pair $\mathsf{L} = (\mathsf{L}_{\mathsf{Mac}}, \mathsf{L}_{\mathsf{Vrfy}})$, we assume $\mathsf{L}_{\mathsf{F}} = (\mathsf{L}_{\mathsf{Eval}}, \mathsf{L}_{\mathsf{Inv}})$ is the leakage function pair of the TBC F. Then we have $\mathsf{L}_{\mathsf{Mac}} = (\mathsf{L}_{\mathsf{Eval}}, \mathsf{L}_{\mathsf{Eval}})$ since there are two TBCs involved. Similarly, $\mathsf{L}_{\mathsf{Vrfy}} = (\mathsf{L}_{\mathsf{Eval}}, \mathsf{L}_{\mathsf{Inv}}, y_4)$ since the second TBC is in the backward direction and the check value y_4 is leaked to the adversary.

Discussion and overview of the proof. Theorem 2 can be interpreted as a claim that LR-MACd provides SUF−FL2 security as long as the underlying hash function is collision resistant and the TBC F is self-preserving unpredictable (SPU−L2). In the proof, the adversary is deemed to win the game if any of her $q_V + 1$ verification queries can be associated to a valid predication against the SPU−L2 security of the TBC F.

Proof. Similarly to the proof of LR-MAC1, we use a sequence of games to facilitate the proof. We next denote as E_i the event that the adversary A wins the ith Game.

Game 0 is exactly the SUF−FL2 game where the (q_{FL}, q_M, q_V, t)-adversary $\mathsf{A}^{\mathsf{L},\mathcal{F}}$ aims at producing a forgery against LR-MACd.

Game 1 is the same as Game 0 except that we abort if there is a collision in the hash function H. Obviously Game 0 and Game 1 are identical if there is no collision in the hash function. We construct an adversary B to bound the difference between these two games. Adversary B plays the game against the collision-resistance property of H, and simulates A's oracles by using its own oracle. The simulation strategy is similar to Theorem 1, since by using s the key of hash function H and k the selected key of the TBC F, adversary B can always simulate correctly A's oracles. The time complexity of adversary B is $t_1 = t + 2(q_M + q_V + q_{FL})(t_{\mathsf{F}} + t_{\mathsf{L}} + t_{\mathsf{H}}) + 2t_{\mathsf{H}}$. Hence, we have

$$|\Pr[E_0] - \Pr[E_1]| \leq \epsilon_{\mathsf{CR}}.$$

Game 2 is the same as Game 1 except that we abort if in some faulty verification query (m_i, τ_i, z_i) where $z_i = (z_{i,1}, z_{i,2}, z_{i,3}, z_{i,4})$ made by A, it can be transformed into a valid prediction against the SPU−L2 security of the TBC F. To analyze the difference between Game 1 and Game 2, we construct a sequence of $q_V + 2$ games Game 1^0,..., Game 1^{q_V+1} as follows. Game 1^j is the same as Game 1 except that we abort if one of the first j faulty verification queries can be associated to a valid prediction against the SPU−L2 security of TBC F. Thus, Game 1^0 is exactly Game 1 while Game 1^{q_V+1} is exactly Game 2. Let E_1^j be the event that the adversary A wins Game 1^j. Clearly, Game 1^j and Game 1^{j+1} are identical if the $(j+1)$th faulty verification query cannot be associated to a valid prediction against the SPU−L2 security of TBC F. We regard adversary A's final output as the $(q_V + 1)$th verification query, where there are no faults otherwise trivial forgery exists.

We then build an adversary C^j to bound the difference between any two sub-games 1^j and 1^{j+1}. Adversary C^j plays the game against the SPU−L2 security of TBC F (illustrated in Table 5), and simulates adversary A's oracles by using its own oracles. At the start of game, adversary C^j picks up a key s uniformly at random from $\mathcal{HK}$ for the hash function H. With the help of s and its own oracles, she can simulate correctly Game 1^j for adversary A. For example, for each tag verification query (m_i, τ_i, z_i) with $i \leq j$ from adversary A where $z_i = (z_{i,1}, z_{i,2}, z_{i,3}, z_{i,4})$, adversary C^j first computes $h_{i,1} = \mathsf{H}_s(0 \parallel z_{i,1})$, and queries her oracle LKey with input $(i, z_{i,2}, 0^n)$ to obtain leakage $\mathsf{l_e}$. She then queries her oracle LInv with input $(i, z_{i,3}, \tau_i)$ to obtain $\tilde{x}$ and leakage $\mathsf{l_i}$. She replies $(\tilde{x} == 1, (\mathsf{l_e}, \mathsf{l_i}))$ to the adversary A. The simulation for tag generation query is similar. Then when A asks her $(j+1)$th faulty verification query $(m_{j+1}, \tau_{j+1}, z_{j+1})$, adversary C^j computes $h_{1,j+1} = \mathsf{H}_s(0 \parallel z_{j+1,1})$, and queries her oracle LKey with input $(j+1, z_{j+1,2}, 0^n)$. She computes $h_{2,j+1} = \mathsf{H}_s(1 \parallel z_{j+1,3})$, and outputs $(j+1, z_{j+1,4}, 0^{n-1}1, \tau_{j+1})$ as her prediction against the SPU−L2 security of the TBC F. Here the adversary can only select one of $z_{j+1,1}$, $z_{j+1,2}$, $z_{j+1,3}$ and $z_{j+1,4}$ to be faulty input since we work on the 1-bounded model. However, the selected one can be either stuck-at fault or differential fault as the adversary wants. In any of these faults, adversary C^j can simulate it correctly for adversary A since she has the key s of the hash function and she has the full control of queries to her oracles. Adversary C^j makes at most q_{L} queries to L, q_M queries to LEval, $j \leq q_V$ queries to LInv, and $q_M + j \leq q_M + q_V$ queries to LKey. She runs in time at most $t_2 = t + 2(q_M + q_V + q_{\mathsf{L}})(t_{\mathsf{H}} + t_{\mathsf{F}} + t_{\mathsf{L}}) + 2t_{\mathsf{H}}$. Thus,

$$|\Pr[E_1^j] - \Pr[E_1^{j+1}]| \leq \epsilon_{\mathsf{SPU\text{-}L2}}.$$

From the hybrid argument,

$$|\Pr[E_1] - \Pr[E_2]| \leq \sum_{j=0}^{q_V} |\Pr[E_1^j] - \Pr[E_1^{j+1}]| \leq (q_V + 1)\epsilon_{\mathsf{SPU\text{-}L2}}.$$

For Game 2, since $(h_{q_V+1}, 0^n, \tau_{q_V+1})$ cannot be a valid prediction against the TBC F, we have $\Pr[E_2] = 0$. Finally, wrapping up,

$$\Pr[E_0] \leq \epsilon_{\mathsf{CR}} + (q_V + 1)\epsilon_{\mathsf{SPU\text{-}L2}}$$

and conclude the proof of Theorem 2. □

6.2 Grating Attack on Iterative Constructions

We finally put forward that the additional (1-bounded fault) assumption used in this section is needed by showing a generic stuck-at attack (coined grating attack) on iterative construction without additional countermeasure. The attack idea is contained in the name "grating attack": it works by placing a portion of one query into a branch of another query in such a way that a union will form a valid forgery. For simplicity, we assume that a scheme S is build from cascading two components H and F, namely $S(m) = F \circ H(m)$ for a message m where $h = H(m)$ is the internal value. Whether the adversary has the oracle access to H or F (namely whether H and F have a secret key or not) is irrelevant to this attack. First, the adversary queries message m_1 to the scheme S. After the computation of component H, she injects an arbitrary faulted value h^* ($h^* \neq h_1$) as the input to F. She then obtains $h_1 = H(m_1)$ from the leakage. Secondly, she queries message m_2 to the scheme S. After the computation of component H, she injects the faulted value h_1 as the input to F. She obtains the tag τ_2 which is the output of the scheme S. Then the pair (m_1, τ_2) is a valid forgery since it is fresh and can pass the verification oracle.

We note that one option to get rid of this attack is to generalize Figure 2 with more protected TBCs, which could lead to security in the ℓ-bounded fault model with larger ℓ values and is left as an open problem. But this naturally also increases the cost of the construction. We next study another option that works with different requirements.

7 LR-MACr: improved security with randomness

In this section, we propose a MAC algorithm called LR-MACr that is secure against leakage and faults in both tag generation and verification with the addition of auxiliary randomness. This randomness helps improving security against differential faults in the fault-then-leak model because it prevents the hash function to be publicly computable during tag generation. As a result, the adversary can only XOR a chosen string to an unknown randomness and the model equivalence of Section 4.4 does not hold. In verification, the hash function remains publicly computable and essentially follows LR-MAC1.

Scheme description. Let $\mathsf{F} : \mathcal{K} \times \mathcal{TW} \times \{0,1\}^n \to \{0,1\}^n$ be a tweakable block cipher (TBC) and $\mathsf{H} : \mathcal{HK} \times \{0,1\}^* \to \mathcal{TW}$ be a hash function. The algorithm LR-MACr is built from a hash function H and a TBC F with an auxiliary randomness $r \in \{0,1\}^n$. The value r is always freshly chosen uniformly at random from the set $\{0,1\}^n$ for each tag generation, while for verification, the adversary can arbitrarily choose r as she wants. The code description and figure of LR-MACr are in Algorithm 3 and Figure 3.

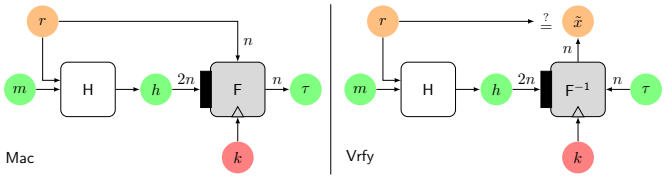

Figure 3: LR-MACr.

Algorithm 3 The LR-MACr algorithm.

It uses a strongly protected TBC $\mathsf{F} : \mathcal{K} \times \mathcal{TW} \times \{0,1\}^* \to \{0,1\}^n$ and a hash function $\mathsf{H} : \mathcal{HK} \times \{0,1\}^* \to \mathcal{TW}$.

Gen:

- $k \xleftarrow{\$} \mathcal{K}$, $s \xleftarrow{\$} \mathcal{HK}$

$\mathsf{Mac}_k(m)$:

- $r \xleftarrow{\$} \{0,1\}^n$
- $h = \mathsf{H}_s(r\|m)$
- $\tau = \mathsf{F}_k^h(r)$
- Return (r, τ)

$\mathsf{Vrfy}_k(m, r, \tau)$:

- $h = \mathsf{H}_s(r\|m)$
- $\tilde{x} = \mathsf{F}_k^{h,-1}(\tau)$
- If $\tilde{x} == r$ Return 1
- Else Return 0

Table 6: The preimage resistance of H in the hash oracle.

The $\mathsf{PRC}_{\mathsf{H},\mathsf{A}}$ experiment	
Initialization: $s \xleftarrow{\$} \mathcal{HK}$ $\mathcal{H} \leftarrow \emptyset$ Oracle $\mathsf{H}(m)$: $h = \mathsf{H}_s(m)$ $\mathcal{H} \leftarrow \mathcal{H} \cup \{(m,h)\}$ Return h	Finalization: $(\tilde{h}, \tilde{m}) \leftarrow \mathsf{A}^{\mathsf{H}}(s)$ If $(*, \tilde{h}) \in \mathcal{H}$ Return 0 If $\tilde{h} = \mathsf{H}_s(\tilde{m})$ Return 1 Return 0

Hash oracle model. In the security analysis we would like to learn the off-line hash evaluations made by the adversary. That is, we model H as a *hash oracle* so that whenever the adversary wants to locally compute $\mathsf{H}_s(x)$ on chosen input x, the environment gets x and stores (x, y) in a hash list $\mathcal{H}$, where $y = \mathsf{H}_s(x)$. The hash oracle allows *knowing* each pair (x, y) at the time of the $\mathsf{H}_s(x)$ computation made by the adversary but it does *not* control the distribution of the outputs y which, given the adversary's input x, remains deterministic. The hash oracle model is thus a *non-idealized* computational model which remains compatible with the fact the the adversary knows the implementation of H.

Preimage resistance in the hash oracle model. In the hash oracle model, we capture the preimage resistance of H by allowing the adversary, on input s, to choose her target y whenever she wants as long as she never got y as an output from a previous evaluation of H_s. This definition captures the essence of preimage resistance in the sense that it is hard to find any preimage of a value that is not already known as an output. The hash oracle allows knowing which targets are already an output or not, thanks to the hash list $\mathcal{H}$.

Definition 8. A hash function $\mathsf{H} : \mathcal{HK} \times \{0,1\}^* \to \{0,1\}^n$ is a $(t, \epsilon_{\mathsf{PRC}})$-*collision resistant after computation* hash function, if for every t-adversary A, the probability

$$\Pr[\mathsf{PRC}_{\mathsf{H},\mathsf{A}} \Rightarrow 1] \leq \epsilon_{\mathsf{PRC}},$$

where the experiment $\mathsf{PRC}_{\mathsf{H},\mathsf{A}}$ is illustrated in Table 6.

Security of LR-MACr. The security of LR-MACr against leakage and faults is formalized by the following theorem (see the experiment in Table 4).

Theorem 3. *Let hash function* H *be* $(t_1, \epsilon_{\mathsf{CR}})$*-collision resistance and* $(t_2, \epsilon_{\mathsf{PRC}})$*-preimage resistance in the hash oracle model. Let* F *be a* $(q_{FL}, q_M, q_V, t_3, \epsilon_{\mathsf{SUP\text{-}L2}})$-$\mathsf{SUP}$–$\mathsf{L2}$ *tweakable block cipher with fault-immune long-term key. Then we show that for any* (q_{FL}, q_M, q_V, t)*-adversary* $\mathsf{A}^{\mathsf{L},\mathcal{F}}$ *with leaking function pair* $\mathsf{L} = (\mathsf{L}_{\mathsf{Mac}}, \mathsf{L}_{\mathsf{Vrfy}})$ *and faulty injection pair* $\mathcal{F} = (\mathcal{F}_{\mathsf{Mac}}, \mathcal{F}_{\mathsf{Vrfy}})$*, LR-MACr is* $(q_{FL}, q_M, q_V, t, \epsilon)$*-strongly existentially unforgeable against unbounded differential fault-then-attacks in tag verification and verification, with*

$$\epsilon \leq \epsilon_{\mathsf{CR}} + (q_V + 1)\epsilon_{\mathsf{SUP\text{-}L2}} + \epsilon_{\mathsf{PRC}} + \frac{q_M^2}{2^{n+1}} + \frac{q_M}{2^n},$$

where $t_1 = t + (q_M + q_V + q_{\mathsf{L}})(t_{\mathsf{F}} + t_{\mathsf{L}} + 2t_{\mathsf{H}}) + t_{\mathsf{H}}$*,* $t_2 = t + (q_M + q_V + q_{\mathsf{L}})(2t_{\mathsf{H}} + t_{\mathsf{F}} + t_{\mathsf{L}})$*, and* $t_3 = t + (q_M + q_V + q_{\mathsf{L}})(t_{\mathsf{H}} + t_{\mathsf{F}} + t_{\mathsf{L}}) + t_{\mathsf{H}}$*.*

Faulty matrix and leakage function. We begin by specifying the faulty function pair $(\mathcal{F}_{\mathsf{Mac}}, \mathcal{F}_{\mathsf{Vrfy}})$ and leaking function pair $\mathsf{L} = (\mathsf{L}_{\mathsf{Mac}}, \mathsf{L}_{\mathsf{Vrfy}})$. Differently from the two previous constructions, we assume that the adversary can *only inject differential faults*. To simplify the notation, we omit all the Δ subscript in the fault descriptions. For the tag generation function of LR-MACr, we consider the mode-level atomic implementation with $f_0 = r \xleftarrow{\$} \{0,1\}^n$, that is, the random sampling of r, $f_1 = \mathsf{H}_s(\cdot)$ and $f_2(\cdot) = \mathsf{F}_k(\cdot)$. For input $(x_1) = m$ we thus have $x_1 = m$, $y_0 = r$, $y_1 = \mathsf{H}_s(r \,\|\, x_1)$ and $y_2 = \mathsf{F}_k(y_1, y_0)$. The dependency matrix and the empty faulty matrix are then given by

$$\begin{pmatrix} \varepsilon & \varepsilon & \varepsilon \\ x_1 & x_2 & \varepsilon \\ \varepsilon & x_2 & y_1 \end{pmatrix}, \qquad \mathcal{F}_{\mathsf{Mac}} = \begin{pmatrix} \varepsilon & \varepsilon & \varepsilon \\ \cdot & \cdot & \varepsilon \\ \varepsilon & \cdot & \cdot \end{pmatrix}.$$

We require no additional protection for Mac_k, i.e., $\mathcal{I}_{\mathsf{Mac}} = \emptyset$. Therefore, a faulty leaky tag generation query has the form $\mathsf{FLMac}_k(m, r, (z_1, z_2, z_3, z_4))$, where the function corresponding to faulty matrix is

$$\mathcal{F}_{\mathsf{Mac}} = \begin{pmatrix} \varepsilon & \varepsilon & \varepsilon \\ z_1 & z_2 & \varepsilon \\ \varepsilon & z_3 & z_4 \end{pmatrix}.$$

Then $\mathcal{F}_{\mathsf{Mac}}(m, (z_1, z_2, z_3, z_4)) = 1$ (i.e., valid query) if and only if $z_1 = \cdot$, as otherwise it comes to trivially replace m by z_1, which is the same as the query $(z_1, (\cdot, z_2, z_3, z_4))$. Thus, we can write $\mathsf{FLMac}_k(m, (z_1, z_2, z_3))$ where z_1, z_2, and z_3 represent the faults injected into y_0 the left n-bit input of H, y_0 the input of the TBC F, and y_1 the hash value.

For the verification function of LR-MACr, we consider the mode-level atomic implementation with $f_1 = \mathsf{H}_s(\cdot)$ and $f_2(\cdot) = \mathsf{F}_k^{-1}(\cdot, \cdot)$. For input $(x_1, x_2, x_3) = (m, r, \tau)$, we thus have $x_1 = m$, $x_2 = r$, $x_3 = \tau$, $y_1 = \mathsf{H}_s(x_1)$ and $y_2 = \mathsf{F}_k^{-1}(y_1, x_3)$. The dependency matrix and the empty faulty matrix are then given by

$$\begin{pmatrix} x_1 & x_2 & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & x_3 & y_1 \end{pmatrix}, \qquad \mathcal{F}_{\mathsf{Vrfy}} = \begin{pmatrix} \cdot & \cdot & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \cdot & \cdot \end{pmatrix}.$$

We require no additional protection for Vrfy_k. Therefore $\mathcal{I}_{\mathsf{Vrfy}} = \emptyset$.

If $\mathsf{L}_{\mathsf{F}} = (\mathsf{L}_{\mathsf{Eval}}, \mathsf{L}_{\mathsf{Inv}})$ is the leakage function pair of the TBC F, we have $\mathsf{L}_{\mathsf{Mac}} = \mathsf{L}_{\mathsf{Eval}}$ as well as $\mathsf{L}_{\mathsf{Vrfy}} = (\mathsf{L}_{\mathsf{Inv}}, y_2)$, since L_{H} gives no more information.

Therefore, a faulty leaky verification query has the form $\mathsf{FLVrfy}_k(m, \tau, (z_1, z_2, z_3, z_4))$, where the function corresponding to faulty matrix is

$$\mathcal{F}_{\mathsf{Vrfy}} = \begin{pmatrix} z_1 & z_2 & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & z_3 & z_4 \end{pmatrix}.$$

Hence $(m, r, \tau, (z_1, z_2, z_3, z_4))$ is a valid verification query if and only if $z_1 = z_2 = z_3 = \cdot$. For simplicity, we assume that there is only a single possible faulty input and write $\mathsf{FLVrfy}_k(m, r, \tau, z)$ where z is the fault injected into z_4, namely the hash value h.

Finally, if $\mathsf{L}_\mathsf{F} = (\mathsf{L}_\mathsf{Eval}, \mathsf{L}_\mathsf{Inv})$ is the leakage function pair of the TBC F, then $\mathsf{L}_\mathsf{Mac} = \mathsf{L}_\mathsf{Eval}$ as well as $\mathsf{L}_\mathsf{Vrfy} = (\mathsf{L}_\mathsf{Inv}, y_2)$, since L_H gives no more information than H does.

Discussion and overview of the proof. The proof is based on the observation that as long as the adversary can only do differential faults, the value $r \oplus \Delta$ still remains random for any differential value Δ. Hence it avoids fault attacks by leveraging the tag generation in LR-MAC1. More concretely, in order to find a forgery, the adversary needs to (1) guess the correct r_i in the tag generation when choosing her differential fault (recall that we work on the fault-and-leak model, the adversary should choose her fault before the evaluation of each query), or find a collision between two r_i and r_j in the tag generation; (2) find a collision against the hash function H; (3) find a preimage for some target value of the hash function H; (4) find a valid prediction against the TBC F.

Proof. As usual, we use a sequence of games to proceed the proof. Denote by E_i the event that the adversary wins the ith Game. Game 0 is exactly the SUF−FL2 game where the adversary (q_{FL}, q_M, q_V, t)-adversary $\mathsf{A}^{\mathsf{L},\mathcal{F}}$ aims at producing a forgery against LR-MAC1.

Game 1 is the same as Game 0 except that we abort if at the i-th tag-generation query, the adversary can somehow guess the correct randomness r_i. If so, then adversary can compute the hash value $h_i = \mathsf{H}_s(r_i \,\|\, m_i)$ locally, and inject a differential fault $\Delta = h_i \oplus h'$ where $h' = \mathsf{H}_s(r_i \,\|\, m')$ to the hash value h_i, and obtain the tag τ_i'. Then (m', r_i, τ_i') is a valid forgery. Recall that for each tag-generation query, the randomness r_i is always selected uniformly at random from $\{0,1\}^n$. Hence, the probability that $r_i = \hat{r}$ is at most $1/2^n$. Summing over at most q_M tag-generation queries,

$$|\Pr[E_0] - \Pr[E_1]| \leq \frac{q_M}{2^n}.$$

Game 2 is the same as Game 1 except that we abort if for two tag generation queries (m_i, r_i, z_i) and (m_j, r_j, z_j) where $z_i = (z_{i,1}, z_{i,2}, z_{i,3})$ and $z_j = (z_{j,1}, z_{j,2}, z_{j,3})$, we have $r_i \oplus z_{i,1} = r_j \oplus z_{j,1}$ or $r_i \oplus z_{i,3} = r_j \oplus z_{j,3}$, namely either the inputs to the hash function H collide or the inputs to the TBC F collide. Note that each r_i is uniformly chosen from the set $\{0,1\}^n$. Hence the probability of any of these two equations hold is $1/2^n$. Summing over at most $\binom{q_M}{2}$ pairs of (i, j), we have

$$|\Pr[E_1] - \Pr[E_2]| \leq \frac{q_M^2}{2^{n+1}}.$$

Game 3 is the same as Game 2 except that we abort if there is a collision in the hash oracle $\mathcal{H}$. We construct an adversary B to bound the difference between these two games. Adversary B plays the collision resistant game against the hash function H, and simulates adversary A's oracle by using its own oracle. Adversary B picks up a key k uniformly at random for the TBC F. For each tag-generation query from A, adversary B will also pick up r_i uniformly at random from the set $\{0,1\}^n$. By using s the key of hash function, k the key of the TBC F, and r_i for each tag-generation query, adversary B can correctly simulates A'oracles. The time complexity of adversary B is $t_1 = t + (q_M + q_V + q_\mathsf{L})(t_\mathsf{F} + t_\mathsf{L} + 2t_\mathsf{H}) + t_\mathsf{H}$. Hence, we have

$$|\Pr[E_2] - \Pr[E_3]| \leq \epsilon_{\mathsf{CR}}.$$

Game 4 is the same as Game 3 except that we abort if in the output tuple $(m_{q_V+1}, r_{q_V+1}, \tau_{q_V+1})$, the value $r_{q_V+1} \,\|\, m_{q_V+1}$ happens to be the preimage of some faulted hash value

$\hat{h}_i$ and $r_i = r_{q_V+1}$ in previous queries and, $\hat{h}_i$ has not been recorded in the hash oracle $\mathcal{H}$. Formally, this difference is captured by the following event

$$r_i = r_{q_V+1} \wedge (*, \hat{h}_i) \notin \mathcal{H} \wedge \hat{h}_i = \mathsf{H}(r_{q_V+1} \parallel m_{q_V+1})$$

This event can be reduced to the preimage resistance of hash function H in the hash oracle that is defined in Definition 8. We then construct another adversary C to bound the probability of this event. Adversary C plays the game against the PRC security of the hash function H, and tries to output a preimage for some target value she selects. She picks up a key k uniformly at random for the TBC F, and simulates A's oracles by using her own oracles. The hash oracle records the hash computation during the interaction of A with her oracles. At the end of the game, when adversary A outputs her forgery $(m_{q_V+1}, r_{q_V+1}, \tau_{q_V+1})$, adversary C outputs $\hat{h}_i$ where $r_i = r_{q_V+1}$ as her target value (if there is any, otherwise adversary C aborts this game), and then outputs $r_{q_v+1} \parallel m_{q_V+1}$ as the image for $\hat{h}_i$. The time complexity of C is $t_2 = t + (q_M + q_V + q_\mathsf{L})(2t_\mathsf{H} + t_\mathsf{F} + t_\mathsf{L})$. Hence,

$$|\Pr[E_3] - \Pr[E_4]| \leq \epsilon_{\mathsf{PRC}}.$$

Game 5 is the same as Game 4 except that we abort if at the i-th verification query (m_i, r_i, τ_i, z_i) by A, we can do a valid prediction against the TBC F, namely $(h_i \oplus z_i, r_i, \tau_i)$ is a valid prediction for F where $h_i = \mathsf{H}_s(m_i)$. Similarly in the proof of $\mathsf{LR\text{-}MAC1}$, we can build a sequence of $q_v + 2$ games Game $4^0, \ldots,$ Game 4^{q_V+1} where Game 4^0 is exactly Game 4 and Game 4^{q_V+1} is exactly Game 5, and construct a $(q_\mathsf{L}, q_M, q_V, t_2)$-$\mathsf{SUP\text{-}L2}$-adversary D^j to bound the difference between any two subgames. The time complexity of D^j is at most $t_3 = t + (q_M + q_V + q_\mathsf{L})(t_\mathsf{H} + t_\mathsf{F} + t_\mathsf{L}) + t_\mathsf{H}$. Hence, by using the hybrid argument,

$$\big|\Pr[E_4] - \Pr[E_5]\big| \leq \sum_{j=0}^{q_V} \big|\Pr[E_4^j] - \Pr[E_4^{j+1}]\big| \leq (q_V + 1)\epsilon_{\mathsf{SUP\text{-}L2}}.$$

For Game 4, since $(h_{q_V+1}, r_{q_V+1}, \tau_{q_V+1})$ cannot be a valid predication against the TBC F, we have $\Pr[E_5] = 0$. Finally, we conclude the proof by

$$\Pr[E_0] = \sum_{i=0}^{4} \big|\Pr[E_i] - \Pr[E_{i+1}]\big| + \Pr[E_5] \leq \epsilon_{\mathsf{CR}} + \epsilon_{\mathsf{PRC}} + (q_V + 1)\epsilon_{\mathsf{SUP\text{-}L2}} + \frac{q_M^2}{2^{n+1}} + \frac{q_M}{2^n}.$$

□

Acknowledgments. Thomas Peters and François-Xavier Standaert are associate researcher and senior research associate of the Belgian Fund for Scientific Research (F.R.S.-FNRS). This work has been funded in parts by the ERC project SWORD (724725).

References

[AOTZ20] Diego F. Aranha, Claudio Orlandi, Akira Takahashi, and Greg Zaverucha. Security of hedged fiat-shamir signatures under fault attacks. In *EUROCRYPT (1)*, volume 12105 of *LNCS*, pages 644–674. Springer, 2020.

[BBB+21] Anubhab Baksi, Shivam Bhasin, Jakub Breier, Mustafa Khairallah, Thomas Peyrin, Sumanta Sarkar, and Siang Meng Sim. DEFAULT: cipher level resistance against differential fault attack. In *ASIACRYPT (2)*, volume 13091 of *LNCS*, pages 124–156. Springer, 2021.

[BBC+20] Davide Bellizia, Olivier Bronchain, Gaëtan Cassiers, Vincent Grosso, Chun Guo, Charles Momin, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. Mode-level vs. implementation-level physical security in symmetric cryptography - A practical guide through the leakage-resistance jungle. In *CRYPTO (1)*, volume 12170 of *LNCS*, pages 369–400. Springer, 2020.

[BBKN12] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proc. IEEE*, 100(11):3056–3076, 2012.

[BCN+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proc. IEEE*, 94(2):370–382, 2006.

[BGP+19] Francesco Berti, Chun Guo, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. Strong authenticity with leakage under weak and falsifiable physical assumptions. In *Inscrypt*, volume 12020 of *LNCS*, pages 517–532. Springer, 2019.

[BGPS21] Francesco Berti, Chun Guo, Thomas Peters, and François-Xavier Standaert. Efficient leakage-resilient macs without idealized assumptions. In *ASIACRYPT (2)*, volume 13091 of *LNCS*, pages 95–123. Springer, 2021.

[BKP+18] Francesco Berti, François Koeune, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. Ciphertext integrity with misuse and leakage: Definition and efficient constructions with symmetric primitives. In *AsiaCCS*, pages 37–50. ACM, 2018.

[BPPS17] Francesco Berti, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. On leakage-resilient authenticated encryption with decryption leakages. *IACR Trans. Symmetric Cryptol.*, 2017(3):271–293, 2017.

[CGLS21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware private circuits: From trivial composition to full verification. *IEEE Trans. Computers*, 70(10):1677–1690, 2021.

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *CHES*, volume 2523 of *LNCS*, pages 13–28. Springer, 2002.

[DEM+20] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, Bart Mennink, Robert Primas, and Thomas Unterluggauer. Isap v2.0. *IACR Trans. Symmetric Cryptol.*, 2020(S1):390–416, 2020.

[DEMS21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2: Lightweight authenticated encryption and hashing. *J. Cryptol.*, 34(3):33, 2021.

[DM21] Christoph Dobraunig and Bart Mennink. Leakage resilient value comparison with application to message authentication. In *EUROCRYPT (2)*, volume 12697 of *LNCS*, pages 377–407. Springer, 2021.

[DMP20] Christoph Dobraunig, Bart Mennink, and Robert Primas. Exploring the golden mean between leakage and fault resilience and practice. *IACR Cryptol. ePrint Arch.*, page 200, 2020.

[DN20] Siemen Dhooghe and Svetla Nikova. My gadget just cares for me - how NINA can prove security against combined attacks. In *CT-RSA*, volume 12006 of *LNCS*, pages 35–55. Springer, 2020.

[FG20] Marc Fischlin and Felix Günther. Modeling memory faults in signature and authenticated encryption schemes. In *CT-RSA*, volume 12006 of *LNCS*, pages 56–84. Springer, 2020.

[GLM+04] Rosario Gennaro, Anna Lysyanskaya, Tal Malkin, Silvio Micali, and Tal Rabin. Algorithmic tamper-proof (ATP) security: Theoretical foundations for security against hardware tampering. In *TCC*, volume 2951 of *LNCS*, pages 258–277. Springer, 2004.

[GLSV14] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, and Kerem Varici. Ls-designs: Bitslice encryption for efficient masked software implementations. In *FSE*, volume 8540 of *LNCS*, pages 18–37. Springer, 2014.

[GR17] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In *EUROCRYPT (1)*, volume 10210 of *LNCS*, pages 567–597, 2017.

[IPSW06] Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and David A. Wagner. Private circuits II: keeping secrets in tamperable circuits. In *EUROCRYPT*, volume 4004 of *LNCS*, pages 308–327. Springer, 2006.

[JT12] Marc Joye and Michael Tunstall, editors. *Fault Analysis in Cryptography.* Information Security and Cryptography. Springer, 2012.

[KR19] Yael Tauman Kalai and Leonid Reyzin. A survey of leakage-resilient cryptography. In *Providing Sound Foundations for Cryptography*, pages 727–794. ACM, 2019.

[LL12] Feng-Hao Liu and Anna Lysyanskaya. Tamper and leakage resilience in the split-state model. In *CRYPTO*, volume 7417 of *LNCS*, pages 517–532. Springer, 2012.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards.* Springer, 2007.

[PSV15] Olivier Pereira, François-Xavier Standaert, and Srinivas Vivek. Leakage-resilient authentication and encryption from symmetric cryptographic primitives. In *CCS*, pages 96–108. ACM, 2015.

[REB+08] Francesco Regazzoni, Thomas Eisenbarth, Luca Breveglieri, Paolo Ienne, and Israel Koren. Can knowledge regarding the presence of countermeasures against fault attacks simplify power attacks on cryptographic devices? In *DFT*, pages 202–210. IEEE Computer Society, 2008.

[RLK11] Thomas Roche, Victor Lomné, and Karim Khalfallah. Combined fault and side-channel attack on protected implementations of AES. In *CARDIS*, volume 7079 of *LNCS*, pages 65–83. Springer, 2011.

[SBD+20] Thierry Simon, Lejla Batina, Joan Daemen, Vincent Grosso, Pedro Maat Costa Massolino, Kostas Papagiannopoulos, Francesco Regazzoni, and Niels Samwel. Friet: An authenticated encryption scheme with built-in fault detection. In *EUROCRYPT (1)*, volume 12105 of *LNCS*, pages 581–611. Springer, 2020.