Authenticity in the Presence of Leakage using a Forkcipher

Francesco Berti[1], François-Xavier Standaert[2], Itamar Levi[1]

[1] Bar-Ilan University, Ramat-Gan, Israel
{francesco.berti, itamar.levi}@biu.ac.il
[2] UCLouvain, ICTEAM/ELEN/Crypto Group
fstandae@uclouvain.be

Abstract. Robust message authentication codes (MACs) and authenticated encryption (AE) schemes that provide authenticity in the presence of side-channel *leakage* are essential primitives. These constructions often rely on primitives designed for strong leakage protection, among others including the use of strong-unpredictable (tweakable) block-ciphers. This paper extends the strong-unpredictability security definition to the versatile and new forkcipher primitive. We show how to construct secure and efficient MAC and AEs that guarantee authenticity in the presence of leakage. We present a leakage-resistant MAC, ForkMAC, and two leakage-resistant AE schemes, ForkDTE1 and ForkDTE2, which use forkciphers instead of traditional secure (tweakable) block-ciphers as compared to the prior art. We prove and analyze their security in the presence of leakage based on a strong unpredictable forkcipher. A comparison with the state-of-the-art in terms of both security and efficiency is followed in the paper. Key advantages and highlights promoted by the proposed constructions are that for the minimal assumptions they require, *unpredictability with leakage-based security*, the tag-generation of ForkMAC is the most efficient among leakage-resilient MAC proposals, equivalent to HBC. ForkDTE 1 and 2 have a more efficient encryption than any other scheme, achieving integrity with leakage (and also providing misuse-resistance).

1 Introduction

One of the main goals of cryptography is to provide *authenticity*. For this purpose, we use Message Authentication Codes (MAC) and Authenticated Encryption Schemes (AE) when in addition to authenticity, privacy is required.

Typically, the security of a scheme is proved against an adversary who interacts with the scheme and obtains its outputs [26]. However, in reality, adversaries and computing devices are in the physical medium, so, we cannot assume that an adversary only receives the outputs of the protocol. Once a cryptographic scheme is implemented on an electronic device, an adversary can also measure the physical quantities involved in the computation, such as time, electromagnetic radiation, power consumption etc. [27,28,32]. From these physical measurements, an adversary can even recover complete secret values (such as the key). Such attacks are called *side-channel attacks* (SCAs). However, it is not always

necessary to recover the key to break a scheme. For example, the well-known AE-scheme OCB can be forged simply by recovering some ephemeral values without any knowledge of the key [8]. In addition, any MAC (or AE scheme), that recomputes the correct tag $\tilde{\tau}$ during verification (or decryption) and checks if it is correct (that is, $\mathsf{Vrfy}_k(m,\tau)$ computes $\tilde{\tau} = \mathsf{Mac}_k(m)$ and checks if $\tilde{\tau} \stackrel{?}{=} \tau$) can be forged simply by recovering $\tilde{\tau}$ via a SCA.

To solve this, one option is to strongly protect (against side-channel adversaries utilizing leakages) both the computation of $\tilde{\tau}$ and the tag-comparison stage [20]. To avoid protecting all these computations, another strategy can be used in the verification: use the inverse of a block-cipher, so that we do not have to avoid recomputing the correct tag, but instead we can perform the check on another value, which, if leaked, will not cause any damage [14].

Many works on authenticity in the presence of leakage assume the existence of a "leak-free" component. Although it may be theoretically possible to obtain such a primitive (e.g., by using a high-order masking protection), such an implementations are extremely costly [24,25,36,17]. Furthermore, it is impossible for an evaluation laboratory to test whether an implementation is leak-free or not since there is no well-defined game. For example, consider the fact that typically all implementations of a block-cipher, even if SCA-protected, trivially leak various parameters from a simple observation: the number of rounds, architectural properties of the block-cipher, information about sub-rounds and number of Sbox executed in parallel, and structural properties of the software or hardware code etc. But is this information meaningful? what part of it is and what is not? So, Berti et al. [9] extended the *unpredictability in the presence of leakage* of Dodis and Steinberger [21] introducing *strong unpredictability in the presence of leakage* (sU–L2) for block-ciphers (BC). Roughly speaking, a BC F is sU–L2 if it is difficult for an adversary to find a fresh and valid couple (input, output), even having oracle access to F_k, its inverse F_k^{-1} and the leakage of all these queries. Using a sU–L2 BC and no other security assumptions in the presence of leakage, it is possible to prove the security of a MAC with both tag-generation and verification leaking in the random oracle model [9], or in the standard model either with a tweakable BC (TBC) (that is, a BC with an additional input, the tweak, allowing more flexibility [29]) or with a strong assumption on the hash [12].

Andreeva et al. [3] introduced a new, efficient, interesting and flexible primitive: *forkciphers* which map N bits into $2N$ bits. The idea is to have two independent pseudo-random permutations, but the cost of computing them is amortized. Therefore, forkciphers use an additional input, $0, 1, \mathsf{b}$ which indicates which output (or both) is wanted. So, if FC is a forkcipher, $\mathsf{FC}_k(x,\mathsf{b}) = (\mathsf{FC}_k(x,0),\mathsf{FC}_k(x,1)) = (y_0,y_1)$, where both $\mathsf{FC}_k(\cdot,0)$, and $\mathsf{FC}_k(\cdot,1)$ are two permutations. We can also define the inverse forkcipher, FC^{-1} for which from y_i outputs x and/or $y_{i\oplus 1}$. Thus, FC^{-1} takes two additional inputs: one input indicating whether we want the inverse (i), the other output (o), or both (b), and another input which is either 0 or 1 to understand which output of FC is the input of FC_k^{-1}. For correctness, $\mathsf{FC}_k^{-1}(\mathsf{FC}_k(x,0),0,\mathsf{o}) = \mathsf{FC}_k(x,1)$ (Fig. 1).

This allows us to have a building block for authenticated encryption schemes that is more efficient and more flexible. Forkciphers have recently been used to build encryption schemes, MAC, AE schemes, pseudorandom-generators and efficient pseudorandom functions [3,1,2,4,5,16,23]. The flexibility of forkciphers allows to build a leakage resilient authenticated encryption scheme, FEDT based on EDT (proposed in [14]) and TEDT (proposed in [10]), where the forkcipher performs the rekeying and produces each time two new refreshed keys [18] (i.e., $k_{2i}\|k_{2i+1} = \mathsf{FC}_{k_i}(i)$ [3]).

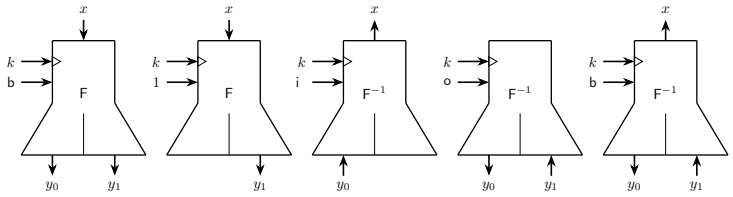

Fig. 1. A schematic illustration describing a forkcipher (Def. 3) [3]. We represent $\mathsf{FC}_k(x, \mathsf{b})$, $\mathsf{FC}_k(x, 1)$, $\mathsf{FC}_k^{-1}(y_0, 0, \mathsf{i})$, $\mathsf{FC}_k^{-1}(y_1, 1, \mathsf{o})$, and $\mathsf{FC}_k^{-1}(y_1, 1, \mathsf{b})$

Contributions. The goal of this paper is to show how we can use forkciphers to build leakage-resilient MACs and AEs to provide authenticity in the presence of leakage. First, we define strong unpredictability (sU–L2) for forkciphers, adapting the definition from [9]. Particularly tricky is the formalization of what are fresh input/output couples.

Second, using a sU–L2 forkcipher FC we can build a leakage-resilient MAC, ForkMAC. The idea is to use one of the FC's outputs as the tag and do the verification on the other. That is, $\tau = \mathsf{FC}_k(\mathsf{H}(m), 0)$, with H a hash function; and we check if $\mathsf{FC}_k^{-1}(\tau, 0, \mathsf{o}) \stackrel{?}{=} \mathsf{FC}_k(\mathsf{H}(m), 1)$.

Third, we prove that using a strongly protected fork-cipher FC, we can construct a nonce-based[4] AE scheme which provides integrity in the presence of leakage in both encryption and decryption. To do this, we start with the DTE2 construction [14], and we use FC to compact the two calls to the leak-free TBC. From the hash of the nonce and the message, we can directly obtain the tag and the first ephemeral key va FC, that is, $(\tau, k_0) = \mathsf{FC}_k(\mathsf{H}(n, m))$, where τ is the tag and k_0 is an ephemeral key. From this ephemeral key, using an encryption scheme based on rekeying, as PSV [31], we can encrypt the message. In decryption, given τ, we can recompute k_0, from which we can recompute the nonce and m.

In the paper we provide two leakage-resilient AE schemes: **(1)** ForkDTE 1 which uses the original check of DTE2 and Hash-then-MAC, that is, checking if

[3] We have simplified their idea, which involves a tweak and a nonce [18].

[4] To avoid using a probabilistic encryption scheme, the encryption takes an additional input, the *nonce*, which should not repeat in different encryption queries [35].

$\mathsf{FC}_k^{-1}(\tau, 0, \mathsf{i}) \stackrel{?}{=} \mathsf{H}(n, m)$, where n and m are retrieved in the verification. ForkDTE 1 uses a single call to FC in decryption, but requires the assumption that FC is leak-free to achieve security; and **(2)** ForkDTE 2 which uses the idea of ForkMAC to establish the validity of a ciphertext. So it needs two calls to FC in decryption. Its security in the presence of leakage is achieved assuming that FC is sU−L2. Finally, we show how we can combine our constructions with the encryption part of FEDT [18].

2 Background

Notations. Let $\{0,1\}^n$ be the set of all the n-bit strings and $\{0,1\}^*$ be the set of all finite strings. Given two strings x and y, let $x\|y$ be their concatenation and $|x|$ be the length of the string x. When we pick x uniformly at random from the set $\mathcal{S}$, we use $x \stackrel{\$}{\leftarrow} \mathcal{S}$. To *parse* a string x in N-bits blocks, we divide x in $x = (x_1, ..., x_\ell)$ with $|x_1| = ... = |x_{\ell-1}| = N$, $|x_\ell| \leq N$, and $x = x_1\|...\|x_\ell$ (ℓ is the *number of blocks* of the string x). Let y be a string and $x \in \mathbb{N}$. With $\pi_x(y)$ we denote the rightmost x bits of the string y. Let $\mathcal{X}$ be a set containing vectors. With $(x, \cdot) \in \mathcal{X}$, we denote that there is an element $(x, y) \in \mathcal{X}$. With $\emptyset$, we denote the empty set.

A $(q_1, ..., q_d, t)$-adversary A is a probabilistic algorithm, which is allowed q_i queries to oracle $\mathcal{O}_i$ and runs in time bounded by t. With $y \leftarrow \mathsf{A}^{\mathcal{O}_1,...,\mathcal{O}_d}(x)$, we denote that adversary A on input x, with access to oracles $\mathcal{O}_1, ..., \mathcal{O}_d$ outputs y.

2.1 Hash Functions, Block-Ciphers, and Forkciphers

To build our schemes we will use hash functions, block-ciphers, and forkciphers.

Hash functions. We use hash functions to compress data. For an adversary, it should be difficult to find a *collision*, (2 different inputs with the same output):

Definition 1. *A hash function* $\mathsf{H} : \mathcal{HK} \times \{0,1\}^* \to \{0,1\}^N$ *is* (t, ϵ)-collision resistant (CR) *if* $\forall$ t*-adversaries* A

$$\Pr[(m_0, m_1) \leftarrow \mathsf{A}(s) \text{ } s.t. \text{ } \mathsf{H}_s(m_0) = \mathsf{H}_s(m_1) \mid s \stackrel{\$}{\leftarrow} \mathcal{HK}] \leq \epsilon.$$

Since the key s of the hash function is public, we may omit it.

Block-ciphers. We use block-ciphers to produce pseudorandom random values.

Definition 2. *A* block-cipher $\mathsf{E} : \mathcal{K} \times \{0,1\}^N \to \{0,1\}^N$ *is a* (q, t, ϵ)-PRP (Pseudo Random Permutation) *if for any* (q, t)*-adversary* A

$$|\Pr[1 \leftarrow \mathsf{A}^{\mathsf{E}_k(\cdot)}] - \Pr[1 \leftarrow \mathsf{A}^{\mathsf{f}(\cdot)}]| \leq \epsilon$$

where $k \stackrel{\$}{\leftarrow} \mathcal{K}$*, and* $\mathsf{f} \stackrel{\$}{\leftarrow} \mathcal{PERM}$*, where* $\mathcal{PERM}$ *is the set of the permutations over* $\{0,1\}^N$*. If* f *is picked from* $\mathcal{FUNC}$*, the set of functions* $\{0,1\}^N \to \{0,1\}^N$*,* E *is a* pseudo-random function*,* PRF*.*

A *tweakable blockcipher* (TBC), Def. 11, App. A, has an additional input, the *tweak*, providing more flexibility. Thus, $\mathsf{E} : \mathcal{K} \times \mathcal{TW} \times \{0,1\}^N \to \{0,1\}^N$, with $\mathsf{E}_k(tw, \cdot)$ a permutation, $\forall(k, tw)$ [29]. We often denote $\mathsf{E}(k, tw, x)$ with $\mathsf{E}_k^{tw}(x)$.

Forkciphers. To have a more flexible primitive than a block-cipher, Andreeva et al. [3] introduced a new primitive, the *fork-cipher*. It takes an input x and produces two outputs, y_0 and y_1. From either y_0 or y_1 we can reconstruct the input x. Formally:

Definition 3 ([3]). *A* forkcipher *is a couple of deterministic algorithms*

$$\mathsf{FC} : \mathcal{K} \times \{0,1\}^N \times \{0,1,\mathsf{b}\} \to \{0,1\}^N \cup (\{0,1\}^N \cup \{0,1\}^N), \text{ and}$$
$$\mathsf{FC}^{-1} : \mathcal{K} \times \{0,1\}^N \times \{0,1\} \times \{\mathsf{b},\mathsf{i},\mathsf{o}\} \to \{0,1\}^N \cup (\{0,1\}^N \cup \{0,1\}^N),$$

s.t. $\forall k \in \mathcal{K}, x \in \{0,1\}^N, j \in \{0,1\}$*:*
- $\mathsf{FC}(k,\cdot,j)$ *is a permutation,*
- $\mathsf{FC}^{-1}(k,\mathsf{FC}(k,x,j),j,\mathsf{i}) = x$*,*
- $\mathsf{FC}^{-1}(k,\mathsf{FC}(k,x,j),j,\mathsf{o}) = \mathsf{FC}(k,x,j\oplus 1)$*,*
- $(\mathsf{FC}(k,x,0),\mathsf{FC}(k,x,1)) = \mathsf{FC}(k,x,\mathsf{b})$*, and*
- $(\mathsf{FC}^{-1}(k,x,j,\mathsf{i}),\mathsf{FC}^{-1}(k,x,j,\mathsf{o})) = \mathsf{FC}^{-1}(k,x,j,\mathsf{b})$

We often use $\mathsf{FC}_k(x,j)$ *for* $\mathsf{FC}(k,x,j)$*.*

We depict a forkcipher in Fig 1. A forkcipher is secure if its outputs are indistinguishable from those of an idealized primitive, that is, one with the same syntax as FC, but implemented with f_0 and f_1, two random permutations.

Definition 4. *A forkcipher* FC *is a* (q_E, q_I, t, ϵ)-pseudo random forkcipher permutation *(*PRFP*) if for any* (q_E, q_I, t)*-adversary* A*,*

$$|\Pr[1 \leftarrow \mathsf{A}^{\mathsf{FC}_k(\cdot,\cdot),\mathsf{FC}_k^{-1}(\cdot,\cdot,\cdot)}] - \Pr[1 \leftarrow \mathsf{A}^{\tilde{F}(\cdot,\cdot),\tilde{F}^{-1}(\cdot,\cdot,\cdot)}]| \leq \epsilon,$$

where $k \xleftarrow{\$} \mathcal{K}$*, and* $\tilde{F}$ *is the ideal version of* FC*, implemented with two permutations* $\mathsf{f}_0, \mathsf{f}_1$ *picked uniformly at random from the set of permutations over* $\{0,1\}^N$*.*

2.2 MACs and Authenticated Encryption Schemes

In the previous section, we have formally defined hash functions, block-ciphers and forkciphers. These are the building blocks of the MAC and AE primitives, which we define here. We use a Message Authentication Code, MAC, to authenticate.

Definition 5. *A* Message Autentication Code (MAC) *is a triple of algorithms* $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ *where*
- *The key-generation algorithm* Gen *generates a key from the sets of keys,* $\mathcal{K}$*.*
- *The tag-generation algorithm* Mac *is a deterministic algorithm which takes as input a key* $k \in \mathcal{K}$*, and a message* $m \in \mathcal{M}$*, and outputs a tag* τ*. We denote this with* $\tau \leftarrow \mathsf{Mac}_k(m)$*.*
- *The verification algorithm* Vrfy *is a deterministic algorithm which takes as input a key* $k \in \mathcal{K}$*, a message* $m \in \mathcal{M}$*, and a tag* τ*, outputs either* $\top$ *("valid") or* $\perp$ *("invalid"). We denote this with* $\top / \perp = \mathsf{Vrfy}_k(m,\tau)$*.*

We require correctness*, that is* $\forall (k,m) \in \mathcal{K} \times \mathcal{M}$*,* $\top = \mathsf{Vrfy}_k(m, \mathsf{Mac}_k(m))$*.*

To authenticate and encrypt we use an authenticated encryption (AE) scheme. We assume that there is an additional input, called the *nonce*, that should not be reused in different encryption query (see [35]).

Definition 6. *A* nonce-based authenticated encryption (nAE) *scheme is a triple of algorithms* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *where*

- *The key-generation algorithm* Gen *generates a key from the sets of keys,* $\mathcal{K}$.
- *The encryption algorithm* Enc *is a deterministic algorithm which takes as input a key* $k \in \mathcal{K}$*, a nonce* $n \in \mathcal{N}$*, and a message* $m \in \mathcal{M}$*, and outputs a ciphertext* $c \in \mathcal{C}$*. We denote this with* $c \leftarrow \mathsf{Enc}_k(n, m)$.
- *The decryption algorithm* Dec *is a deterministic algorithm which takes as input a key* $k \in \mathcal{K}$*, and a ciphertext* $c \in \mathcal{C}$*, and outputs a message* $m \in \mathcal{M}$ *or* $\perp$ (*"invalid"*)*. We denote this with* $\perp / m = \mathsf{Dec}_k(c)$.

We require correctness*, that is* $\forall (k, m) \in \mathcal{K} \times \mathcal{M}$, $m = \mathsf{Dec}_k(c)$ *if* $c \leftarrow \mathsf{Enc}_k(n, m)$ *for any nonce* $n \in \mathcal{N}$.

Here we follow the syntax proposed by Bellare et al. [6], where the nonces are not an input of the decryption algorithm. For simplicity, in the main bulk of the paper, we do not consider here associated data, that is, data that need only to be authenticated, and not encrypted [35]. The appropriate definition (Def. 16) can be found in App. A.

2.3 Authenticity in the Presence of Leakage

Both the previous constructions, MAC and AE, aim to provide authenticity. Here, we give the authenticity definitions in the presence of leakage. We start introducing the leakage and how we model it.

Leakage. Cryptographic algorithms are usually implemented on electronic devices. When an adversary has physical access to an electronic device, not only can she query the oracle O, to get its answer, but she can also access and measure the physical quantities produced during the oracle's computation, as time, power consumption and electronic-magnetic radiation [27,28,32]. We represent this additional information with the leakage function L_O, and we denote that an oracle leaks appending the suffix L to the oracle, that is, OL. Thus, when an adversary has access to a leaking oracle, $\mathsf{A}^{\mathsf{OL}_k}$, and she queries the oracle on input x, she receives the oracle's answer with the leakage function output $\mathsf{L}_\mathsf{O}(x; k)$.

When an adversary can model the leakage of an oracle, we denote this with A^{L}. This means that the adversary can query the leakage function L_O, choosing all the inputs, that is, *both* x and the key k' (k' is different from the key k of the oracle OL_k). These queries correspond to the training phase that can be performed as part of an attack (for example in profiled side-channel analyses) [31].

Now, we move to the security definitions in the presence of leakage. A secure MAC in the presence of leakage, is a MAC for which it is difficult to *forge*, that is to provide a *fresh* and *valid* couple message, tag, even if the adversary has access to MacL, VrfyL and can model the leakage. Formally:

Definition 7 ([9]). *A* $\mathsf{MAC} = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ *with tag-generation leakage function* L_M *and verification leakage function* L_V *is* $(q_L, q_M, q_V, t, \epsilon)$-strongly existentially unforgeable against chosen message and verification attacks with leak-

age in the tag-generation and the verification (sUF–L2) *if for all* (q_L, q_M, q_V, t)*-adversaries* A^{L}*, we have:*

$$\Pr\left[1 \leftarrow \mathsf{FORGEL2}^{\mathsf{suf\text{-}vcma\text{-}L2}}_{\mathsf{MAC},\mathsf{L}_M,\mathsf{L}_V,\mathsf{A}}\right] \leq \epsilon,$$

where the $\mathsf{FORGEL2}^{\mathsf{suf\text{-}vcma\text{-}L2}}$ *experiment is defined in Tab. 1.*

For simplicity, in the proofs, we consider the verification query induced by the final output of the adversary as the $(q_V + 1)^{\text{th}}$ verification query.

The $\mathsf{FORGEL2}^{\mathsf{suf\text{-}vcma\text{-}L2}}_{\mathsf{MAC},\mathsf{L}_M,\mathsf{L}_V,\mathsf{A}^{\mathsf{L}}}$ experiment	
Initialization: $k \leftarrow \mathsf{Gen}$ $\mathcal{S} \leftarrow \emptyset$ Finalization: $(m, \tau) \leftarrow \mathsf{A}^{\mathsf{L},\mathsf{MacL}_k(\cdot),\mathsf{VrfyL}_k(\cdot,\cdot)}$ If $(m, \tau) \in \mathcal{S}$ or $\perp = \mathsf{Vrfy}_k(m, \tau)$ Return 0 Return 1	Oracle $\mathsf{MacL}_k(m)$: $\tau = \mathsf{Mac}_k(m)$ $\mathcal{S} \leftarrow \mathcal{S} \cup \{(m, \tau)\}$ Return $(\tau, \mathsf{L}_M(m; k))$ Oracle $\mathsf{VrfyL}_k(m, \tau)$: Return $(\mathsf{Vrfy}_k(m, \tau), \mathsf{L}_V(m, \tau; k))$

Table 1. The $\mathsf{FORGEL2}^{\mathsf{suf\text{-}vcma\text{-}L2}}_{\mathsf{MAC},\mathsf{L}_M,\mathsf{L}_V,\mathsf{A}}$ experiment (vcma stands for Verification and Chosen Message Attacks).

The definition for authenticity with leakage for nAE schemes is analogous. We want that it will be difficult for an adversary to find a *fresh* and *valid* ciphertext, even if the adversary has access to EncL and DecL. Here, we allow the adversary to *misuse* the nonce, that is, the adversary can repeat the nonce in different encryption queries.

Definition 8 ([14]). *A* nAE*-scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *provides* (q_E, q_D, t, ϵ)*-*ciphertext integrity with nonce misuse and leakage in encryption and decryption *(*CIML2*), if for any* (q_E, q_D, t)*-adversary* A

$$\Pr[c \leftarrow \mathsf{A}^{\mathsf{EncL}_k(\cdot,\cdot),\mathsf{DecL}_k(\cdot)} \mid \textit{s.t. } c \textit{ is fresh and valid}] \leq \epsilon. \tag{1}$$

With fresh *we denote that* c *has never been obtained as an answer from a* $\mathsf{EncL}_k(n, m)$ *query for any* (n, m)*, and with* valid *that* $\mathsf{Dec}_k(c) \neq \perp$*.*

The CIML2 game, is a straightforward adaptation of the $\mathsf{FORGEL2}^{\mathsf{suf\text{-}vcma\text{-}L2}}$ game (Tab. 1) to the nAE -syntax. We depict it in Tab. 4 in App. A.7.

Unbounded leakage model. To give a leakage function that is both: (1) *realistic*, that is, coherent with concrete attacks on actual implementations, and which does not give artificial bounds on what can be leaked (e.g., limiting the number of bits of leakage), and (2) *useful*, that is, which we can use to prove the security of a scheme, is a tough problem. For these reasons [31], the *leveled implementation* has been introduced: there are two types of implementations for the building blocks of a scheme: strongly protected, modeled as leak-free or strongly unpredictable with leakage (sU–L2), and weakly or unprotected. Since the strongly

protected implementations are very slow and costly [24,25,36,17], the idea is to use as few calls to them as possible and to process the bulk of the computations with far lighter implementations in terms of cost. For integrity, we assume that the strongly protected blocks are either leak-free or $\mathsf{sU\text{-}L2}$, that is, they leak the inputs, outputs, but not the secret values (for example, for a block-cipher, the key), moreover, for $\mathsf{sU\text{-}L2}$, there is a leakage of the computation, while weakly or unprotected implementations leak all their inputs and outputs, even the secret ones. This is the so called *unbounded leakage model* [31,13]. Here, we use a forkcipher as a strongly protected component.

The unbounded leakage model has a nice illustrative figure which we explain later with Fig. 3 in respect with our proposed forkcipher based construction.

Note that for privacy, we cannot assume that these weakly protected blocks leaks unboundedly. Since their keys are used few times, we can assume that their leakage is not substantial and it does not lead to a break in the security of the protocol, as typically done with *leveled-implementations* refreshing the keys, for example [31,19,10][5].

3 Strongly Unpredictability with Leakage for Forkcipher

For the security of a forkcipher FC in the presence of leakage L we start from the strong unpredictability definition in the presence of leakage [9] (the natural extension of unpredictability [21,22]) for block-ciphers, and we adapt to forkciphers. Roughly speaking we want that an adversary cannot produce a fresh and valid triple (input, selector, output), even if she can model the leakage, has oracle access to FC with its leakage, and FC^{-1} with its leakage. Let (x, sel, y) be the prediction of the adversary, $x, y \in \{0,1\}^N$, $sel \in \{0,1,\mathsf{o}\}$. We deem (x, sel, y) *valid*, if $\mathsf{FC}_k(x, sel) = y$ if $sel \in \{0,1\}$, otherwise if $\mathsf{FC}_k^{-1}(x, 0, \mathsf{o}) = y$.

Particularly tricky is to precisely formalize what means *fresh*. For example, if an adversary has only called FC_k on input $(x, 0)$, we should deem $(x, 1, z)$ fresh. On the other hand, if an adversary has called FC_k on input $(x, 0)$ obtaining y, then she has called FC_k^{-1} on input $(y, 0, \mathsf{o})$ obtaining z, if she outputs $(x, 1, z)$, which is correct, we deem this prediction *invalid*. We give an illustrative representation of this in Fig. 2. For simplicity, we do not allow the adversary to output $(x, (y_0, y_1), \mathsf{b})$ as her prediction because if this is valid, that is, $\mathsf{FC}_k(x, \mathsf{b}) = (y_0, y_1)$ and fresh, then the adversary could also win with one of these two predictions: $(x, 0, y_0)$ or $(x, 1, y_1)$. We formalize this in the following:

Definition 9 ($\mathsf{sU\text{-}L2}$). *A forkcipher* $\mathsf{FC} : \mathcal{K} \times \{0,1\}^N \times \{0,1,\mathsf{b}\} \to \{0,1\}^N \cup (\{0,1\}^N \cup \{0,1\}^N)$ *with leakage function pair* $\mathsf{L} = (\mathsf{L}_{\mathsf{FC}}, \mathsf{L}_{\mathsf{FC}^{-1}})$ *is* $(q_{\mathsf{L}}, q_F, q_{F^{-1}}, t, \epsilon)$ strongly unpredictable with leakage in evaluation and inversion *($\mathsf{sU\text{-}L2}$), if for any* $(q_{\mathsf{L}}, q_{\mathsf{FC}}, q_{\mathsf{FC}^{-1}}, t)$*-adversary* A*, we have* $\Pr[1 \leftarrow \mathsf{sU\text{-}L2}_{\mathsf{A},\mathsf{FC},\mathsf{L}}] \leq \epsilon$*, where the* $\mathsf{sU\text{-}L2}$ *experiment is defined in Tab. 2, and where* A^{L} *makes at most* q_{L} *(offline) queries to* L.

[5] The concept of lightly protected primitives (as compared to no protection) was already practiced in [31,19]

The $\mathsf{sU\text{-}L2}_{\mathsf{A},\mathsf{FC},\mathsf{L}}$ experiment.	
Initialization: $k \xleftarrow{\$} \mathcal{K}$ $\mathcal{C} \leftarrow \emptyset$ Finalization: $(x, sel, y) \leftarrow \mathsf{A}^{\mathsf{L},\mathsf{FCL}_k(\cdot,\cdot),\mathsf{FC}^{-1}\mathsf{L}_k(\cdot,\cdot,\cdot)}$ If $0 = \mathsf{Fresh}((x, sel, z), \mathcal{C})$ Return 0 If $y = \mathsf{FC}_k(x, sel)$ Return 1 Return 0	Oracle $\mathsf{FCL}_k(x, sel)$: $y = \mathsf{FC}_k(x, sel)$ $\mathsf{leak} = \mathsf{L}_{\mathsf{FC}}(x, sel; k)$ $\mathcal{C} \leftarrow \mathsf{Add}((x, sel, y, \mathsf{d}), \mathcal{C})$ Return (y, leak) Oracle $\mathsf{FC}^{-1}\mathsf{L}_k(x, sel, sel')$: $y = \mathsf{FC}_k^{-1}(x, sel, sel')$ $\mathsf{leak} = \mathsf{L}_{\mathsf{FC}^{-1}}(x, sel, sel'; k)$ $\mathcal{C} \leftarrow \mathsf{AddI}((x, sel, sel', y), \mathcal{C})$ Return (y, leak)

Table 2. Strong unpredictability with leakage in evaluation and inversion experiment. The Add, Fresh, and AddI and Fresh algorithms are depicted in Tab. 5, and Tab. 6 in App. D.

Fig. 2. A schematic description of fresh and not fresh queries options for forkciphers. With black (solid) arrows we denote queries done by the adversary, with blue (dashed) arrows fresh predictions, while with red (dotted) not fresh predictions.

To distinguish between fresh and not fresh predictions, we use a set $\mathcal{C}$ to keep in memory the queries done by the adversary. The idea is to keep in memory (x, y_0, y_1) for any query with $\mathsf{FC}_k(x, \mathsf{b}) = (y_0, y_1)$. But, when a query does not give all these values, we flag the missing value with gu (which stands for this value can be guessed). When a new query to either FC or FC^{-1} is done, first, we see if it completes a previous query (thus removing the flag gu and replacing it with the correct value), otherwise we add the obtained values as a new triple. The Add oracle takes the set $\mathcal{C}$ and add the input and outputs of an FC query, while the AddI oracle does the same for FC^{-1} queries. We deem a prediction as fresh if either there is no entry in $\mathcal{C}$ with any of the two values of the prediction (in the exact places) or if there is an entry with one value in the correct place and the other value is deemed guessable (gu). We do this with the oracle Fresh. These oracles are described in Tab. 5 and Tab. 6 in App. D.

4 ForkMAC, a MAC based on a Forkcipher

Here, we show that we can use a forkcipher to build a MAC which is secure in the presence of leakage in the unbounded leakage model. We start introducing the scheme, then, we prove its security.

4.1 Description of ForkMAC

One of the main challenges in designing a leakage-resilient MAC is the verification algorithm. Often, the verification queries are done by recomputing the correct tag, and comparing this value with the tag provided to assess the validity of the query (systematic). That is, for example, in the well-known Hash-then-MAC [26], the Mac first hashes the message and then uses a block-cipher with the hash as input to compute the tag (that is, $\tau = \mathsf{E}_k(\mathsf{H}(m))$). For verification, simply on input (m, τ), the algorithm checks if $\tilde{\tau} = \mathsf{E}_k(\mathsf{H}(m)) \stackrel{?}{=} \tau$. This comparison may be attacked when the adversary exploits leakage, and needs to be protected [20,14]. In the unbounded leakage model every verification algorithm that recomputes the correct tag is insecure since the correct tag is leaked. Thus, if we want a leakage-resilient MAC in the unbounded leakage model, we need the verification algorithm to perform its *check* on something else.

Berti et al. [14] proposed with HBC2 to exploit the inverse of the block-cipher in verification for the Hash-then-MAC: for a verification query on input (m, τ), they check if $\mathsf{H}(m) = \mathsf{E}_k^{-1}(\tau)$. The idea is that if E is a leak-free block-cipher, $\tilde{h} = \mathsf{E}_k^{-1}(\tau)$ is random, thus, even the adversary knows $\tilde{h}$, she cannot forge because she should find a pre-image for a random value, and it is enough to assume that the hash function is range-oriented pre-image resistant (Def. 10, App. A). If E is sU–L2, the previous MAC is leakage -resistant either in the random oracle model [9] or in the standard model adding a strong hypothesis on the hash function [12].

Forkciphers allows us to find a different value to check the validity of a verification query: we use $\mathsf{FC}_k(\mathsf{H}(m), 0)$ as the tag and in verification we check if $\mathsf{FC}_k(\mathsf{H}(m), 1) = \mathsf{FC}_k^{-1}(\tau, 0, \mathsf{o})$. If an adversary gets $\mathsf{FC}_k(\mathsf{H}(m), 1)$, she has obtained nothing because still she needs to guess $\mathsf{FC}_k(\mathsf{H}(m), 0)$. This is the idea behind ForkMAC which is detailed in Alg. 1 and Fig. 3 (the tag-generation is depicted in Fig. 5, in App. E).

4.2 sUF–L2 Security of ForkMAC

Now, we prove that our ForkMAC is leakage resistant. First, we define the leakage functions of Mac and Vrfy, then we give the intuition for security (a full proof is provided in the appendix).

Leakage functions. We assume that only the forkcipher is strongly protected. Let $\mathsf{L}_{\mathsf{FC}}(x, sel; k)$ and $\mathsf{L}_{\mathsf{FC}^{-1}}(x, sel, sel'; k)$ be its leakage functions. Then, according to the unbounded leakage model

- $\mathsf{L}_{\mathsf{Mac}}(m; k) := (h, \mathsf{L}_{\mathsf{FC}}(h, 0; k))$ with $h = \mathsf{H}_s(m)$.
- $\mathsf{L}_{\mathsf{Vrfy}}((m, \tau); k) := (h, v, \tilde{v}, \mathsf{L}_{\mathsf{FC}}(h, 1; k), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k))$ with $v = \mathsf{FC}_k(h, 1)$ and $\tilde{v} = \mathsf{FC}_k^{-1}(\tau, 0, \mathsf{o})$.

Now, we can state the leakage-resistance of ForkMAC

Theorem 1. *Let* $\mathsf{FC} : \mathcal{K} \times \{0,1\}^N \times \{0,1,\mathsf{b}\} \to \{0,1\}^N \cup (\{0,1\}^N)^2$ *be a* $(2q_L, q_M + q_V, q_V, t_1, \epsilon_{\mathsf{sU\text{-}L2}})$*-strongly unpredictable forkcipher. Let* $\mathsf{H} : \mathcal{HK} \times$

Algorithm 1 ForkMAC, a sUF−L2-secure MAC based on a forkcipher.

– Gen:
- $k \xleftarrow{\$} \mathcal{K}$
- $s \xleftarrow{\$} \mathcal{HK}$ (s is a public parameter)

– $\mathsf{Mac}_k(m)$:
- $h = \mathsf{H}_s(m)$ // digest
- $\tau = \mathsf{FC}_k(h, 0)$ // tag
- Return τ

– $\mathsf{Vrfy}_k(m, \tau)$:
- $h = \mathsf{H}_s(m)$
- $v = \mathsf{FC}_k(h, 1)$
- $\tilde{v} = \mathsf{FC}_k^{-1}(\tau, 0, \mathsf{o})$
- If $v = \tilde{v}$ Return $\top$, Else Return $\perp$

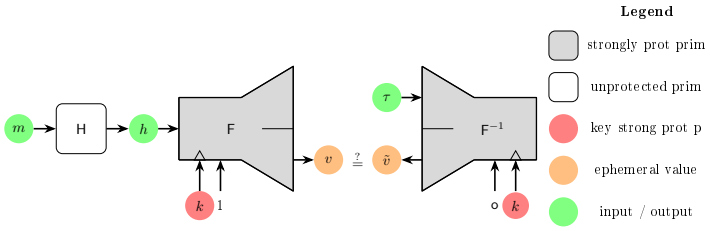

Fig. 3. The verification of ForkMAC - Alg. 1 (the tag-generation algorithm can be found in Fig. 5, App. E). We have a leveled implementation (Sec. 2.3), which is graphically represented. We distinguish between strongly protected primitive (in gray), and unprotected (in white). In the unbounded leakage model, we give to the adversary the ephemeral values (the orange ones) via leakage. We assume that only the keys of the strongly protected primitives are protected (these are the red ones) and not leaked.

$\{0,1\}^* \to \{0,1\}^N$ *be a* $(t_2, \epsilon_{\mathsf{CR}})$*-collision resistant hash function. Then,* ForkMAC *with the leakage function described above is* $(q_L, q_M, q_V, t, \epsilon)$-sUF−L2 MAC, *with*

$$\epsilon \leq \epsilon_{\mathsf{CR}} + [(q_V)^2 + 1]\epsilon_{\mathsf{sU-L2}},$$

$t_1 = t + (q_L + q_M + q_v + 1)t_{\mathsf{H}}$, $t_2 = t + (q_M + q_V + 1)t_{\mathsf{H}} + (q_M + 2q_V)t_{\mathsf{FC}}$, *where* t_{H} *is the time needed to evaluate once the hash function, and* t_{FC} *is the time to evaluate once* FC *and collect its leakage.*

Here, we have a worse bound $[(q_V)^2 + 1]\epsilon_{\mathsf{sU-L2}}$ instead of the expected $(q_V + 1)\epsilon_{\mathsf{sU-L2}}$ because the adversary has an additional winning strategy: finding a collision between v^i and $\tilde{v}^j$ for two different verification queries, the i^{th} with $v^i = \mathsf{FC}_k(h^i, 1)$, and the j^{th} where $\tilde{v}^j = \mathsf{FC}_k^{-1}(\tau^i, 0, \mathsf{o})$. Note that without leakage, the security of this scheme would have been the standard security (as proved in Thm. 4, App. B.2),

$$\epsilon = \epsilon_{\mathsf{PRF}} + \epsilon_{\mathsf{CR}} + \frac{q_V + 1}{2^N}.$$

Idea of the security argument. Let (m, τ) be the first fresh and valid verification query (if there is one, the adversary can simply output it as her forgery) and let $h = \mathsf{H}(m)$. There are two cases:

- There is a previous tag-generation query with input m' s.t $\mathsf{H}(m') = h' = h$
- There is no such tag-generation query.

We can easily treat both cases. In fact:

- If $h' = h$ we have found a collision for the hash function.
- Otherwise, either $(h, 0, \tau)$ is a fresh and valid prediction or it is not fresh.
  - If it is fresh, then, we have broken sU−L2.
  - If it is not fresh, this means that there have been two previous verification queries one on input (m^i, τ^i) and one on input (m^j, τ^j) s.t. $m^i = m$ and $\tau^j = \tau$ s.t. $v^i = \tilde{v}^j$. We have two possible situations: $i < j$ or $i > j$ (if $j = i$, then (m^i, τ^i) would have been the first fresh and valid verification queries, contrary to our hypothesis). Thus,
    - if $i < j$, then, in the jth verification query the prediction $(\tau^j, v^i, \mathsf{o})$ would have broken the sU−L2 security of FC_k. In the jth query there are at most $j - 1$ possible target for v^i.
    - if $i > j$, then, in the ith verification query the prediction $(h^i, \tilde{v}^j, 1)$ would have broken the sU−L2 security of FC_k. In the ith query there are at most $i - 1$ possible target for $\tilde{v}^j$.

In the proof, which can be found in App. B.1, we present the ideas slightly differently to have a better security bound.

4.3 Comparison with Other Leakage Resistant MACs

Notably, BCs implementations were deeply investigated in literature. Consequently plethora of implementations, performance- and security- analysis exist. Forkciphers on the other hand are new, whereas only few implementations proposals exist. Meaning comparison in terms of implementation efficiency is hardly fair. Even-though, in this section we compare security aspects in a high-level; we supplement this comparison with so called rough efficiency analysis. We make the general assumption (similarly to [3,1,2,4]) that a forkcipher is cheaper than a blockcipher *per the functionality it provides*, that protecting a forkcipher is cheaper than protecting a blockcipher *per the functionality it provides*. I.e., whereas a BC and a $\mathsf{FC}(\cdot, 0)$ are equivalent, a FC provides more functionality, with an amortized cost [3,1,2,4], especially in the full (e.g.,) AE or MAC levels. And similarly, that an unsecured forkcipher is less expensive than an unsecured blockcipher, again per the functionality it provides.

With respect to HBC2 [14] ForkMAC achieves the same black-box security. We can prove its sUF−L2-security assuming that FC is sU−L2 in the standard model without any ideal (or strong) hypothesis on the hash function H. On the other hand, it is less efficient in verification because we need two calls to the forkcipher with respect to a single call to the blockcipher, and sUF−L2-security is worse when we assume that the strongly protected component is leak-free.

With respect to HTBC [9] (Alg. 6, App D), a version of HBC, where a tweakable block-cipher (TBC) is used, ForkMAC will achieve worse security bound in black-box (because HTBC is beyond-birthday) as we are bounded by the collision resistance of the hash function. However, we believe ForkMAC is more efficient in tag-generation since we are comparing a single forkcipher call to a single TBC call. Moreover, for HTBC, as for HBC, it is impossible to provide sUF−L2-security in the standard model when the TBC is sU−L2 without a strong hypothesis on H. Inspired by HTBC, we can build ForkTMAC (Alg. 4, App D) a variant of ForkMAC which is based on a tweakable forkcipher, Def. 11). ForkTMAC provides beyond-birthday security black-box.

With respect to the LR−MAC [12] which is sUF−L2-secure in the standard model using a sU−L2 TBC, we have the same black-box security, but worse sUF−L2-security. However, we believe, ForkMAC is more efficient in tag-generation since we are using a single call to a forkcipher, with respect to a single call to a TBC. Using a tweakable forkcipher with tweak as big as the block we can have beyond birthday blackbox security, while LR−MAC needs a TBC whose tweaks have twice the size of its blocks.

Finally, ForkMAC is much more efficient than ISAPMAC [19] which is a leakage resilient MAC since, here we have only a call to a forkcipher to achieve leakage protection while ISAPMAC execute n rounds of a sponge. I.e., to prevent differential power attack susceptibility the key is absorb bit-by-bit by the sponge. Moreover, ISAPMAC have to protect the comparison in verification using a permutation-based value processing function [20].

5 ForkDTE - Authenticated Encryption

In this section, we show that we can use a forkcipher as a strongly protected component to build a nAE-scheme.

5.1 Overview of ForkDTE

DTE *[13] and* DTE2 *[14]:* We start from the DTE (Digest-Tag-and-Encrypt) nAE encryption scheme [13]. DTE starts from a leakage-resistant encryption scheme, PSV (detailed in Alg. 3 in App. D, and in Fig. 6) [31].

- PSV [31]: from a first ephemeral key k_1 (which is generated from the master key), a pseudo-random value y_1 is created, $y_1 = \mathsf{FC}_{k_1}(P_B)$ which is XORed to the first block of the message, generating the first ciphertext block $c_1 = y_1 \oplus m_1$, then, the key k_1 is *refreshed*, $k_2 = \mathsf{FC}_{k_1}(p_A)$ (p_A and p_B are two public values). Iterating, we can encrypt the full message.
- DTE [13] (Digest, Tag and Encrypt): it aims to use PSV to build a leakage-resistant and nonce misuse-resistant (Def. 15) nAE-scheme (Def. 6). The idea is to first use the well-known Hash-then-MAC paradigm on the nonce and the message, to compute the tag τ. Thus, they digest (n, m) with H obtaining the digest $h = \mathsf{H}(n\|m)$, then, they tag it $\tau = \overline{\mathsf{E}_k(h)}$. Finally, *the nonce and the message* are encrypted using PSV. From τ the first ephemeral key $k_0 = \mathsf{E}_k(\tau)$

is generated, and then the encryption follows PSV. In decryption, from the tag τ, k_0 is recomputed, then, the couple nonce, message (n, m) is retrieved and it is checked to verify τ is the correct tag. Only the two calls of E using k as key (that is, the one to generate the tag τ and the one to generate the first ephemeral key k_0), must be strongly protected against leakage. DTE provides ciphertext integrity in the presence of leakage in encryption in the unbounded leakage model [13] (and privacy in the presence of leakage), but it does not provide CIML2 in the unbounded leakage model since the tag is recomputed in verification. DTE is misuse-resistant because, roughly speaking, every bit of the ciphertext depend on all the plaintext since all ephemeral keys depend on $h = \mathsf{H}(n\|m)$ [13].

– DTE2 [14]: The CIML2-security can be obtained using the idea of inverting the E in decryption, as it was done for Hash-then-MAC (see Sec. 4). Encryption is done as in DTE, while in decryption as before we start from τ to recompute k_0 and retrieve both (n, m). But to verify the validity of the ciphertext we compute $\tilde{h} = \mathsf{E}_k^{-1,0}(\tau)$ and we check if $\tilde{h} \stackrel{?}{=} h = \mathsf{H}(n\|m)$ (for security reasons, we cannot use E_k to both compute τ and k_0, instead we separate these two calls using a tweakable blockcipher, with a single bit tweak: we use the tweak 0, E_k^0 to generate τ and the tweak 1, E_k^1 for k_0). This is DTE2 [14] (Alg. 2). It is CIML2-secure in the unbounded leakage model if E is leak-free [13], or if E is sU–L2 in the random oracle model [9] or in the standard model with a strong hypothesis on H [12].

ForkDTE*:* In ForkDTE we replace the two calls to the strongly protected tweakable block-cipher E with calls to a strongly protected forkcipher FC. In encryption from h instead of computing $\tau = \mathsf{E}_k^0(h)$ and $k_0 = \mathsf{E}_k^1(\tau)$, we can simply compute $(\tau, k_0) = \mathsf{FC}_k(h, \mathsf{b})$. In decryption we have two choices:

1. ForkDTE1, we use only once FC and we compute $(\tilde{h}, k_0) = \mathsf{FC}_k^{-1}(\tau, 0, \mathsf{b})$. From k_0 and $\tilde{h}$ we decrypt as for DTE2.
2. ForkDTE2, using the idea of ForkMAC (Sec. 4) we use twice FC: once to recompute k_0, $k_0 = \mathsf{FC}_k(\tau, 0, \mathsf{o})$, then, from k_0, we retrieve (n, m) and we compute h. Finally, we check if the first ephemeral key is the right one given n, m and we compute $\tilde{k_0} = \mathsf{FC}_k(h, 1)$ and we check if it is equal to the k_0 we have obtained.

We describe ForkDTE1, and 2, and DTE2 in Alg. 2. We depict the encryption in Fig. 4, the encryption of DTE2 in Fig. 6, the decryption of ForkDTE1 in Fig. 7, and the decryption of ForkDTE2 in Fig. 8 (some of these figures can be found in App. E). Note that from a functional point of view, the decryptions of ForkDTE 1 and 2 are equivalent, that is, these algorithms with the same input, give the same result. We have given two different algorithms because their efficiency and security in the presence of leakage is different. Finally, in Alg. 7 we show that we can use the leakage-resilient encryption scheme proposed with FEDT [18] which uses forkciphers.

Algorithm 2 The ForkDTE1, ForkDTE2, and DTE2 [14] algorithms. DTE2 uses the dashed-lines-boxed instructions, both ForkDTE 1 and 2 the double boxed instruction. ForkDTE1 uses also the boxed instructions, while ForkDTE2 the stacked-dashed double box ones.

– Gen:
- $k \xleftarrow{\$} \mathcal{K}$ — DTE 2 | ForkDTE1 and 2 | ForkDTE1 | ForkDTE2
- $s \xleftarrow{\$} \mathcal{HK}$
- $p_A, p_B \xleftarrow{\$} \{0,1\}^N$ (s, p_A, p_B are public parameters)

– $\mathsf{Enc}_k(n, m)$:
- $h = \mathsf{H}_s(n \| m)$ digest
- $\tau = \mathsf{E}^0_k(h)$ tag
- $k_0 = \mathsf{E}^1_k(\tau)$ generate the first ephemeral key
- $(\tau, k_0) = \mathsf{FC}_k(h, \mathsf{b})$ tag and generate the first ephemeral key
- Parse $m = (m_1, m_2, \ldots, m_\ell)$ in N-bit blocks ...and encrypt
- $y_0 = \mathsf{E}_{k_0}(p_B)$
- $c_0 = y_0 \oplus n$
- For $i = 1, \ldots l$
  - $k_i = \mathsf{E}_{k_{i-1}}(p_A)$
  - $y_i = \mathsf{E}_{k_i}(p_B)$
  - $c_i = \pi_{|m_i|}(y_i) \oplus m_i$
- $C = (c_0, c_1, \ldots, c_\ell)$
- Return $c = (\tau, C)$

– $\mathsf{Dec}_k(c)$:
- Parse $c = (\tau, C)$ with $|\tau| = N$
- Parse $C = (c_0, c_1, c_2, \ldots, c_\ell)$ in N-bit blocks
- $k_0 = \mathsf{E}^{-1,1}_k(\tau)$ Recovering the first ephemeral key
- $(\tilde{h}, k_0) = \mathsf{FC}^{-1}_k(\tau, 0, \mathsf{b})$ Recovering the first ephemeral key and check value
- $k_0 = \mathsf{FC}^{-1}_k(\tau, 0, \mathsf{o})$ Recovering the first ephemeral key
- $y_0 = \mathsf{E}_{k_0}(p_B)$
- $n = y_0 \oplus c_0$
- For $i = 1, .., \ell$
  - $k_i = \mathsf{E}_{k_{i-1}}(p_A)$
  - $y_i = \mathsf{E}_{k_i}(p_B)$
  - $m_i = \pi_{|c_i|}(y_i \oplus c_i$
- $(n, m) = (n, (m_1, ..., m_\ell))$
- $h = \mathsf{H}_s(n \| m)$
- $\tilde{h} = \mathsf{FC}^{-1,0}_k(\tau)$ check value
- If $h = \tilde{h}$ Return m; Else Return $\perp$
- If $h = \tilde{h}$ Return m; Else Return $\perp$
- $\tilde{k_0} = \mathsf{FC}_k(h, 1)$ check value
- If $k_0 = \tilde{k_0}$ Return m; Else Return $\perp$ check value

5.2 Security of ForkDTE 1 and 2

Here, we give the security properties of ForkDTE 1 and 2.

Idea of the black-box security. The black-box security of ForkDTE 1 and 2 is the same as DTE2. Thus, both ForkDTE 1 and 2 are secure nAE-schemes (Def. 14), and both are misuse-resistant (Def. 15). Here we give a simple argument which justify the previous statement. Consider the following construction:

$$\mathsf{E}_k^{tw}(x) := \begin{cases} \mathsf{FC}_k(x,0) & \text{if } tw = 0 \\ \mathsf{FC}_k^{-1}(x,0,\mathsf{o}) & \text{if } tw = 1 \end{cases}$$

is a secure TBC if FC is a secure forckcipher. On the other hand, given E a secure TBC, the following construction $\widetilde{\mathsf{FC}}_k(x,\mathsf{b}) := (\mathsf{E}_k^0(x), \mathsf{E}_k^1(\mathsf{E}_k^0(x)))$ is a secure forkcipher. From the previous argument and [13,14] we obtain the claimed security. To improve the quantitative bounds, we prove the nAE-security and misuse-security in App. B.6 and App. B.5, respectively with a direct proof.

CIML2 *security of* ForkDTE *1.* The fact that DTE2 and ForkDTE 1 are the same if we are using the constructions for E and FC described before implies that in the unbounded leakage model ForkDTE 1 is CIML2-secure if FC is leak-free. If FC is sU−L2, applying the result to Hash-then-MAC of [9] or [12], we obtain the CIML2 security of ForkDTE 1 in the random oracle model, or in the standard model with a strong hypothesis on H, respectively. Here, we only give an idea of the CIML2 security of ForkDTE1 when FC is leak-free (clearly it is strongly inspired by [14]). First, we need to give the leakage functions for encryption and decryption: $\mathsf{L}_{\mathsf{Enc}}$ and $\mathsf{L}_{\mathsf{Dec}}$. According to the unbounded leakage model

- $\mathsf{L}_{\mathsf{Enc}}((n,m);k) := (h,k_0)$ with $h = \mathsf{H}_s(m)$.
- $\mathsf{L}_{\mathsf{Dec}}(c;k) := (\tilde{h},k_0)$ with $(\tilde{h},k_0) = \mathsf{FC}_k^{-1}(h,0,\mathsf{b})$.

Note that from k_0 all ephemeral values y_i, k_i can be recomputed in both encryption and decryption and that FC does not leak since it is leak-free.

Here, we give a security result in the standard model assuming that the forkcipher FC is leak-free. For the security proof, similarly to [14], we need to assume that for the hash function H it is hard to find a pre-image for a random value. This is the *range-oriented pre-image resistance* (Def. 10, App. A).

Theorem 2. *Let* FC *be a* $(q_E, q_D+1, t_1, \epsilon_{\mathsf{PRFP}})$*-pseudo random forkcipher permutation whose implementation is leak-free. Let* H *be a* $(t_2, \epsilon_{\mathsf{CR}})$*-collision resistant and* $(t_2, \epsilon_{\mathsf{ro\text{-}PR}})$*-range-oriented-pre-image resistant hash function. Then* ForkDTE1 *is* (q_E, q_D, t, ϵ)*-*CIML2*-secure with*

$$\epsilon \leq \epsilon_{\mathsf{PRFP}} + \epsilon_{\mathsf{CR}} + q_D\epsilon_{\mathsf{ro\text{-}PR}} + (q_D+1)2^{-N},$$

where ForkDTE1 *encrypts at most* Ln*-bits message,* $t_1 = t + (q_E+q_D+1)[t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}]$*,* $t_2 = t + (q_E+q_D+1)[2t_{\mathsf{f}} + t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}]$*, with* t_{H} *the time needed to execute once the hash function* H*,* t_{E} *to execute* E*, and* t_{f} *to randomly sample a random permutation.*

Idea of the proof. (The complete proof is in App. B.3) To every decryption query, we associate the couple nonce-message (n, m) retrieved, and $h = \mathsf{H}(n, m)$, $\tilde{h} = \mathsf{FC}_k^{-1}(\tau, 0, \mathsf{i})$. Let c^* be the first fresh and valid decryption query. There are three cases:

1. There is a previous encryption query with input (n', m') such that $\mathsf{H}(n'\|m') = h' = h^*$
2. There is no encryption query s.t. $h = h^*$, but there is a previous decryption query, the i^{th} s.t. $\tilde{h}^i = h^*$.
3. None of the previous cases.

We can easily treat all cases. In fact:

1. If $h' = h^*$ we have found a collision for the hash function.
2. Since FC is a PRFP all $\tilde{h}$ are random. Thus, we have found a hash pre-image for a random $\tilde{h}$.
3. Since FC is a PRFP and we have never queried $\mathsf{FC}_k^{-1}(\tau, 0, \mathsf{i})$ (otherwise, we would have been in the previous case), the probability that given the digest h we compute in decryption, $h = \mathsf{FC}_k^{-1}(\tau, 0, \mathsf{i})$ is negligible.

$\mathsf{CIML2}$ *security of* $\mathsf{ForkDTE}$ *2.* First, we need to give the leakage functions for encryption and decryption: $\mathsf{L_{Enc}}$ and $\mathsf{L_{Dec}}$. According to the unbounded leakage model

- $\mathsf{L_{Enc}}((n, m); k) := (h, k_0, \mathsf{L_{FC}}(h, \mathsf{b}; k))$ with $h = \mathsf{H}_s(m)$.
- $\mathsf{L_{Dec}}(c; k) := (\tilde{h}, k_0, \mathsf{L_{FC^{-1}}}(\tau, 0, \mathsf{o}; k), \mathsf{L_{FC}}(h, 1; k))$ with $(\tilde{h}, k_0)v = \mathsf{FC}_k^{-1}(h, 0, \mathsf{b})$.

Note that from k_0 all ephemeral values y_i, k_i can be recomputed. Here, we give a proof assuming that FC is $\mathsf{sU\text{–}L2}$.

Theorem 3. *Let* FC *be a* $(2q_L, q_E + q_D + 1, q_D + 1, t_1, \epsilon_{\mathsf{sU\text{-}L2}})$*-strongly unpredictable forkcipher in the presence of leakage. Let* H *be a* $(t_2, \epsilon_{\mathsf{CR}})$*-collision resistant. Then* $\mathsf{ForkDTE1}$ *is* $(q_L, q_E, q_D, t, \epsilon)$*-*$\mathsf{CIML2}$*-secure with*

$$\epsilon \leq \epsilon_{\mathsf{CR}} + [(q_D)^2 + 1]\epsilon_{\mathsf{sU\text{-}L2}},$$

where $\mathsf{ForkDTE2}$ *encrypts at most* Ln*-bits message,* $t_1 = t + (q_L + q_E + q_D + 1)[t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}]$, $t_2 = t + (q_E + q_D + 1)t_{\mathsf{H}} + (q_E + 2q_D + 2)t_{\mathsf{FC}} + (q_E + q_D + 1)(2L+1)t_{\mathsf{E}}$, *with* t_{H} *the time needed to execute once the hash function* H*,* t_{E} *to execute* E*, and* t_{f} *to randomly sample a random permutation.*

Idea of the security. We observe that we can reduce the $\mathsf{CIML2}$ security of $\mathsf{ForkDTE2}$ to the $\mathsf{sUF\text{–}L2}$-security of $\mathsf{ForkMAC}$ (Sec. 4) simply observing that an adversary against $\mathsf{ForkMAC}$ can simulate $\mathsf{ForkDTE2}$ simply asking in addition $\mathsf{FC}_k(h, 1)$ in every encryption query because from k_0 in both encryption and decryption, she can compute C and and (n, m) respectively.

To improve the bound, the proof has a slightly different flow (App. B.4).

Privacy in the presence of leakage. $\mathsf{ForkDTE}$ 1 and 2 provides CPAL-security, that is, CPA where the adversary gets the leakage of all encryption queries (Def. 20) as DTE does [13]. This follows from the fact that PSV is CPAL [31], and we are using the master key k only in a strongly protected component and FC is a secure forkcipher. For privacy in the presence of leakage, clearly, we cannot use the unbounded leakage model, but other models as in [31,13,10].

5.3 Comparison with Other Leakage-Resistant Schemes

Since the strongly protected components are the slowest, reducing the number of them significantly increases efficiency, especially for short messages. With respect to DTE2, the proposed constructions have faster encryption since only a single call (although obtaining both outputs) to the strongly protected component is made. Moreover, ForkDTE1 is also faster in decryption for the same reason as compared to DTE2, while, ForkDTE2 should be as fast as DTE2 in verification. The comparison of the security of ForkDTE2 and DTE2 is the same as the comparison between HBC2 and ForkMAC. ForkDTE1 is faster in decryption than ForkDTE2, and has the same security properties as DTE2.

There are other schemes with two calls to the strongly protected component: EDT (Encrypt-Digest and Tag) [14], its tweakable versions, TEDT [10] and TEDT2 [30], and Spook a version where encryption and digestion are computed simultaneously with a sponge [7]. They have the same security for CIML2 and black-box integrity as for DTE2. These constructions have to renounce the nonce-misuse security, but have better privacy properties in the presence of leakage.

With respect to ISAP, which follows the Encryption-then-MAC paradigm, using the ISAPMAC to authenticate, we believe the proposed constructions are more efficient in both encryption and decryption and we provide nonce-misuse black-box security. From the previous discussion, the comparison between DTE2 and ISAP can be lifted to ForkDTE 1 and 2.

Moreover, we mention CONCRETE [15] an AE-scheme which provides CIML2 with a single call to a strongly protected primitive, a TBC. CONCRETE is not a nAE, because it is a probabilistic scheme, but it provides CIML2 (where misuse means that the adversary has taken control of the randomness source) if the TBC is leak-free [6]. We believe that CONCRETE is more efficient than our schemes, but its security has only been proved with FC being a leak-free TBC.

Finally, FEDT [18] is based on EDT and they do a clever use of a forkcipher. On the other hand, FEDT, having the same structure of EDT, uses two calls to the leak-free forkcipher, while here in encryption we use only once. Moreover, their security check is similar to the one of LR–MAC.

6 Conclusion

In this paper, we demonstrate that unpredictability with leakage definition can also be extended for the very flexible forkcipher primitive. We have proved that the flexibility that forkciphers is useful to provide authenticity in the presence of leakage; and that such schemes are versatile and cost efficient.

In particular, we have provided three constructions, a MAC - ForkMAC, and two Authenticated Encryption AE - ForkDTE1 and ForkDTE2 schemes, inspired by previous constructions, where the use of a forkcipher has allowed security and efficiency gains. We believe that the flexibility provided by forkciphers can give even nicer constructions for other AE schemes. Rigorously, we detail on the

[6] We strongly suspect that CONCRETE is CIML2 even if the TBC is sU–L2, but this have not been proved yet.

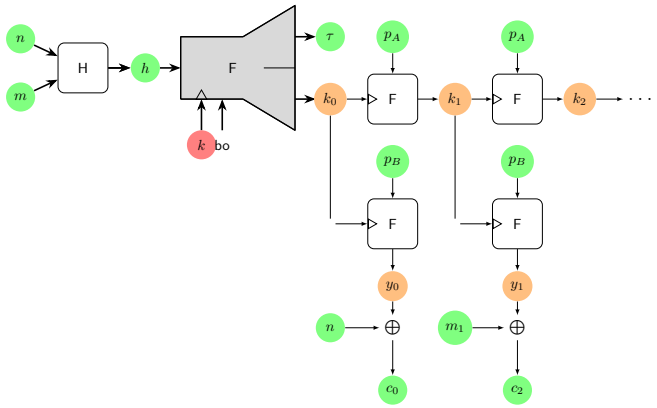

Fig. 4. The encryption algorithm of ForkDTE 1 and 2 - Alg. 2.

security proofs including high-level overview of the different constructions and analyze security characteristics with forkciphers in the presence of *leakage*. The paper also provides a comparison to the state-of-the-art in terms of both security and efficiency. Finally, we conjecture that our MAC provides authenticity in the presence of leakage and faults (at least faults in verification) following [11].

References

1. E. Andreeva, A. S. Bhati, B. Preneel, and D. Vizár. 1, 2, 3, fork: Counter mode variants based on a generalized forkcipher. *IACR Trans. Symmetric Cryptol.*, (3):1–35, 2021.
2. E. Andreeva, B. Cogliati, V. Lallemand, M. Minier, A. Purnal, and A. Roy. Masked iterate-fork-iterate: A new design paradigm for tweakable expanding pseudorandom function. In *ACNS*, 2024.
3. E. Andreeva, V. Lallemand, A. Purnal, R. Reyhanitabar, A. Roy, and D. Vizár. Forkcipher: A new primitive for authenticated encryption of very short messages. In *ASIACRYPT*, 2019.
4. E. Andreeva, R. Reyhanitabar, K. Varici, and D. Vizár. Forking a blockcipher for authenticated encryption of very short messages. *IACR Cryptol. ePrint Arch.*, 2018.
5. E. Andreeva and A. Weninger. A forkcipher-based pseudo-random number generator. In *ACNS*, 2023.
6. M. Bellare, R. Ng, and B. Tackmann. Nonces are noticed: AEAD revisited. In *CRYPTO*, 2019.
7. D. Bellizia, F. Berti, O. Bronchain, G. Cassiers, S. Duval, C. Guo, G. Leander, G. Leurent, I. Levi, C. Momin, O. Pereira, T. Peters, F. Standaert, B. Udvarhelyi, and F. Wiemer. Spook: Sponge-based leakage-resistant authenticated encryption with a masked tweakable block cipher. *IACR Trans. Symmetric Cryptol.*, (S1):295–349, 2020.
8. F. Berti, S. Bhasin, J. Breier, X. Hou, R. Poussier, F. Standaert, and B. Udvarhelyi. A finer-grain analysis of the leakage (non) resilience of OCB. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):461–481, 2022.

9. F. Berti, C. Guo, O. Pereira, T. Peters, and F. Standaert. Strong authenticity with leakage under weak and falsifiable physical assumptions. In *Inscrypt*, 2019.
10. F. Berti, C. Guo, O. Pereira, T. Peters, and F. Standaert. Tedt, a leakage-resist AEAD mode for high physical security applications. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, (1):256–320, 2020.
11. F. Berti, C. Guo, T. Peters, Y. Shen, and F. Standaert. Secure message authentication in the presence of leakage and faults. *IACR Trans. Symmetric Cryptol.*, (1):288–315, 2023.
12. F. Berti, C. Guo, T. Peters, and F. Standaert. Efficient leakage-resilient macs without idealized assumptions. In *ASIACRYPT*, 2021.
13. F. Berti, F. Koeune, O. Pereira, T. Peters, and F. Standaert. Ciphertext integrity with misuse and leakage: Definition and efficient constructions with symmetric primitives. In *AsiaCCS*, 2018.
14. F. Berti, O. Pereira, T. Peters, and F. Standaert. On leakage-resilient authenticated encryption with decryption leakages. *IACR Trans. Symmetric Cryptol.*, 2017.
15. F. Berti, O. Pereira, and F. Standaert. Reducing the cost of authenticity with leakages: a ciml2 -secure ae scheme with one call to a strongly protected tweakable block cipher. In *AFRICACRYPT*, 2019.
16. A. Bhattacharjee, R. Bhaumik, A. Dutta, and E. List. PAE: towards more efficient and bbb-secure AE from a single public permutation. In *ICICS*, 2023.
17. G. Cassiers, B. Grégoire, I. Levi, and F.-X. Standaert. Hardware private circuits: From trivial composition to full verification. *IEEE Transactions on Computers*, 70(10):1677–1690, 2020.
18. N. Datta, A. Dutta, E. List, and S. Mandal. FEDT: Forkcipher-based leakage-resilient beyond-birthday-secure AE. *IACR Communications in Cryptology*, 1(2).
19. C. Dobraunig, M. Eichlseder, S. Mangard, F. Mendel, and T. Unterluggauer. ISAP - towards side-channel secure authenticated encryption. *IACR Trans. Symmetric Cryptol.*, 2017(1):80–105, 2017.
20. C. Dobraunig and B. Mennink. Leakage resilient value comparison with application to message authentication. In *EUROCRYPT*, 2021.
21. Y. Dodis and J. P. Steinberger. Message authentication codes from unpredictable block ciphers. In *CRYPTO*, 2009.
22. Y. Dodis and J. P. Steinberger. Domain extension for macs beyond the birthday barrier. In *EUROCRYPT*, 2011.
23. A. Dutta, J. Guo, and E. List. Forking sums of permutations for optimally secure and highly efficient prfs. *IACR Cryptol. ePrint Arch.*, 2022.
24. D. Goudarzi and M. Rivain. How fast can higher-order masking be in software? In *EUROCRYPT*, 2017.
25. A. Journault and F. Standaert. Very high order masking: Efficient implementation and security evaluation. In *CHES*, 2017.
26. J. Katz and Y. Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.
27. P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, pages 104–113, 1996.
28. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.
29. M. D. Liskov, R. L. Rivest, and D. A. Wagner. Tweakable block ciphers. *J. Cryptol.*, 24(3):588–613, 2011.
30. E. List. TEDT2 - highly secure leakage-resilient tbc-based authenticated encryption. In P. Longa and C. Ràfols, editors, *LATINCRYPT*, 2021.

31. O. Pereira, F. Standaert, and S. Vivek. Leakage-resilient authentication and encryption from symmetric cryptographic primitives. In *CCS*, 2015.
32. J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In *E-smart*, 2001.
33. P. Rogaway. Authenticated-encryption with associated-data. In *CCS*, 2002.
34. P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *FSE*, 2004.
35. P. Rogaway and T. Shrimpton. A provable-security treatment of the key-wrap problem. In *EUROCRYPT*, 2006.
36. D. Salomon and I. Levi. Masksimd-lib: on the performance gap of a generic c optimized assembly and wide vector extensions for masked software with an ascon-p test case. *Journal of Cryptographic Engineering*, 13(3):325–342, 2023.

A Additional definitions

A.1 Pre-image resistant hash functions

There are many possible pre-image resistance definitions for hash functions, see [34]. In this paper we make use of the following definition where a value is picked uniformly at random from the target space of H:

Definition 10. *A hash function* $\mathsf{H} : \mathcal{HK} \times \{0,1\}^* \to \{0,1\}^N$ *is* (t,ϵ)-range oriented pre-image resistant *(*ro–PR*) if* $\forall$ t*-adversaries* A

$$\Pr[m \leftarrow \mathsf{A}(s,y) \text{ s.t. } \mathsf{H}_s(m) = y \mid s \xleftarrow{\$} \mathcal{HK}, y \xleftarrow{\$} \{0,1\}^N] \leq \epsilon.$$

A.2 Tweakable block-ciphers and tweakable forkciphers

Liskov et al. [29] introduced tweakable blockciphers. These are block-ciphers with an additional input, the tweak, that provide more flexibility. We give their syntax and their security definitions.

Definition 11. *A* tweakable block-cipher *(*TBC*) is a function* $\mathsf{E} : \mathcal{K} \times \mathcal{TW} \times \{0,1\}^N \to \{0,1\}^N$*, s.t.* $\forall (k,tw) \in \mathcal{K} \times \mathcal{TW}$ $\mathsf{E}_k(tw,\cdot)$ *is a permutation.*

Definition 12. *A* TBC $\mathsf{E} : \mathcal{K} \times \{0,1\}^N \to \{0,1\}^N$ *is a* (q,t,ϵ)-TPRP (Tweakable Pseudo Random Permutation) *if for any* (q,t)*-adversary* A

$$|\Pr[1 \leftarrow \mathsf{A}^{\mathsf{FC}_k(\cdot)}] - \Pr[1 \leftarrow \mathsf{A}^{\mathsf{f}(\cdot)}]| \leq \epsilon$$

where $k \xleftarrow{\$} \mathcal{K}$*, and* $\mathsf{f} \xleftarrow{\$} \mathcal{TPERM}$*, where* $\mathcal{TPERM}$ *is the set of the tweakable permutations over* $\mathcal{TW} \times \{0,1\}^N$*, that is the set of functions* $\mathsf{f} : \mathcal{TW} \times \{0,1\}^N \to \{0,1\}^N$ *s.t.* $\forall tw \in \mathcal{TW} \mathsf{f}(tw)$ *is a permutation on* $\{0,1\}^N$*.*

When the adversary has oracle access also to the inverse of the TBC we have the strong version of the previous definition.

Definition 13. *A* TBC $\mathsf{E} : \mathcal{K} \times \{0,1\}^N \to \{0,1\}^N$ *is a* (q,t,ϵ)-sTPRP (Strong Tweakable Pseudo Random Permutation) *if for any* $(q,q_I,,t)$*-adversary* A

$$|\Pr[1 \leftarrow \mathsf{A}^{\mathsf{FC}_k(\cdot),\mathsf{FC}_k^{-1,\cdot}(\cdot)}] - \Pr[1 \leftarrow \mathsf{A}^{\mathsf{f}(\cdot),\mathsf{f}^{-1,\cdot}(\cdot)}]| \leq \epsilon$$

where $k \xleftarrow{\$} \mathcal{K}$*, and* $\mathsf{f} \xleftarrow{\$} \mathcal{TPERM}$*, where* $\mathcal{TPERM}$ *is the set of the tweakable permutations over* $\mathcal{TW} \times \{0,1\}^N$*, that is the set of functions* $\mathsf{f} : \mathcal{TW} \times \{0,1\}^N \to \{0,1\}^N$ *s.t.* $\forall tw \in \mathcal{TW} \mathsf{f}(tw)$ *is a permutation on* $\{0,1\}^N$*.*

A.3 Additional Definitions for Authenticated Encryptions

First, we give the security definition for authenticated encryption schemes. This definition provides both privacy and authenticity.

Definition 14. *A* nAE*-scheme* Π *is* (q_E,q_D,t,ϵ)-nAE*-secure if* $\forall (q_E,q_D,t)$*-adversary*

$$|\Pr[\mathsf{A}^{\mathsf{Enc}_k(\cdot,\cdot),\mathsf{Dec}_k(\cdot)} \Rightarrow 1] - \Pr[\mathsf{A}^{\$(\cdot,\cdot),\perp(\cdot)} \Rightarrow 1]| \leq \epsilon,$$

where $\perp$ *is an oracle that outputs always* $\perp$*.* A *is not allowed to ask the second oracle on an input* c *if she has received* c *as the output of the first oracle with input* (n,m)*. Moreover,* A *is not allowed to repeat a nonce in different* $\mathsf{Enc}/\$$*-queries.*

When we remove the latter condition, we have misuse-resistance

Definition 15. *A* nAE*-scheme* Π *is* (q_E, q_D, t, ϵ)-nmAE*-secure (nonce-misuse resistant) if* $\forall(q_E, q_D, t)$*-adversary*

$$|\Pr[\mathsf{A}^{\mathsf{Enc}_k(\cdot,\cdot),\mathsf{Dec}_k(\cdot)} \Rightarrow 1] - \Pr[\mathsf{A}^{\$(\cdot,\cdot),\perp(\cdot)} \Rightarrow 1]| \leq \epsilon,$$

where $\perp$ *is an oracle that outputs always* $\perp$*.* A *is not allowed to ask the second oracle on an input* c *if she has received* c *as the output of the first oracle with input* (n, m)*.*

In some cases, there are data that needs only to be authenticated (for example the header). These are the so called associated data [33].

Definition 16. *A* nonce-based authenticated encryption with associated data (nAAE) *scheme is a triple of algorithms* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *where*

- *The key-generation algorithm* Gen *generates a key from the sets of keys,* $\mathcal{K}$*.*
- *The encryption algorithm* Enc *is a deterministic algorithm which takes as input a key* $k \in \mathcal{K}$*, a nonce* $n \in \mathcal{N}$*, an associated data* $a \in \mathcal{AD}$*, and a message* $m \in \mathcal{M}$*, and outputs a ciphertext* $c \in \mathcal{C}$*. We denote this with* $c \leftarrow \mathsf{Enc}_k(n, a, m)$*.*
- *The decryption algorithm* Dec *is a deterministic algorithm which takes as input a key* $k \in \mathcal{K}$*, a associated data* $a \in \mathcal{AD}$*, and a ciphertext* $c \in \mathcal{C}$*, and outputs a message* $m \in \mathcal{M}$ *or* $\perp$ *("invalid"). We denote this with* $\perp /m = \mathsf{Dec}_k(a, c)$*.*

We require correctness*, that is* $\forall(k, a, m) \in \mathcal{K} \times \mathcal{AD} \times \mathcal{M}$*,* $m = \mathsf{Dec}_k(a, c)$ *if* $c \leftarrow \mathsf{Enc}_k(n, a, m)$ *for any nonce* $n \in \mathcal{N}$*.*

Definition 17. *A* nAAE*-scheme* Π *is* (q_E, q_D, t, ϵ)-nAE*-secure if* $\forall(q_E, q_D, t)$*-adversary*

$$|\Pr[\mathsf{A}^{\mathsf{Enc}_k(\cdot,\cdot,\cdot),\mathsf{Dec}_k(\cdot,\cdot)} \Rightarrow 1] - \Pr[\mathsf{A}^{\$(\cdot,\cdot,\cdot),\perp(\cdot,\cdot)} \Rightarrow 1]| \leq \epsilon,$$

where $\perp$ *is an oracle that outputs always* $\perp$*.* A *is not allowed to ask the second oracle on an input* (a, c) *if she has received* c *as the output of the first oracle with input* (n, a, m)*. Moreover,* A *is not allowed to repeat a nonce in different* Enc/$*-queries.*

A.4 Encryption schemes and privacy definitions

Historically, encryption schemes are assumed to be probabilistic, and the first security [26]. On the other hand, since it is hard to build a probabilistic scheme, many schemes assume that there is an additional input, called the *initialization vector*, IV which is assumed to be randomly picked (there is a fall-back security notion, nonce-security, where, for the security it is enough that the IV has never been repeated [35]). We now formalize the syntax and the security definition:

Definition 18. *A* IV-based encryption (IVE)*-scheme is a triple of algorithms* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *where*

- *The key-generation algorithm* Gen *generates a key from the sets of keys,* $\mathcal{K}$*.*
- *The encryption algorithm* Enc *is a deterministic algorithm which takes as input a key* $k \in \mathcal{K}$*, a* IV $iv \in \mathcal{IV}$*, and a message* $m \in \mathcal{M}$*, and outputs a ciphertext* $c \in \mathcal{C}$*. We denote this with* $c \leftarrow \mathsf{Enc}_k(iv, m)$*.*

– *The decryption algorithm* Dec *is a deterministic algorithm which takes as input a key* $k \in \mathcal{K}$*, an* IV $iv \in \mathcal{IV}$*, and a ciphertext* $c \in \mathcal{C}$*, and outputs a message* $m \in \mathcal{M}$ *or* $\perp$ *("invalid"). We denote this with* $\perp / m = \mathsf{Dec}_k(iv, c)$.

We require correctness*, that is* $\forall (k, iv, m) \in \mathcal{K} \times \mathcal{IV} \times \mathcal{M}$, $m = \mathsf{Dec}_k(iv, \mathsf{Enc}_k(iv, m))$.

Definition 19. *An* IVE*-scheme* Π *is* (q, t, ϵ)-IVE*-secure if* $\forall (q, t)$*-adversary*

$$|\Pr[\mathsf{A}^{\mathsf{Enc}_k^{\$}(\cdot)} \Rightarrow 1] - \Pr[\mathsf{A}^{\$(\cdot)} \Rightarrow 1]| \leq \epsilon,$$

where $\mathsf{Enc}^{\$}(\cdot)$ *is an oracle that on input* m*, picks* $iv \xleftarrow{\$} \mathcal{IV}$*, and output* $(iv, c = \mathsf{Enc}_k(iv, m))$*, while* $\$(\cdot)$ *is an oracle that on input* m *outputs* (iv, c)*, with* $iv \xleftarrow{\$} \mathcal{IV}$*, and* $c \xleftarrow{\$} \{0,1\}^{|\mathsf{Enc}_k(iv,m)|}$.

A.5 Privacy in the presence of leakage

Here we give the security definition for encryption and (authenticated encryption) in the presence of leakage.

Definition 20. *A* nAE*-scheme is* $(q, t, \epsilon)-$Chosen Plaintext Attacks Secure with leakage (CPAL)*-secure if*

$$\Pr[b = b' | \mathsf{A}^{\mathsf{L}, \mathsf{EncL}_k}(c^*) \Rightarrow b', c^* = \mathsf{EncL}_k(n^*, m_b)] \leq \frac{1}{2} + \epsilon,$$

$\forall (q, t)$*-adversary* A*, with* $b \xleftarrow{\$} \{0,1\}$*, where* $\mathsf{A}^{\mathsf{L}, \mathsf{EncL}_k}$ *outputs* (n^*, m_0, m_1) *with* $|m_0| = |m_1|$*. Moreover,* A *is not allowed to repeat a nonce in different* Enc*-queries.*

A.6 Unforgeability for MACs

We give the black-box authenticity definition for MACs.

Definition 21. *A* $\mathsf{MAC} = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ *is* (q_M, q_V, t, ϵ)-strongly existentially unforgeable against chosen message and verification attacks (sUF) *if for all* (q_M, q_V, t)*-adversaries* A*, we have:*

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

where the $\mathsf{FORGE}^{\mathsf{suf\text{-}vcma}}$ *experiment is defined in Tab. 3.*

For simplicity, in the proofs, we consider the verification query induced by the final output of the adversary as the $(q_V + 1)th$ verification query.

A.7 Ciphertext-integrity with Misuse and Leakage

We give the description of the experiment mentioned in the $\mathsf{CIML2}$ security definition (Def. 8).

The $\mathsf{FORGEL2}^{\mathsf{suf\text{-}vcma\text{-}L2}}_{\mathsf{MAC},\mathsf{L}_M,\mathsf{L}_V,\mathsf{A}^{\mathsf{L}}}$ experiment	
Initialization: $k \leftarrow \mathsf{Gen}$ $\mathcal{S} \leftarrow \emptyset$ Finalization: $(m,\tau) \leftarrow \mathsf{A}^{\mathsf{Mac}_k(\cdot),\mathsf{Vrfy}_k(\cdot,\cdot)}$ If $(m,\tau) \in \mathcal{S}$ or $\perp = \mathsf{Vrfy}_k(m,\tau)$ Return 0 Return 1	Oracle $\mathsf{Mac}_k(m)$: $\tau = \mathsf{Mac}_k(m)$ $\mathcal{S} \leftarrow \mathcal{S} \cup \{(m,\tau)\}$ Return τ Oracle $\mathsf{Vrfy}_k(m,\tau)$: Return $\mathsf{Vrfy}_k(m,\tau)$

Table 3. The $\mathsf{FORGE}^{\mathsf{suf\text{-}vcma}}_{\mathsf{MAC},\mathsf{A}}$ experiment (vcma stands for Verification and Chosen Message Attacks).

The $\mathsf{CIML2}_{\Pi,\mathsf{L}_E,\mathsf{L}_D,\mathsf{A}^{\mathsf{L}}}$ experiment	
Initialization: $k \leftarrow \mathsf{Gen}$ $\mathcal{S} \leftarrow \emptyset$ Finalization: $c \leftarrow \mathsf{A}^{\mathsf{L},\mathsf{EncL}_k(\cdot,\cdot),\mathsf{DecL}_k(\cdot)}$ If $c \in \mathcal{S}$ or $\perp = \mathsf{Dec}_k(c)$ Return 0 Return 1	Oracle $\mathsf{EncL}_k(n,m)$: $c = \mathsf{Enc}_k(n,m)$ $\mathcal{S} \leftarrow \mathcal{S} \cup \{(c)\}$ Return $(c, \mathsf{L}_E(n,m;k))$ Oracle $\mathsf{DecL}_k(c)$: Return $(\mathsf{Dec}_k(c), \mathsf{L}_D(c;k))$

Table 4. The $\mathsf{CIML}_{\Pi,\mathsf{L}_E,\mathsf{L}_D,\mathsf{A}}$ experiment.

B Proofs

B.1 Proof of the sUF–L2-security of ForkMAC

Proof. We use a sequence of games Game 0, ... , Game 4. We denote with E_i the event that the output of Game i is 1, that is, that the adversary wins.

Game 0. This is the $\mathsf{sUF\text{–}L2}$ game where the adversary A is playing against $\mathsf{ForkMAC}$.

Game 1. Similar to Game 0, except that we abort if there is a collision for the hash function.

Transition between Game 0 and 1. Since Game 0 and Game 1 are the same except if a hash collision is produced, we only need to bound the probability that such a collision is found. To do this, we build a t_2-adversary B which works as follows:

At the start of the game B obtains the key of the hash function, s, which she forwards to A. She picks a random key k and forwards s to A. Moreover, she has a list $\mathcal{S}$ which is empty.

When A perform a tag-generation query on input m, B simply computes $h = \mathsf{H}_s(m)$, $\tau = \mathsf{FC}_k(h, 0)$ and collects the leakage $\mathsf{leak} = \mathsf{L}_{\mathsf{FC}}(h, 0; k)$. She returns τ and the leakage $(h, \mathsf{L}_{\mathsf{FC}}(h, 0; k))$ to A and she adds (m, h) to $\mathcal{S}$. This takes time $t_{\mathsf{H}} + t_{\mathsf{FC}}$.

When A does a verification query on input (m, τ), B simply computes $h = \mathsf{H}_s(m)$, $v = \mathsf{FC}_k(h, 1)$, $\tilde{v} = \mathsf{FC}_k^{-1}(\tau, 0, \mathsf{o})$ and collects the leakage $\mathsf{L}_{\mathsf{FC}}(h, 1; k), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k)$. She returns $\top$ if $v = \tilde{v}$, otherwise $\perp$ to A. Moreover, she returns to A the leakage $(h, v, \tilde{v}, \mathsf{L}_{\mathsf{FC}}(h, 1; k), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k))$. Finally, she adds (m, h) to $\mathcal{S}$. This takes time $t_{\mathsf{H}} + 2t_{\mathsf{FC}}$.

When A outputs her forgery (m, τ), B simply computes $h = \mathsf{H}_s(m)$, she adds (m, h) to $\mathcal{S}$, and she looks into $\mathcal{S}$ to find a collision. If this is the case, she outputs it, otherwise $(0^N, 1^N)$. This takes time t_{H}.

The adversary A can do the modeling queries (which are q_L) by himself, thus, we do not have to explain how B treats them.

Thus, in total B runs in time bounded by $t+(q_M+q_V+1)t_{\mathsf{H}}+(q_M+2q_V)t_{\mathsf{FC}} = t_2$.

Bounding $|\Pr[E_0] - \Pr[E_1]|$. Since B is t_2-adversary, H is a $(t_2, \epsilon_{\mathsf{CR}})$-collision resistant hash function, and Game 0 and Game 1 are the same except if B finds a collision, then

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

Game 2. Let Game 2 be Game 1, where we abort if there exist two verification queries, the i^{th} and the j^{th} s.t. $j < i$ and $v^i = \tilde{v}^j$.

Games $1^0, ..., 1^{q_V}$. Let Game 1^i be Game 1 where we abort if in one of the first i verification queries there exist two verification queries, the l^{th} and the j^{th} s.t. $j < l$ and $v^l = \tilde{v}^j$. Note that Game 1^0 is Game 1, while Game 1^{q_V} is Game 2.

Transition between Game 1^i and 1^{i+1}. Since Game i and Game $i + 1$ are the same except if in the ith verification query $v^i = \tilde{v}^j$ for $j < i$, we only need to bound the latter event.

To do this, we build a t_1-adversary C^i which works as follows: At the start of the game C^i obtains the key of the hash function, s, which she forwards to A. Moreover, C^i has a list $\mathcal{V}$ which is empty.

When A does a modelling tag-generation query on input $(m; k')$, C^i simply computes $h = \mathsf{H}_s(m)$, computes $\tau = \mathsf{FC}_{k'}(h, 0)$ and collects the leakage $\mathsf{leak} = \mathsf{L}_{\mathsf{FC}}(h, 0; k')$. She returns τ and the leakage $(h, \mathsf{L}_{\mathsf{FC}}(h, 0; k'))$ to A and she adds (m, h) to $\mathcal{S}$. This takes time t_{H} and one modeling query to FC.

When A does a modelling verification query on input (m, τ), C^i simply computes $h = \mathsf{H}_s(m)$, $v = \mathsf{FC}_{k'}(h, 1)$, $\tilde{v} = \mathsf{FC}^{-1}_{k'}(\tau, 0, \mathsf{o})$ and collects the leakage $(\mathsf{L}_{\mathsf{FC}}(h, 1; k'), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k'))$. She returns $\top$ if $v = \tilde{v}$, otherwise $\perp$ to A. Moreover, she returns to A the leakage $(h, v, \tilde{v}, \mathsf{L}_{\mathsf{FC}}(h, 1; k'), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k'))$. This takes time t_{H} and two modeling queries to FC.

When A does a tag-generation query on input m, C^i simply computes $h = \mathsf{H}_s(m)$, calls her oracle FCL_k on input $(h, 0)$, obtaining τ and the leakage $\mathsf{L}_{\mathsf{FC}}(h, 0; k)$. She returns τ and the leakage $(h, \mathsf{L}_{\mathsf{FC}}(h, 0; k))$ to A. This takes time t_{H} and one oracle query to FC.

When A does one of the first $i - 1$ verification queries on input (m, τ), C^i simply computes $h = \mathsf{H}_s(m)$, calls her oracle FC on input $(h, 1)$ obtaining v and the leakage $\mathsf{L}_{\mathsf{FC}}(h, 1; k)$. She queries her oracle FC^{-1} on input $(\tau, 0, \mathsf{o})$, obtaining $\tilde{v}$ and the leakage $\mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k)$. She returns $\top$ if $v = \tilde{v}$, otherwise $\perp$ to A. Moreover, she returns to A the leakage $(h, v, \tilde{v}, \mathsf{L}_{\mathsf{FC}}(h, 1; k), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k))$. Finally, she adds $\tilde{v}$ to $\mathcal{S}$. This takes time t_{H} and one query to FC and one to FC^{-1}.

When A outputs the ith verification query on input (m, τ), C^i simply computes $h = \mathsf{H}_s(m)$, picks an element x randomly from $\mathcal{V}$ and she outputs $(h, 1, x)$ as her prediction. This takes time t_{H}.

Thus, in total C^i runs in time bounded by $t + (q_L + q_M + q_v + 1)t_{\mathsf{H}} = t_1$, does at most $2q_L$ modelling queries and at most $q_M + q_V$ queries to FC and q_V to FC^{-1}.

Bounding $|\Pr[E_{1^i}] - \Pr[E_{1^{i+1}}]|$ **.** Since C^i is $(2q_L, q_M + q_V, q_V, t_1)$-adversary, FC is a $(2q_L, q_M + q_V, q_V, t_1, \epsilon_{\mathsf{sU\text{-}L2}})$-unpredictable forkcipher, and Game 1^i and Game 1^{i+1} are the same except if in the ith verification query $v^i = \tilde{v}^j$ with $j < i$, then

$$|\Pr[E^i_1] - \Pr[E_{1^{i+1}}]| = \Pr[\text{correct guess}]\Pr[\mathsf{C}^i \text{ wins }] \leq (i-1)\epsilon_{\mathsf{sU\text{-}L2}},$$

because we have randomly picked x from the set of possible $\tilde{v}$s, thus, if $i = 1$, $|\mathcal{V}| = 0$, so C^1 can never win, while, if $i > 1$, we have guessed correctly with probability at least $1/|\mathcal{V}| = (i-1)^{-1}$.

Bounding $|\Pr[E_1] - \Pr[E_2]|$**.** Summing all the previous probabilities, we obtain

$$|\Pr[E_{1^0}] - \Pr[E_{1^{q_V}}]| = \sum_{i=1}^{q_V}(i-1)\epsilon_{\mathsf{sU\text{-}L2}} = \sum_{i=1}^{q_V-1} i\epsilon_{\mathsf{sU\text{-}L2}} = \frac{q_V(q_V-1)}{2}\epsilon_{\mathsf{sU\text{-}L2}}.$$

Game 3. Let Game 3 be Game 2, where we abort if there exist two verification queries, the i^{th} and the j^{th} s.t. $j < i$ and $\tilde{v}^i = v^j$.

Games $2^0, ..., 2^{q_V}$**.** Let Game 2^0 be Game 2 where we abort if in one of the first i verification queries there exist two verification queries, the l^{th} and the j^{th} s.t. $j < l$ and $\tilde{v}^l = v^j$. Note that Game 2^0 is Game 2, while Game 2^{q_V} is Game 3.

Transition between Game 2^i and 2^{i+1}. Since Game 2^i and Game 2^{i+1} are the same except if in the ith verification query $\tilde{v}^i = v^j$ for $j < i$, we only need to bound the latter event.

To do this, we build a t_1-adversary D^i which works as follows: At the start of the game D^i obtains the key of the hash function, s, which she forwards to A. Moreover, D^i has a list $\mathcal{V}$ which is empty.

When A does a modelling tag-generation query on input $(m; k')$, D^i simply computes $h = \mathsf{H}_s(m)$, computes $\tau = \mathsf{FC}_{k'}(h, 0)$ and collects the leakage $\mathsf{leak} = \mathsf{L}_{\mathsf{FC}}(h, 0; k')$. She returns τ and the leakage $(h, \mathsf{L}_{\mathsf{FC}}(h, 0; k'))$ to A and she adds (m, h) to $\mathcal{S}$. This takes time t_{H} and one modeling query to FC.

When A does a modelling verification query on input (m, τ), D^i simply computes $h = \mathsf{H}_s(m)$, $v = \mathsf{FC}_{k'}(h, 1)$, $\tilde{v} = \mathsf{FC}^{-1}_{k'}(\tau, 0, \mathsf{o})$ and collects the leakage $(\mathsf{L}_{\mathsf{FC}}(h, 1; k'), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k'))$. She returns $\top$ if $v = \tilde{v}$, otherwise $\bot$ to A. Moreover, she returns to A the leakage $(h, v, \tilde{v}, \mathsf{L}_{\mathsf{FC}}(h, 1; k'), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k'))$. This takes time t_{H} and two modeling queries to FC.

When A does a tag-generation query on input m, D^i simply computes $h = \mathsf{H}_s(m)$, calls her oracle FCL_k on input $(h, 0)$, obtaining τ and the leakage $\mathsf{L}_{\mathsf{FC}}(h, 0; k)$. She returns τ and the leakage $(h, \mathsf{L}_{\mathsf{FC}}(h, 0; k))$ to A. This takes time t_{H} and one oracle query to FC.

When A does one of the first $i - 1$ verification queries on input (m, τ), D^i simply computes $h = \mathsf{H}_s(m)$, calls her oracle FC on input $(h, 1)$ obtaining v and the leakage $\mathsf{L}_{\mathsf{FC}}(h, 1; k)$. She queries her oracle FC^{-1} on input $(\tau, 0, \mathsf{o})$, obtaining $\tilde{v}$ and the leakage $\mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k)$. She returns $\top$ if $v = \tilde{v}$, otherwise $\bot$ to A. Moreover, she returns to A the leakage $(h, v, \tilde{v}, \mathsf{L}_{\mathsf{FC}}(h, 1; k), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k))$. Finally, she adds v to $\mathcal{S}$. This takes time t_{H} and one query to FC and one to FC^{-1}.

When A outputs the ith verification query on input (m, τ), D^i simply computes $h = \mathsf{H}_s(m)$, picks an element x randomly from $\mathcal{V}$ and she outputs (τ, o, x) as her prediction. This takes time t_{H}.

Thus, in total D^i runs in time bounded by $t + (q_L + q_M + q_v + 1)t_{\mathsf{H}} = t_1$, does at most $2q_L$ modelling queries and at most $q_M + q_V$ queries to FC and q_V to FC^{-1}.

Bounding $|\Pr[E_{2^i}] - \Pr[E_{2^{i+1}}]|$. Since D^i is $(2q_L, q_M + q_V, q_V, t_1)$-adversary, FC is a $(2q_L, q_M + q_V, q_V, t_1, \epsilon_{\mathsf{sU\text{-}L2}})$-unpredictable forkcipher, and Game 2^i and Game 2^{i+1} are the same except if in the ith verification query $\tilde{v}^i = v^j$ with $j < i$, then

$$|\Pr[E_2^i] - \Pr[E_{2^{i+1}}]| = \Pr[\text{correct guess}]\Pr[\mathsf{D}^i \text{ wins }] \leq (i-1)\epsilon_{\mathsf{sU\text{-}L2}},$$

because we have randomly picked x from the set of possible $\tilde{v}$s, thus, if $i = 1$, $|\mathcal{V}| = 0$, so D^1 can never win, while, if $i > 1$, we have guessed correctly with probability at least $1/|\mathcal{V}| = (i-1)^{-1}$.

Bounding $|\Pr[E_2] - \Pr[E_3]|$. Summing all the previous probabilities, we obtain

$$|\Pr[E_{2^0}] - \Pr[E_{2^{q_V}}]| = \sum_{i=1}^{q_V}(i-1)\epsilon_{\mathsf{sU\text{-}L2}} = \sum_{i=1}^{q_V - 1} i\epsilon_{\mathsf{sU\text{-}L2}} = \frac{q_V(q_V - 1)}{2}\epsilon_{\mathsf{sU\text{-}L2}}.$$

Games 4. Let Game 4 be Game 3 where we abort there is one fresh and valid verification query.

Games $3^1, ..., 3^{q_V+1}$. Let Game 3^i be Game 3 where we abort if one of the first i verification queries is fresh and valid. (We remind that we consider the verification query induced by A output as the $q_V + 1^{\text{th}}$ verification query. Note that Game 3^0 is Game 3, while Game 3^{q_V+1} is Game 3.

Transition between Game 3^i and 3^{i+1}. Since Game 3^i and Game 3^{i+1} are the same except if the ith verification query is fresh and valid, we only need to bound the probability that the input of the ith verification query, (m^i, τ^i), is fresh and $\mathsf{Vrfy}_k(m^i, \tau^i) = \top$.

To do this, we build a t_1-adversary EE^i which works as follows: At the start of the game EE^i obtains the key of the hash function, s, which she forwards to A.

When A does a modelling tag-generation query on input $(m; k')$, EE^i simply computes $h = \mathsf{H}_s(m)$, computes $\tau = \mathsf{FC}_{k'}(h, 0)$ and collects the leakage $\mathsf{leak} = \mathsf{L}_{\mathsf{FC}}(h, 0; k')$. She returns τ and the leakage $(h, \mathsf{L}_{\mathsf{FC}}(h, 0; k'))$ to A and she adds (m, h) to $\mathcal{S}$. This takes time t_{H} and one modelling query to FC.

When A does a modelling verification query on input (m, τ), EE^i simply computes $h = \mathsf{H}_s(m)$, $v = \mathsf{FC}_{k'}(h, 1)$, $\tilde{v} = \mathsf{FC}^{-1}_{k'}(\tau, 0, \mathsf{o})$ and collects the leakage $(\mathsf{L}_{\mathsf{FC}}(h, 1; k'), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k'))$. She returns $\top$ if $v = \tilde{v}$, otherwise $\perp$ to A. Moreover, she returns to A the leakage $(h, v, \tilde{v}, \mathsf{L}_{\mathsf{FC}}(h, 1; k'), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k'))$. Finally, she adds (m, h) to $\mathcal{S}$. This takes time t_{H} and two modelling querlies to FC.

When A does a tag-generation query on input m, EE^i simply computes $h = \mathsf{H}_s(m)$, calls her oracle FCL_k on input $(h, 0)$, obtaining τ and the leakage $\mathsf{L}_{\mathsf{FC}}(h, 0; k)$. She returns τ and the leakage $(h, \mathsf{L}_{\mathsf{FC}}(h, 0; k))$ to A. This takes time t_{H} and one oracle query to FC.

When A does one of the first $i - 1$ verification queries on input (m, τ), EE^i simply computes $h = \mathsf{H}_s(m)$, calls her oracle FC on input $(h, 1)$ obtaining v and the leakage $\mathsf{L}_{\mathsf{FC}}(h, 1; k)$. She queries her oracle FC^{-1} on input $(\tau, 0, \mathsf{o})$, obtaining $\tilde{v}$ and the leakage $\mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k)$. She returns $\top$ if $v = \tilde{v}$, otherwise $\perp$ to A. Moreover, she returns to A the leakage $(h, v, \tilde{v}, \mathsf{L}_{\mathsf{FC}}(h, 1; k), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k))$. Finally, she adds (m, h) to $\mathcal{S}$. This takes time t_{H} and one query to FC and one to FC^{-1}.

When A outputs the ith verification query on input (m, τ), EE^i simply computes $h = \mathsf{H}_s(m)$, and she outputs $(h, 0, \tau)$ as her prediction. This takes time t_{H}.

Thus, in total EE^i runs in time bounded by $t + (q_L + q_M + q_v + 1)t_{\mathsf{H}} = t_1$, does at most $2q_L$ modelling queries and at most $q_M + q_V$ queries to FC and q_V to FC^{-1}.

Bounding $|\Pr[E_{3^i}] - \Pr[E_{3^{i+1}}]|$ and $|\Pr[E_3] - \Pr[E_4]|$. Since EE^i is $(2q_L, q_M + q_V, q_V, t_1)$-adversary, FC is a $(2q_L, q_M + q_V, q_V, t_1, \epsilon_{\mathsf{sU\text{-}L2}})$-unpredictable forkcipher, and Game 3^i and Game 3^{i+1} are the same except if the ith verification query is the first fresh and valid verification query, then

$$|\Pr[E_3^i] - \Pr[E_{3^{i+1}}]| = \Pr[\mathsf{B} \text{ wins }] \leq \epsilon_{\mathsf{sU\text{-}L2}}.$$

$$\text{So, } |\Pr[E_3] - \Pr[E_4]| \leq \sum_{i=0}^{q_V+1} |\Pr[E_{3^i}] - \Pr[E_{4^{i+1}}]| \leq (q_V + 1)\epsilon_{\mathsf{sU\text{-}L2}}$$

Concluding the proof. We can conclude the proof, since $\Pr[E_4] = 0$, since none of the q_V verification query and the verification query induced by the forgery output of A can be fresh and valid. Thus,

$$\Pr[E_0] \leq \Pr[E_4] + \sum_{i=0}^{3} |\Pr[E_i] - \Pr[E_{i+1}]| \leq \epsilon_{\mathsf{CR}} + \frac{2q_V(q_V-1)}{2}\epsilon_{\mathsf{sU\text{-}L2}} + (q_V+1)\epsilon_{\mathsf{sU\text{-}L2}} = \epsilon.$$

B.2 Unforgeability (Black-Box) of ForkMAC

The security definition is given by Def. 21.

Theorem 4. *Let* FC *be a* $(q_M + q_V + 1, 0, t_1, \epsilon_{\mathsf{PRFP}})$*-pseudorandom forkcipher. Let* H *be a* $(t_2, \epsilon_{\mathsf{CR}})$*-collision resistant. Then* ForkMAC *is* (q_M, q_V, t, ϵ)-sUF*-secure with*

$$\epsilon \leq \epsilon_{\mathsf{PRFP}} + \epsilon_{\mathsf{CR}} + \frac{(q_M + q_V + 1)^2 + 2q_V + 2}{2^{N+1}},$$

with $t_1 = t + (q_M + q_V + 1)t_{\mathsf{H}}$*,* $t_2 = t + (q_M + q_V + 1)t_{\mathsf{H}} + (q_M + q_V)t_{\mathsf{f}}$*, with* t_{H} *the time needed to execute once the hash function* H*,* t_{f} *to randomly sample a random permutation.*

Proof. We use a sequence of games Game 0, ... , Game 5. We denote with E_i the event that the output of Game i is 1, that is, that the adversary wins.

Game 0. This is the sUF game where the adversary A is playing against ForkMAC.

Game 1. This is the sUF game where the adversary A is playing against $\mathsf{ForkMAC}'$, which is ForkMAC, where the verification is done recomputing the correct tag and checking it.

Bounding $|\Pr[E_0] - \Pr[E_1]|$**.** It is clear that ForkMAC and $\mathsf{ForkMAC}'$ are functionally equivalent, that is $\mathsf{Mac}(m)_k = \mathsf{Mac}'_k(m)$, and $\mathsf{Vrfy}_k(m,\tau) = \mathsf{Vrfy}'_k(m,\tau)$, since FC implements two permutations.

$$|\Pr[E_0] - \Pr[E_1]| = 0.$$

Game 2. It is Game 1, where we replace FC with two random permutations.

Transition between Game 1 and 2. Game 0 and Game 1 are the same except for how τ and $\tilde{h}$ are computed in Mac and Vrfy queries respectively. In Game 0, they are computed with FC_k and FC_k^{-1}, respectively, while in Game 1 with the ideal counterpart $\tilde{F}, \tilde{F}^{-1}$. Thus, we build a $(q_M, q_V + 1, t_1)$-adversary B.

B has to distinguish if she is interacting with two oracle implemented either with $\mathsf{FC}_k(\cdot,\cdot), \mathsf{FC}_k(\cdot,\cdot,\cdot)$ or with $\tilde{\mathsf{FC}}(\cdot,\cdot), \tilde{\mathsf{FC}}(\cdot,\cdot,\cdot)$. At the start of the game, B receives a key s for the hash function, which she forwards to A. Moreover, she has a list $\mathcal{S}$ which is empty.

When A does a tag-generation query on input m, B simply computes $h = \mathsf{H}_s(m)$ and queries her oracle on input $(h, 0)$ obtaining τ. B answers A τ and she adds (m, τ) to $\mathcal{S}$. This requires an oracle query to the first oracle (implemented with either $\mathsf{FC}_k(\cdot,\cdot)$ or $\tilde{F}(\cdot,\cdot,\cdot)$) and time t_{H}.

When A does a verification query on input (m, τ), B simply computes $h = \mathsf{H}_s(m)$ and queries her oracle on input $(h, 0)$ obtaining $\tilde{\tau}$. B answers A $\top$ if $\tau = \tilde{tau}$; otherwise $\bot$. This requires an oracle query to the first oracle (implemented with either $\mathsf{FC}_k(\cdot,\cdot)$ or $\tilde{F}(\cdot,\cdot,\cdot)$) and time t_{H}.

When A outputs its forgery (m, τ), B simply computes $h = \mathsf{H}_s(m)$ and queries her oracle on input $(h, 0)$ obtaining $\tilde{\tau}$. B outputs 1 if $\tau = t\tilde{a}u$ and $(m, \tau) \notin \mathcal{S}$; otherwise $\perp$. This requires an oracle query to the first oracle (implemented with either $\mathsf{FC}_k(\cdot,\cdot)$ or $\tilde{F}(\cdot,\cdot,\cdot)$) and time t_{H}.

Thus, in total B does $q_M + q_V + 1$ queries to the first oracle (implemented with either $\mathsf{FC}_k(\cdot,\cdot)$ or $\tilde{F}(\cdot,\cdot,\cdot)$), no queries to the second oracle, and time at most $t + (q_M + q_V + 1)t_{\mathsf{H}} \leq t_1$.

Bounding $|\Pr[E_1] - \Pr[E_2]|$**.** Since B is $(q_M + q_V + 1, 0, t_1)$-adversary, and FC is a $(q_M + q_V + 1, 0, t_1, \epsilon_{\mathsf{PRFP}})$-pseudorandom forkcipher, and B simulates correctly Game 1 for A if her oracles are implemented with $\mathsf{FC}_k(\cdot,\cdot)$ and $\mathsf{FC}_k^{-1}(\cdot,\cdot,\cdot)$, otherwise Game 2, thus

$$|\Pr[E_1] - \Pr[E_2]| = |\Pr[1 \leftarrow \mathsf{B}^{\mathsf{FC}_k(\cdot,\cdot),\mathsf{FC}_k^{-1}(\cdot,\cdot,\cdot)}] - \Pr[1 \leftarrow \mathsf{B}^{tildeF(\cdot,\cdot),\tilde{F}^{-1}(\cdot,\cdot,\cdot)}]| \leq \epsilon_{\mathsf{PRFP}}.$$

Observe, that since we are only using $\mathsf{FC}_k(\cdot, 0)$-queries, we can consider that we are using a random permutation.

Game 3. It is Game 2, where we assume that no collision for the hash function is found.

Transition between Game 2 and 3. Game 2 and Game 1 are the same except if the following event happens: a collision for the hash function. Thus, we have only to bound the probability that the previous event happens. For this, we build a t_2-adversary against C H which aims to find a collision.

C proceeds at follows: At the start of the game, C receives a key s for the hash function, which she forwards to A. Moreover, she picks a random permutation f, which she lazy samples and she has a list $\mathcal{S}$ which is empty.

When A does a tag-generation query on input m, C simply computes $h = \mathsf{H}_s(m)$, and lazy samples $\tau = \mathsf{f}(h)$. Then, she answers τ to A, and she adds (m, h) to $\mathcal{S}$. This takes time $t_{\mathsf{H}} + t_{\mathsf{f}}$.

When A does a verification query on input (m, τ), C simply computes $h = \mathsf{H}_s(m)$ and lazy samples $\tilde{\tau} = \mathsf{f}(h)$. C answers A $\top$ if $\tau = t\tilde{a}u$; otherwise $\perp$. Moreover, she adds (m, h) to $\mathcal{S}$. This needs time $t_{\mathsf{H}} + t_{\mathsf{f}}$.

When A outputs its forgery (m, τ), C simply computes $h = \mathsf{H}_s(m)$. Then, she looks into $\mathcal{S}$ to see if she can find a collision. If it is the case, she outputs it, otherwise $(0, 1)$. This requires time t_{H}.

Thus, in total C runs in time at most $t + (q_M + q_V + 1)t_{\mathsf{H}} + (q_M + q_V)t_{\mathsf{f}} \leq t_2$.

Bounding $|\Pr[E_2] - \Pr[E_3]|$**.** Game 2 and 3 are the same except if a collision for the hash function has been found. Since C wins if a collision for the has function is found in Game 2, C is t_2-adversary, H is a $(t_2, \epsilon_{\mathsf{CR}})$-collision resistant hash function,

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

Game 4. Let Game 4 be Game 3 where we replace f with a random function.

Transition between Game 3 and 4. Using the well-known lemma to switch from a PRP to a PRF, since the only difference between Game 3 and 4 is the use of a PRP or a PRF, and we use f is $q_M + q_V$ times,

$$|\Pr[E_3] - \Pr[E_4]| \leq \frac{(q_M + q_V + 1)^2}{2}^{N+1}.$$

Game 5. Let Game 5 be Game 4 where we assume that there is no fresh and valid decryption query.

Bounding $|\Pr[E_4] - \Pr[E_5]|$**.** Since the probability that the ith is valid, if it is fresh is bounded by 2^{-N}. Thus,

$$|\Pr[E_4] - \Pr[E_5]| =\leq (q_V + 1)2^{-N}.$$

Concluding the proof. We can conclude the proof, since $\Pr[E_5] = 0$, since none of the q_V verification query and the verification query induced by the forgery output of A can be fresh and valid. Thus,

$$\Pr[E_0] \leq \Pr[E_5] + \sum_{i=0}^{4} |\Pr[E_i] - \Pr[E_{i+1}]| \leq \epsilon_{\mathsf{PRFP}} + \epsilon_{\mathsf{CR}} + \frac{(q_M + q_V + 1)^2 + 2q_V + 2}{2^{N+1}} = \epsilon.$$

B.3 Proof of the CIML2 Security of ForkDTE1

Proof. We use a sequence of games Game 0, ... , Game 4. We denote with E_i the event that the output of Game i is 1, that is, that the adversary wins.

Game 0. This is the CIML2 game where the adversary A is playing against ForkDTE1.

Game 1. It is Game 0, where we replace FC_k with its ideal counterpart.

Transition between Game 0 and 1. Since Game 0 and Game 1 are the same except for the use of FC, we need to build the probability an adversary distinguish the use of FC to its ideal counterpart. To do this, we build a $(q_{\mathsf{FC}}, q_{\mathsf{FC}^{-1}}, t_1)$-adversary B which has access to two oracles which are either implemented with $\mathsf{FC}_k, \mathsf{FC}_k^{-1}$ or their ideal counterparts. B works as follows: At the start of the game B obtains the key of the hash function, s, which she forwards to A. Moreover, B has a list $\mathcal{S}$ which is empty.

When A does an encryption query on input (n, m), B simply computes $h = \mathsf{H}_s(n\|m)$, and calls her oracle on input (h, b) obtaining (τ, k_0). From k_0, B computes $y_0 = \mathsf{E}_{k_0}(p_B)$, and $c_0 = y_0 \oplus n$. Then, she parses m in n-bit blocks, $m_1, \ldots, m_\ell$. After that, for all $i = 1, \ldots, \ell$, B computes $k_i = \mathsf{E}_{k_{i-1}}(p_A)$, $y_i = \mathsf{E}_{k_i}(p_B)$, and $c_i = \pi_{|m_i|}(y_i) \oplus m_i$. Finally, she returns A $c = (\tau, C)$ and the leakage k_0, with $C = (c_0, \ldots, c_\ell)$ and she adds c to $\mathcal{S}$. This takes one oracle query to FC_k and time $t_{\mathsf{H}} + (2\ell + 1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L + 1)t_{\mathsf{E}}$, since $\ell \leq L$.

When A does a decryption query on input c, she parses it in n-bits blocks, $\tau, c_0, c_1, \ldots, m_\ell$. Then, B simply calls her inverse oracle on input $(\tau, 0, \mathsf{b})$, obtaining $(\tilde{h}, k_0)$. From k_0, B computes $y_0 = \mathsf{E}_{k_0}(p_B)$, and $n = y_0 \oplus c_0$. After that, for all $i = 1, \ldots, \ell$, B computes $k_i = \mathsf{E}_{k_{i-1}}(p_A)$, $y_i = \mathsf{E}_{k_i}(p_B)$, and $m_i = \pi_{|m_i|}(y_i) \oplus c_i$. Finally, she computes $h = \mathsf{H}_s(n\|m)$ and checks if $h \stackrel{?}{=} \tilde{h}$. If it is the case, B returns A $m = (m_1, \ldots, m_\ell)$, and the leakage $(\tilde{h}, k_0)$; otherwise, $\perp$ and the leakage $(\tilde{h}, k_0)$. This takes one oracle query to FC_k^{-1} and time $t_{\mathsf{H}} + (2\ell + 1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L + 1)t_{\mathsf{E}}$, since $\ell \leq L$.

The adversary A can do the modeling queries (which are q_L) by himself, thus, we do not have to explain how B treats them.

When A outputs its forgery c, she proceeds as for a normal decryption query except that she does not return anything to A. Instead, if at the end of the verification $h = \tilde{h}$ and $c \notin \mathcal{S}$, B outputs 1; otherwise 0.

Thus, in total B does q_E queries to FC, $q_D + 1$ to FC^{-1} and runs in time bounded by $t + (q_E + q_D + 1)[t_{\mathsf{H}} + (2L + 1)t_{\mathsf{E}}] = t_1$.

Bounding $|\Pr[E_0] - \Pr[E_1]|$**.** If the oracles B has access to are implemented by $(\mathsf{FC}_k, \mathsf{FC}_k^{-1})$, B is correctly simulating Game 0 for A; otherwise, Game 1. Since B is (q_E, q_D+1, t_1)-adversary, and FC is a $(q_E, q_D+1, t_1, \epsilon_{\mathsf{CR}})$-$\mathsf{PRFP}$ secure forkcipher, then

$$|\Pr[E_0] - \Pr[E_1]| = |\Pr[1 \leftarrow \mathsf{B}^{\mathsf{FC},\mathsf{FC}^{-1}}] - \Pr[1 \leftarrow \mathsf{B}^{\mathsf{f},\mathsf{f}^{-1}}]| \leq \epsilon_{\mathsf{PRFP}}.$$

Game 2

It is Game 1, where we abort if there is a collision for the hash function.

Transition between Game 1 and 2. Since Game 1 and Game 2 are the same except if a hash collision is found, we build a t_2-adversary C based on A to find a collision for the hash function. C works as follows: At the start of the game C obtains the key of the hash function, s, which she forwards to A. Moreover, C has a list $\mathcal{S}$ which is empty, and she picks two random permutation $\mathsf{f}_0, \mathsf{f}_1$, which she lazy samples.

When A does an encryption query on input (n, m), C simply computes $h = \mathsf{H}_s(n\|m)$ and she adds $(n\|m, h)$ to $\mathcal{S}$, and computes $\tau = \mathsf{f}_0(h)$, and $k_0 = \mathsf{f}_1(h)$. From k_0, C computes $y_0 = \mathsf{E}_{k_0}(p_B)$, and $c_0 = y_0 \oplus n$. Then, she parses m in n-bit blocks, $m_1, \ldots, m_\ell$. After that, for all $i = 1, \ldots, \ell$, C computes $k_i = \mathsf{E}_{k_{i-1}}(p_A)$, $y_i = \mathsf{E}_{k_i}(p_B)$, and $c_i = \pi_{|m_i|}(y_i) \oplus m_i$. Finally, she returns A $c = (\tau, C)$ and the leakage k_0, with $C = (c_0, \ldots, c_\ell)$. This takes time $t_{\mathsf{H}} + 2t_{\mathsf{f}} + (2\ell+1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}$, since $\ell \leq L$.

When A does a decryption query on input c, she parses it in n-bits blocks, $\tau, c_0, c_1, \ldots, m_\ell$. Then, C computes $\tilde{h} = \mathsf{f}_0(\tau)$, $k_0 = \mathsf{f}_1(\tilde{h})$. From k_0, C computes $y_0 = \mathsf{E}_{k_0}(p_B)$, and $n = y_0 \oplus c_0$. After that, for all $i = 1, \ldots, \ell$, C computes $k_i = \mathsf{E}_{k_{i-1}}(p_A)$, $y_i = \mathsf{E}_{k_i}(p_B)$, and $m_i = \pi_{|m_i|}(y_i) \oplus c_i$. Finally, she computes $h = \mathsf{H}_s(n\|m)$, she adds $(n\|m, h)$ to $\mathcal{S}$, and checks if $h \stackrel{?}{=} \tilde{h}$. If it is the case, C returns A $m = (m_1, \ldots, m_\ell)$, and the leakage $(\tilde{h}, k_0)$; otherwise, $\perp$ and the leakage $(\tilde{h}, k_0)$. This takes time $t_{\mathsf{H}} + 2t_{\mathsf{f}} + (2\ell+1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}$, since $\ell \leq L$.

The adversary A can do the modeling queries (which are q_L) by himself, thus, we do not have to explain how C treats them.

When A outputs its forgery c, she proceeds as for a normal decryption query except that she does not return anything to A. After this, C looks into $\mathcal{S}$ to see whether she can find a collision in $\mathcal{S}$. If she finds it, she outputs it; otherwise $(0, 1)$. Thus, in total C runs in time bounded by $t + (q_E + q_D + 1)[2t_{\mathsf{f}} + t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}] = t_2$.

Bounding $|\Pr[E_1] - \Pr[E_2]|$**.** Game 1 and Game 2 are the same except if the event that there is a collision for the hash function happens. Since C wins only if that event happens, C is a t_2-adversary and H is $(t_2, \epsilon_{\mathsf{CR}})$-collision resistant hash function,

$$|\Pr[E_1] - \Pr[E_2]| = \Pr[\text{ There is a collision for } \mathsf{H}] \leq \Pr[\mathsf{C} \text{ wins}] \leq \epsilon_{\mathsf{CR}}.$$

Game 3

It is Game 3, where we abort if there exists a not valid decryption query which generates $\tilde{h}$ and an encryption or decryption query s.t. the hash computed h is equal to that $\tilde{h}$.

Games $2^0, ..., 2^{q_D}$. Let Game 2^i be Game 2 where we abort if for one of the first i decryption queries there exist j s.t. there exists an h s.t. $h = \tilde{h}^j$. Note that Game 2^0 is Game 2, while Game 2^{q_D} is Game 3.

Transition between Game 2^j and 2^{j+1}. Since Game 2^j and Game 2^{i+1} are the same except there is a hash query s.t. $h = \tilde{h}^j$, we only need to bound the latter event.

To do this, we build a t_3-adversary D^j which works as follows: At the start of the game D^i obtains the key of the hash function, s, which she forwards to A, and a random target x. Moreover, D^j has two list $\mathcal{S}$, and $\mathcal{F}$ which are empty, and she picks two random permutation $\mathsf{f}_0, \mathsf{f}_1$, which she lazy samples.

When A does an encryption query on input (n, m), D^j simply computes $h = \mathsf{H}_s(n\|m)$ and she adds $(n\|m, h)$ to $\mathcal{S}$, computes $\tau = \mathsf{f}_0(h)$ checking before if there is an entry $(h, \tau) \in \mathcal{F}$, if it is the case, it answers τ, and $k_0 = \mathsf{f}_1(h)$, and she adds (h, τ) to $\mathcal{F}$. From k_0, D^j computes $y_0 = \mathsf{E}_{k_0}(p_B)$, and $c_0 = y_0 \oplus n$. Then, she parses m in n-bit blocks, $m_1, \ldots, m_\ell$. After that, for all $= 1, \ldots, \ell$, C computes $k_i = \mathsf{E}_{k_{i-1}}(p_A)$, $y_i = \mathsf{E}_{k_i}(p_B)$, and $c_i = \pi_{|m_i|}(y_i) \oplus m_i$. Finally, she returns A $c = (\tau, C)$ and the leakage k_0, with $C = (c_0, \ldots, c_\ell)$. This takes time $t_{\mathsf{H}} + 2t_{\mathsf{f}} + (2\ell + 1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L + 1)t_{\mathsf{E}}$, since $\ell \leq L$.

When A does a decryption query on input c, she parses it in n-bits blocks, $\tau, c_0, c_1, \ldots, m_\ell$. Then, if this is not the jth verification query D^j computes $\tilde{h} = \mathsf{f}_0(\tau)$ (checking before if there is an entry in $(\tilde{h}, \tau) \in \mathcal{F}$, if it is the case she sets $\tilde{h}$ coherently the with the entry in $\mathcal{F}$) Instead, if this is the jth verification query, D^j checks if there is an entry in $(\tilde{h}, \tau) \in \mathcal{F}$, if it is the case she sets $\tilde{h}$ coherently the with the entry in $\mathcal{F}$, otherwise she sets $\tilde{h} := x$. Then, she adds $(\tilde{h}, \tau)$ to $\mathcal{F}$. Moreover, she computes $k_0 = \mathsf{f}_1(\tilde{h})$, and adds . From k_0, D^j computes $y_0 = \mathsf{E}_{k_0}(p_B)$, and $n = y_0 \oplus c_0$. After that, for all $i = 1, \ldots, \ell$, C computes $k_i = \mathsf{E}_{k_{i-1}}(p_A)$, $y_i = \mathsf{E}_{k_i}(p_B)$, and $m_i = \pi_{|m_i|}(y_i) \oplus c_i$. Finally, she computes $h = \mathsf{H}_s(n\|m)$, she adds $(n\|m, h)$ to $\mathcal{S}$, and checks if $h \stackrel{?}{=} \tilde{h}$. If it is the case, D^j returns A $m = (m_1, \ldots, m_\ell)$, and the leakage $(\tilde{h}, k_0)$; otherwise, $\perp$ and the leakage $(\tilde{h}, k_0)$. This takes time $t_{\mathsf{H}} + 2t_{\mathsf{f}} + (2\ell + 1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L + 1)t_{\mathsf{E}}$, since $\ell \leq L$.

The adversary A can do the modeling queries (which are q_L) by himself, thus, we do not have to explain how D^j treats them.

When A outputs its forgery c, she proceeds as for a normal decryption query except that she does not return anything to A. After this, D^j looks into $\mathcal{S}$ to see whether she can find an entry $(\cdot, x)$ in $\mathcal{S}$. If she finds it, she outputs it; otherwise 0.

Thus, in total D^j runs in time bounded by $t + (q_E + q_D + 1)[2t_{\mathsf{f}} + t_{\mathsf{H}} + (2L + 1)t_{\mathsf{E}}] = t_2$. .

Bounding $|\Pr[E_{2^j}] - \Pr[E_{2^{j+1}}]|$. We observe that if we have not set in the jth decryption query $\tilde{h} := x$, it means that the pre-image for x has already been found in a previous query. Since D^i is t_2-adversary, H is a $(t_2, \epsilon_{\mathsf{ro\text{-}PR}})$-range-oriented-pre-image resistant hash function, and Game 2^j and Game 2^{j+1} are the same except if a preimage for x is found, then

$$|\Pr[E_2^j] - \Pr[E_{2^{j+1}}]| = \Pr[\text{find a pre-image}] = \Pr[\mathsf{D}^j \text{ wins }] \leq \epsilon_{\mathsf{ro\text{-}PR}}.$$

Bounding $|\Pr[E_2] - \Pr[E_3]|$**.** Summing all the previous probabilities, we obtain

$$|\Pr[E_{2^0}] - \Pr[E_{2^{q_D}}]| = \sum_{i=1}^{q_D} \epsilon_{\mathsf{ro\text{-}PR}} = q_D \epsilon_{\mathsf{ro\text{-}PR}}.$$

Game 4. Let Game 4 be Game 3, where we abort if there exist a fresh and valid verification query.

Games $3^1, ..., 3^{q_V+1}$**.** Let Game 3^i be Game 3 where we abort if one of the first i decryption queries is fresh and valid. (We remind that we consider the verification query induced by A output as the $q_V + 1^{\text{th}}$ verification query. Note that Game 3^0 is Game 4, while Game 3^{q_V+1} is Game 5.

Transition between Game 3^i **and** 3^{i+1}**.** Since Game 3^i and Game 3^{i+1} are the same except if the ith decryption query is fresh and valid, we only need to bound the probability of the latter event. By hypothesis, the only possibility for an adversary to win is to ask a decryption query on input $c = (\tau, C)$, with a fresh τ (otherwise, $\tilde{h} = \mathsf{f}_0^{-1}(\tau)$ had already been set, thus, the adversary can win by finding a pre-image for $\tilde{h}$, but we have already excluded this). Since f_0 is random permutation, the probability that $h = \tilde{h}$ with $\tilde{h}$ random is $1/2^N$. Thus, $|\Pr[E_{4^i}] - \Pr[E_{4^{i+1}}]| \leq 2^{-N}$. Thus, $|\Pr[E_4] - \Pr[E_5]| \leq (q_D + 1)2^{-N}$

Concluding the proof. We can conclude the proof, since $\Pr[E_5] = 0$, since none of the q_V decryption query and the decryptionn query induced by the forgery output of A can be fresh and valid. Thus,

$$\Pr[E_0] \leq \Pr[E_4] + \sum_{i=0}^{3} |\Pr[E_i] - \Pr[E_{i+1}]| \leq \epsilon_{\mathsf{PRFP}} + \epsilon_{\mathsf{CR}} + q_D \epsilon_{\mathsf{ro\text{-}PR}} + + (q_D+1)2^{-N} = \epsilon.$$

B.4 Proof of the CIML2 Security of ForkDTE2

Proof. We use a sequence of games Game 0, ... , Game 5. We denote with E_i the event that the output of Game i is 1, that is, that the adversary wins.

Game 0. This is the CIML2 game where the adversary A is playing against ForkDTE2.

Game 1. It is Game 0, where we abort if there is a collision for the hash function.

Transition between Game 0 and 1. Since Game 0 and Game 1 are the same except if a hash collision is produced, we only need to bound the probability that such a collision is found. To do this, we build a t_2-adversary B which works as follows:

At the start of the game B obtains the key of the hash function, s, which she forwards to A. She picks a random key k and forwards s to A. Moreover, she has a list $\mathcal{S}$ which is empty.

When A does an encryption query on input m, B simply computes $h = \mathsf{H}_s(m)$, $(\tau, k_0) = \mathsf{FC}_k(h, \mathsf{b})$ and collects the leakage $\mathsf{leak} = \mathsf{L}_{\mathsf{FC}}(h, \mathsf{b}; k)$. After having parsed m in N-bit blocks, $m = (m_1, \ldots, m_\ell)$, from k_0, B computes $y_0 = \mathsf{E}_{k_0}(p_B)$, $c_0 = y_0 \oplus n$, and, for every $i = 1, \ldots \ell$, B computes $k_i = \mathsf{FC}_{k_{i-1}}(p_A)$, $y_i = \mathsf{FC}_{k_i}(p_B)$, $c_i = \pi_{|m_i|}(y_i) \oplus m_i$. She sets $C = (c_0, \ldots, c_\ell)$. She returns $c = (\tau, C)$ and the leakage $(h, k_0, \mathsf{L}_{\mathsf{FC}}(h, \mathsf{b}; k))$ to A and she adds c to $\mathcal{S}$. This takes time $t_{\mathsf{H}} + t_{\mathsf{FC}} + (2\ell + 1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + t_{\mathsf{FC}} + (2L + 1)t_{\mathsf{E}}$.

When A does a decryption query on input c, B simply parses c in (τ, C). $|\tau| = N$, and she computes $k_0 = \mathsf{FC}_k^{-1}(\tau, 0, \mathsf{o})$ and collects the leakage $\mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k)$. After having parsed C in N-bit blocks, $C = (c_1, \ldots, c_\ell)$, from k_0, B computes $y_0 = \mathsf{E}_{k_0}(p_B)$, $n = y_0 \oplus c_0$, and, for every $i = 1, \ldots \ell$, B computes $k_i = \mathsf{FC}_{k_{i-1}}(p_A)$, $y_i = \mathsf{FC}_{k_i}(p_B)$, $m_i = \pi_{|c_i|}(y_i) \oplus c_i$. She sets $m = (m_1, \ldots, m_\ell)$. Then, B computes $h = \mathsf{H}_s(n \| m)$, $\tilde{k}_0 = \mathsf{FC}_k(h, 1)$, and collects the leakage $\mathsf{L}_{\mathsf{FC}}(h, 1; k)$. She returns m if $k_0 = \tilde{k}_0$, otherwise $\perp$ to A. Moreover, she returns to A the leakage $(h, k_0, \tilde{k}_0, \mathsf{L}_{\mathsf{FC}}(h, 1; k), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k))$. Finally, she adds $(n \| m, h)$ to $\mathcal{S}$. This takes time $t_{\mathsf{H}} + 2t_{\mathsf{FC}} + (2\ell + 1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + 2t_{\mathsf{FC}} + (2L + 1)t_{\mathsf{E}}$.

When A outputs her forgery c, B simply proceeds as for a normal decryption query, with the exception that she does not answer A anything. Instead, she looks into $\mathcal{S}$ to find a collision. If this is the case, she outputs it, otherwise $(0^N, 1^N)$. This takes time $t_{\mathsf{H}} + 2t_{\mathsf{FC}} + (2\ell + 1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + 2t_{\mathsf{FC}} + (2L + 1)t_{\mathsf{E}}$.

The adversary A can do the modeling queries (which are q_L) by himself, thus, we do not have to explain how B treats them.

Thus, in total B runs in time bounded by $t + (q_E + q_D + 1)t_{\mathsf{H}} + (q_E + 2q_D + 2)t_{\mathsf{FC}} + (q_E + q_D + 1)(2L + 1)t_{\mathsf{E}} = t_2$.

Bounding $|\Pr[E_0] - \Pr[E_1]|$**.** Since B is t_2-adversary, H is a $(t_2, \epsilon_{\mathsf{CR}})$-collision resistant hash function, and Game 0 and Game 1 are the same except if B finds a collision, then

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

Game 2. Let Game 2 be Game 1, where we abort if there exist two decryption queries, the i^{th} and the j^{th} s.t. $j < i$ and $\tilde{k}_0^i = k_0^j$.

Games $1^0, ..., 1^{q_D}$**.** Let Game 1^i be Game 1 where we abort if in one of the first i decryption queries there exist two verification queries, the l^{th} and the j^{th} s.t. $j < l$ and $\tilde{k_0}^l = k_0^j$. Note that Game 1^0 is Game 1, while Game 1^{q_D} is Game 2.

Transition between Game 1^i **and** 1^{i+1}**.** Since Game i and Game $i + 1$ are the same except if in the ith decryption query $\tilde{k}_0^i = k_0^j$ for $j < i$, we only need to bound the latter event.

To do this, we build a t_1-adversary C^i which works as follows: At the start of the game C^i obtains the key of the hash function, s, which she forwards to A. Moreover, C^i has a list $\mathcal{V}$ which is empty.

When A does a modelling encryption query on input $(n, m; k')$, C^i simply computes $h = \mathsf{H}_s(n, m)$, computes $(\tau, k_0) = \mathsf{FC}_{k'}(h, \mathsf{b})$ and collects the leakage $\mathsf{leak} = \mathsf{L}_{\mathsf{FC}}(h, \mathsf{b}; k')$. After having parsed m in N-bit blocks, $m = (m_1, \ldots, m_\ell)$, from k_0, C^i computes $y_0 = \mathsf{E}_{k_0}(p_B)$, $c_0 = y_0 \oplus n$, and, for every $i = 1, \ldots \ell$, C^i computes $k_i = \mathsf{FC}_{k_{i-1}}(p_A)$, $y_i = \mathsf{FC}_{k_i}(p_B)$, $c_i = \pi_{|m_i|}(y_i) \oplus m_i$. She sets $C = (c_0, \ldots, c_\ell)$. She returns $c = (\tau, C)$ and the leakage $(h, \mathsf{L}_{\mathsf{FC}}(h, \mathsf{b}; k'))$ to A and she adds (m, h) to $\mathcal{S}$. This takes time $t_{\mathsf{H}} + (2\ell + 1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L + 1)t_{\mathsf{E}}$ and one modeling query to FC.

When A does a modelling decryption query on input c, C^i simply parses $c = (\tau, C)$. Then she computes,$k_0 = \mathsf{FC}_{k'}^{-1}(\tau, 0, \mathsf{o})$ and collects the leakage $\mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k')$. After having parsed C in N-bit blocks, $C = (c_1, \ldots, c_\ell)$, from k_0, C^i computes $y_0 = \mathsf{E}_{k_0}(p_B)$, $n = y_0 \oplus c_0$, and, for every $i = 1, \ldots \ell$, C^i computes $k_i = \mathsf{FC}_{k_{i-1}}(p_A)$, $y_i = \mathsf{FC}_{k_i}(p_B)$, $m_i = \pi_{|c_i|}(y_i) \oplus c_i$. She sets $m = (m_1, \ldots, m_\ell)$.

After that, she computes $h = \mathsf{H}_s(n\|m)$, $\tilde{k_0} = \mathsf{FC}_{k'}(h,1)$, and collects the leakage $\mathsf{L}_{\mathsf{FC}}(h,1;k')$. She returns m if $k_0 = \tilde{k_0}$, otherwise $\perp$ to A. Moreover, she returns to A the leakage $(h, k_0, \tilde{k}_0, \mathsf{L}_{\mathsf{FC}}(h,1;k'), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau,0,\mathsf{o};k'))$. This takes time $t_{\mathsf{H}} + (2\ell+1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}$ and two modeling queries to FC.

When A does an encryption query on input m, C^i simply computes $h = \mathsf{H}_s(m)$, calls her oracle FCL_k on input (h,b), obtaining (τ, k_0) and the leakage $\mathsf{L}_{\mathsf{FC}}(h,\mathsf{b};k)$. After having parsed m in N-bit blocks, $m = (m_1,\ldots,m_\ell)$, from k_0, C^i computes $y_0 = \mathsf{E}_{k_0}(p_B)$, $c_0 = y_0 \oplus n$, and, for every $i = 1,\ldots\ell$, B computes $k_i = \mathsf{FC}_{k_{i-1}}(p_A)$, $y_i = \mathsf{FC}_{k_i}(p_B)$, $c_i = \pi_{|m_i|}(y_i) \oplus m_i$. She sets $C = (c_0,\ldots,c_\ell)$. She returns $c = (\tau, C)$ and the leakage $(h, \mathsf{L}_{\mathsf{FC}}(h,\mathsf{b};k))$ to A. This takes time $t_{\mathsf{H}} + (2\ell+1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}$ and one oracle query to FC.

When A does one of the first $i-1$ decryption queries on input c, C^i simply parses $c = (\tau, C)$, with $|\tau| = N$. Then, she queries her oracle FC^{-1} on input $(\tau,0,\mathsf{o})$, obtaining k_0 and the leakage $\mathsf{L}_{\mathsf{FC}^{-1}}(\tau,0,\mathsf{o};k)$. After having parsed C in N-bit blocks, $C = (c_1,\ldots,c_\ell)$, from k_0, C^i computes $y_0 = \mathsf{E}_{k_0}(p_B)$, $n = y_0 \oplus c_0$, and, for every $i = 1,\ldots\ell$, C^i computes $k_i = \mathsf{FC}_{k_{i-1}}(p_A)$, $y_i = \mathsf{FC}_{k_i}(p_B)$, $m_i = \pi_{|c_i|}(y_i) \oplus c_i$. She sets $m = (m_1,\ldots,m_\ell)$. After that, C^i simply computes $h = \mathsf{H}_s(n\|m)$, and she calls her oracle FC on input $(h,1)$ obtaining $\tilde{k}_0$ and the leakage $\mathsf{L}_{\mathsf{FC}}(h,1;k)$. She returns m if $k = \tilde{k}_0$, otherwise $\perp$ to A. Moreover, she returns to A the leakage $(h, k_0, \tilde{k}_0, \mathsf{L}_{\mathsf{FC}}(h,1;k), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau,0,\mathsf{o};k))$. Finally, she adds k_0 to $\mathcal{S}$. This takes time t_{H} and one query to FC and one to FC^{-1}.

When A outputs the ith decryption query on input (m,τ), C^i proceeds as for the previous decryption queries until having retrieved n and m. Then, she computes $h = \mathsf{H}_s(m)$, picks an element x randomly from $\mathcal{V}$ and she outputs $(h,1,x)$ as her prediction. This takes time $t_{\mathsf{H}} + (2\ell+1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}$.

Thus, in total C^i runs in time bounded by $t + (q_L + q_E + q_D)[t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}] \leq t_1$, does at most $2q_L$ modelling queries and at most $q_E + q_D$ queries to FC and q_D to FC^{-1}.

Bounding $|\Pr[E_{1^i}] - \Pr[E_{1^{i+1}}]|$ **.** Since C^i is $(2q_L, q_E+q_D, q_D, t_1)$-adversary, FC is a $(2q_L, q_E+q_D+1, q_D+1, t_1, \epsilon_{\mathsf{sU\text{-}L2}})$-unpredictable forkcipher, and Game 1^i and Game 1^{i+1} are the same except if in the ith verification query $\tilde{k}_0^i = k_0^j$ with $j < i$, then

$$|\Pr[E_1^i] - \Pr[E_{1^{i+1}}]| = \Pr[\text{correct guess}]\Pr[\mathsf{C}^i \text{ wins }] \leq (i-1)\epsilon_{\mathsf{sU\text{-}L2}},$$

because we have randomly picked x from the set of possible k_0s, thus, if $i = 1$, $|\mathcal{V}| = 0$, so C^1 can never win, while, if $i > 1$, we have guessed correctly with probability at least $1/|\mathcal{V}| = (i-1)^{-1}$.

Bounding $|\Pr[E_1] - \Pr[E_2]|$**.** Summing all the previous probabilities, we obtain

$$|\Pr[E_{1^0}] - \Pr[E_{1^{q_D}}]| = \sum_{i=1}^{q_V}(i-1)\epsilon_{\mathsf{sU\text{-}L2}} = \sum_{i=1}^{q_D-1} i\epsilon_{\mathsf{sU\text{-}L2}} = \frac{q_D(q_D-1)}{2}\epsilon_{\mathsf{sU\text{-}L2}}.$$

Game 3. Let Game 3 be Game 2, where we abort if there exist two decryption queries, the i^{th} and the j^{th} s.t. $j < i$ and $\tilde{k}_0^i = k_0^i$.

Games $2^0, ..., 2^{q_D}$**.** Let Game 2^0 be Game 2 where we abort if in one of the first i verification queries there exist two verification queries, the l^{th} and the j^{th} s.t. $j < l$ and $\tilde{v}^l = v^j$. Note that Game 2^0 is Game 2, while Game 2^{q_D} is Game 3.

Transition between Game 2^i and 2^{i+1}. Since Game 2^i and Game 2^{i+1} are the same except if in the ith verification query $\tilde{v}^i = v^j$ for $j < i$, we only need to bound the latter event.

To do this, we build a t_1-adversary D^i which works as follows: At the start of the game D^i obtains the key of the hash function, s, which she forwards to A. Moreover, D^i has a list $\mathcal{V}$ which is empty.

When A does a modelling encryption query on input $(n, m; k')$, D^i simply computes $h = \mathsf{H}_s(n, m)$, computes $(\tau, k_0) = \mathsf{FC}_{k'}(h, \mathsf{b})$ and collects the leakage $\mathsf{leak} = \mathsf{L}_{\mathsf{FC}}(h, \mathsf{b}; k')$. After having parsed m in N-bit blocks, $m = (m_1, \ldots, m_\ell)$, from k_0, D^i computes $y_0 = \mathsf{E}_{k_0}(p_B)$, $c_0 = y_0 \oplus n$, and, for every $i = 1, \ldots \ell$, D^i computes $k_i = \mathsf{FC}_{k_{i-1}}(p_A)$, $y_i = \mathsf{FC}_{k_i}(p_B)$, $c_i = \pi_{|m_i|}(y_i) \oplus m_i$. She sets $C = (c_0, \ldots, c_\ell)$. She returns $c = (\tau, C)$ and the leakage $(h, \mathsf{L}_{\mathsf{FC}}(h, \mathsf{b}; k'))$ to A. This takes time $t_{\mathsf{H}} + (2\ell + 1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L + 1)t_{\mathsf{E}}$ and one modeling query to FC.

When A does a modelling decryption query on input c, D^i simply parses $c = (\tau, C)$. Then she computes,$k_0 = \mathsf{FC}^{-1}_{k'}(\tau, 0, \mathsf{o})$ and collects the leakage $\mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k')$. After having parsed C in N-bit blocks, $C = (c_1, \ldots, c_\ell)$, from k_0, C^i computes $y_0 = \mathsf{E}_{k_0}(p_B)$, $n = y_0 \oplus c_0$, and, for every $i = 1, \ldots \ell$, D^i computes $k_i = \mathsf{FC}_{k_{i-1}}(p_A)$, $y_i = \mathsf{FC}_{k_i}(p_B)$, $m_i = \pi_{|c_i|}(y_i) \oplus c_i$. She sets $m = (m_1, \ldots, m_\ell)$. After that, she computes $h = \mathsf{H}_s(n \| m)$, $\tilde{k}_0 = \mathsf{FC}_{k'}(h, 1)$, and collects the leakage $\mathsf{L}_{\mathsf{FC}}(h, 1; k')$. She returns m if $k_0 = \tilde{k}_0$, otherwise $\perp$ to A. Moreover, she returns to A the leakage $(h, k_0, \tilde{k}_0, \mathsf{L}_{\mathsf{FC}}(h, 1; k'), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k'))$. This takes time $t_{\mathsf{H}} + (2\ell + 1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L + 1)t_{\mathsf{E}}$ and two modeling queries to FC.

When A does an encryption query on input (n, m), D^i simply computes $h = \mathsf{H}_s(m)$, calls her oracle FCL_k on input (h, b), obtaining (τ, k_0) and the leakage $\mathsf{L}_{\mathsf{FC}}(h, \mathsf{b}; k)$. After having parsed m in N-bit blocks, $m = (m_1, \ldots, m_\ell)$, from k_0, D^i computes $y_0 = \mathsf{E}_{k_0}(p_B)$, $c_0 = y_0 \oplus n$, and, for every $i = 1, \ldots \ell$, B computes $k_i = \mathsf{FC}_{k_{i-1}}(p_A)$, $y_i = \mathsf{FC}_{k_i}(p_B)$, $c_i = \pi_{|m_i|}(y_i) \oplus m_i$. She sets $C = (c_0, \ldots, c_\ell)$. She returns $c = (\tau, C)$ and the leakage $(h, \mathsf{L}_{\mathsf{FC}}(h, \mathsf{b}; k))$ to A. This takes time $t_{\mathsf{H}} + (2\ell + 1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L + 1)t_{\mathsf{E}}$ and one oracle query to FC.

When A does one of the first $i - 1$ verification queries on input (m, τ), D^i simply parses $c = (\tau, C)$, with $|\tau| = N$. Then, she queries her oracle FC^{-1} on input $(\tau, 0, \mathsf{o})$, obtaining k_0 and the leakage $\mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k)$. After having parsed C in N-bit blocks, $C = (c_1, \ldots, c_\ell)$, from k_0, D^i computes $y_0 = \mathsf{E}_{k_0}(p_B)$, $n = y_0 \oplus c_0$, and, for every $i = 1, \ldots \ell$, D^i computes $k_i = \mathsf{FC}_{k_{i-1}}(p_A)$, $y_i = \mathsf{FC}_{k_i}(p_B)$, $m_i = \pi_{|c_i|}(y_i) \oplus c_i$. She sets $m = (m_1, \ldots, m_\ell)$. After that, D^i simply computes $h = \mathsf{H}_s(n \| m)$, and she calls her oracle FC on input $(h, 1)$ obtaining $\tilde{k}_0$ and the leakage $\mathsf{L}_{\mathsf{FC}}(h, 1; k)$. She returns m if $k = \tilde{k}_0$, otherwise $\perp$ to A. Moreover, she returns to A the leakage $(h, k_0, \tilde{k}_0, \mathsf{L}_{\mathsf{FC}}(h, 1; k), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k))$. Finally, she adds $\tilde{k}_0$ to $\mathcal{S}$. This takes time $t_{\mathsf{H}} + (2\ell + 1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L + 1)t_{\mathsf{E}}$ and one query to FC and one to FC^{-1}.

When A outputs the ith decryption query on input (m, τ), D^i simply picks an element x randomly from $\mathcal{S}$ and she outputs (τ, o, x) as her prediction. This takes no time. Thus, in total D^i runs in time bounded by $t + (q_L + q_E + q_D)[t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}] \leq t_1$, does at most $2q_L$ modelling queries and at most $q_E + q_D$ queries to FC and q_D to FC^{-1}.

Bounding $|\Pr[E_{2^i}] - \Pr[E_{2^{i+1}}]|$ **.** Since D^i is $(2q_L, q_E + q_D, q_D, t_1)$-adversary, FC is a $(2q_L, q_E + q_D, q_D, t_1, \epsilon_{\mathsf{sU\text{-}L2}})$-unpredictable forkcipher, and Game 2^i and Game 2^{i+1} are the same except if in the ith decryption query $k_0^i = \tilde{k}_0^j$ with $j < i$, then

$$|\Pr[E_2^i] - \Pr[E_{2^{i+1}}]| = \Pr[\text{correct guess}]\Pr[\mathsf{D}^i \text{ wins }] \leq (i-1)\epsilon_{\mathsf{sU\text{-}L2}},$$

because we have randomly picked x from the set of possible $\tilde{k}_0$s, thus, if $i = 1$, $|\mathcal{V}| = 0$, so D^1 can never win, while, if $i > 1$, we have guessed correctly with probability at least $1/|\mathcal{V}| = (i-1)^{-1}$.

Bounding $|\Pr[E_2] - \Pr[E_3]|$**.** Summing all the previous probabilities, we obtain

$$|\Pr[E_{2^0}] - \Pr[E_{2^{q_D}}]| = \sum_{i=1}^{q_D}(i-1)\epsilon_{\mathsf{sU\text{-}L2}} = \sum_{i=1}^{q_D-1} i\epsilon_{\mathsf{sU\text{-}L2}} = \frac{q_D(q_D-1)}{2}\epsilon_{\mathsf{sU\text{-}L2}}.$$

Games 4. Let Game 4 be Game 3 where we abort there is one fresh and valid decryption query.

Games $3^1, ..., 3^{q_D+1}$**.** Let Game 3^i be Game 3 where we abort if one of the first i decryption queries is fresh and valid. (We remind that we consider the verification query induced by A output as the $q_V + 1^{\text{th}}$ decryption query. Note that Game 3^0 is Game 3, while Game 3^{q_D+1} is Game 3.

Transition between Game 3^i **and** 3^{i+1}**.** Since Game 3^i and Game 3^{i+1} are the same except if the ith decryption query is fresh and valid, we only need to bound the probability that the input of the ith decryption query, c, is fresh and $\mathsf{Dec}_k(c) \neq \perp$.

To do this, we build a t_1-adversary EE^i which works as follows: At the start of the game EE^i obtains the key of the hash function, s, which she forwards to A.

When A does a modelling encryption query on input $(n, m; k')$, EE^i simply computes $h = \mathsf{H}_s(n, m)$, computes $(\tau, k_0) = \mathsf{FC}_{k'}(h, \mathsf{b})$ and collects the leakage $\mathsf{leak} = \mathsf{L}_{\mathsf{FC}}(h, \mathsf{b}; k')$. After having parsed m in N-bit blocks, $m = (m_1, \ldots, m_\ell)$, from k_0, EE^i computes $y_0 = \mathsf{E}_{k_0}(p_B)$, $c_0 = y_0 \oplus n$, and, for every $i = 1, \ldots \ell$, EE^i computes $k_i = \mathsf{FC}_{k_{i-1}}(p_A)$, $y_i = \mathsf{FC}_{k_i}(p_B)$, $c_i = \pi_{|m_i|}(y_i) \oplus m_i$. She sets $C = (c_0, \ldots, c_\ell)$. She returns $c = (\tau, C)$ and the leakage $(h, \mathsf{L}_{\mathsf{FC}}(h, \mathsf{b}; k'))$ to A. This takes time $t_{\mathsf{H}} + (2\ell+1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}$ and one modeling query to FC.

When A does a modelling decryption query on input c, EE^i simply parses $c = (\tau, C)$. Then she computes,$k_0 = \mathsf{FC}^{-1}_{k'}(\tau, 0, \mathsf{o})$ and collects the leakage $\mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k')$. After having parsed C in N-bit blocks, $C = (c_1, \ldots, c_\ell)$, from k_0, C^i computes $y_0 = \mathsf{E}_{k_0}(p_B)$, $n = y_0 \oplus c_0$, and, for every $i = 1, \ldots \ell$, D^i computes $k_i = \mathsf{FC}_{k_{i-1}}(p_A)$, $y_i = \mathsf{FC}_{k_i}(p_B)$, $m_i = \pi_{|c_i|}(y_i) \oplus c_i$. She sets $m = (m_1, \ldots, m_\ell)$. After that, she computes $h = \mathsf{H}_s(n \| m)$, $\tilde{k}_0 = \mathsf{FC}_{k'}(h, 1)$, and collects the leakage $\mathsf{L}_{\mathsf{FC}}(h, 1; k')$. She returns m if $k_0 = \tilde{k}_0$, otherwise $\perp$ to A. Moreover, she returns to A the leakage $(h, k_0, \tilde{k}_0, \mathsf{L}_{\mathsf{FC}}(h, 1; k'), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k'))$. This takes time $t_{\mathsf{H}} + (2\ell+1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}$ and two modeling queries to FC.

When A does an encryption query on input (n, m), EE^i simply computes $h = \mathsf{H}_s(m)$, calls her oracle FCL_k on input (h, b), obtaining (τ, k_0) and the leakage $\mathsf{L}_{\mathsf{FC}}(h, \mathsf{b}; k)$. After having parsed m in N-bit blocks, $m = (m_1, \ldots, m_\ell)$, from k_0, EE^i computes $y_0 = \mathsf{E}_{k_0}(p_B)$, $c_0 = y_0 \oplus n$, and, for every $i = 1, \ldots \ell$,

EE^i computes $k_i = \mathsf{FC}_{k_{i-1}}(p_A)$, $y_i = \mathsf{FC}_{k_i}(p_B)$, $c_i = \pi_{|m_i|}(y_i) \oplus m_i$. She sets $C = (c_0, \ldots, c_\ell)$. She returns $c = (\tau, C)$ and the leakage $(h, \mathsf{L}_{\mathsf{FC}}(h, \mathsf{b}; k))$ to A. This takes time $t_{\mathsf{H}} + (2\ell+1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}$ and one oracle query to FC.

When A does one of the first $i-1$ verification queries on input (m, τ), EE^i simply parses $c = (\tau, C)$, with $|\tau| = N$. Then, she queries her oracle FC^{-1} on input $(\tau, 0, \mathsf{o})$, obtaining k_0 and the leakage $\mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k)$. After having parsed C in N-bit blocks, $C = (c_1, \ldots, c_\ell)$, from k_0, EE^i computes $y_0 = \mathsf{E}_{k_0}(p_B)$, $n = y_0 \oplus c_0$, and, for every $i = 1, \ldots \ell$, B computes $k_i = \mathsf{FC}_{k_{i-1}}(p_A)$, $y_i = \mathsf{FC}_{k_i}(p_B)$, $m_i = \pi_{|c_i|}(y_i) \oplus c_i$. She sets $m = (m_1, \ldots, m_\ell)$. After that, EE^i simply computes $h = \mathsf{H}_s(n \| m)$, and she calls her oracle FC on input $(h, 1)$ obtaining $\tilde{k}_0$ and the leakage $\mathsf{L}_{\mathsf{FC}}(h, 1; k)$. She returns m if $k = \tilde{k}_0$, otherwise $\perp$ to A. Moreover, she returns to A the leakage $(h, k_0, \tilde{k}_0, \mathsf{L}_{\mathsf{FC}}(h, 1; k), \mathsf{L}_{\mathsf{FC}^{-1}}(\tau, 0, \mathsf{o}; k))$. This takes time $t_{\mathsf{H}} + (2\ell+1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}$ and one query to FC and one to FC^{-1}.

When A outputs the ith decryption query on input c, EE^i simply behaves as for a normal decryption query except that instead of calling her oracle on input $(h, 1)$, she outputs $(h, 1, k_0)$ as her prediction. This takes time $t_{\mathsf{H}} + (2\ell+1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}$ and one query to FC^{-1}.

Thus, in total EE^i runs in time bounded by $t + (q_L + q_E + q_D + 1)[t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}] \leq t_1$, does at most $2q_L$ modelling queries and at most $q_E + q_D$ queries to FC and $q_D + 1$ to FC^{-1}.

Bounding $|\Pr[E_{3^i}] - \Pr[E_{3^{i+1}}]|$ and $|\Pr[E_3] - \Pr[E_4]|$. Since EE^i is $(2q_L, q_E + q_D, q_D + 1, t_1)$-adversary, FC is a $(2q_L, q_E + q_D, q_D + 1, t_1, \epsilon_{\mathsf{sU\text{-}L2}})$-unpredictable forkcipher, and Game 3^i and Game 3^{i+1} are the same except if the ith decryption query is the first fresh and valid verification query, then

$$|\Pr[E_3^i] - \Pr[E_{3^{i+1}}]| = \Pr[\mathsf{B} \text{ wins }] \leq \epsilon_{\mathsf{sU\text{-}L2}}.$$

$$\text{So, } |\Pr[E_3] - \Pr[E_4]| \leq \sum_{i=0}^{q_V+1} |\Pr[E_{3^i}] - \Pr[E_{4^{i+1}}]| \leq (q_V + 1)\epsilon_{\mathsf{sU\text{-}L2}}$$

Concluding the proof. We can conclude the proof, since $\Pr[E_4] = 0$, since none of the q_V decryption query and the decryption query induced by the forgery output of A can be fresh and valid. Thus,

$$\Pr[E_0] \leq \Pr[E_4] + \sum_{i=0}^{3} |\Pr[E_i] - \Pr[E_{i+1}]| \leq \epsilon_{\mathsf{CR}} + \frac{2q_V(q_V-1)}{2}\epsilon_{\mathsf{sU\text{-}L2}} + (q_V+1)\epsilon_{\mathsf{sU\text{-}L2}} = \epsilon.$$

B.5 Nonce-Misuse-Security of ForkDTE 1 and 2

Theorem 5. *Let* FC *be a* $(q_E, q_D + 1, t_1, \epsilon_{\mathsf{PRFP}})$*-pseudo random forkcipher permutation. Let* H *be a* $(t_2, \epsilon_{\mathsf{CR}})$*-collision resistant hash function. Let* E *be a* $(2, t_3, \epsilon_{\mathsf{PRF}})$-$\mathsf{PRF}$ *Then* ForkDTE1 *(and* ForkDTE2*) is* (q_E, q_D, t, ϵ)-nmAE*-secure with*

$$\epsilon \leq \epsilon_{\mathsf{PRFP}} + \epsilon_{\mathsf{CR}} + q_E(L+1)\epsilon_{\mathsf{PRF}} + (q_D + 1 + q_E^2 + q_E^2(L+1)^2)2^{-N},$$

where ForkDTE1 *encrypts at most* Ln*-bits message,* $t_1 = t + (q_E + q_D + 1)[t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}]$, $t_2 = t + (q_E + q_D + 1)[2t_{\mathsf{f}} + t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}]$, $t_3 = t + (q_E + q_D + 1)[2t_{\mathsf{f}} + t_{\mathsf{H}} + (2L+1)t_{\mathsf{E}}]$ *with* t_{H} *the time needed to execute once the hash function* H*,* t_{E} *to execute* E*, and* t_{f} *to randomly sample a random permutation.*

ForkDTE1 and 2 have the same encryption algorithm and its verification algorithm gives the same output (without leakage). Thus, a proof for the first is the same as a proof of the second scheme.

Proof. We do not give the complete proof here. We only give the proof until we have arrived to a step which is already covered in the original DTE paper [13].

We use a sequence of games Game 0, ... , Game 2. We denote with E_i the event that the output of Game i is 1, that is, that the adversary wins.

Game 0. This is the nmAE game where the adversary A is playing against ForkDTE1.

Game 1. It is Game 0, where we replace FC_k with its ideal counterpart.

Transition between Game 0 and 1. Since Game 0 and Game 1 are the same except for the use of FC, we need to build the probability an adversary distinguish the use of FC to its ideal counterpart. To do this, we build a $(q_{\mathsf{FC}}, q_{\mathsf{FC}^{-1}}, t_1)$-adversary B which has access to two oracles which are either implemented with $\mathsf{FC}_k, \mathsf{FC}_k^{-1}$ or their ideal counterparts. B works as follows: At the start of the game B obtains the key of the hash function, s, which she forwards to A. Moreover, B has a list $\mathcal{S}$ which is empty.

When A does an encryption query on input (n, m), B simply computes $h = \mathsf{H}_s(n\|m)$, and calls her oracle on input (h, b) obtaining (τ, k_0). From k_0, B computes $y_0 = \mathsf{E}_{k_0}(p_B)$, and $c_0 = y_0 \oplus n$. Then, she parses m in n-bit blocks, $m_1, \ldots, m_\ell$. After that, for all $i = 1, \ldots, \ell$, B computes $k_i = \mathsf{E}_{k_{i-1}}(p_A)$, $y_i = \mathsf{E}_{k_i}(p_B)$, and $c_i = \pi_{|m_i|}(y_i) \oplus m_i$. Finally, she returns A $c = (\tau, C)$, with $C = (c_0, \ldots, c_\ell)$ and she adds c to $\mathcal{S}$. This takes one oracle query to FC_k and time $t_{\mathsf{H}} + (2\ell + 1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L + 1)t_{\mathsf{E}}$, since $\ell \leq L$.

When A does a decryption query on input c, she parses it in n-bits blocks, $\tau, c_0, c_1, \ldots, m_\ell$. Then, B simply calls her inverse oracle on input $(\tau, 0, \mathsf{b})$, obtaining $(\tilde{h}, k_0)$. From k_0, B computes $y_0 = \mathsf{E}_{k_0}(p_B)$, and $n = y_0 \oplus c_0$. After that, for all $i = 1, \ldots, \ell$, B computes $k_i = \mathsf{E}_{k_{i-1}}(p_A)$, $y_i = \mathsf{E}_{k_i}(p_B)$, and $m_i = \pi_{|m_i|}(y_i) \oplus c_i$. Finally, she computes $h = \mathsf{H}_s(n\|m)$ and checks if $h \stackrel{?}{=} \tilde{h}$. If it is the case, B returns A $m = (m_1, \ldots, m_\ell)$; otherwise, $\perp$. This takes one oracle query to FC_k^{-1} and time $t_{\mathsf{H}} + (2\ell + 1)t_{\mathsf{E}} \leq t_{\mathsf{H}} + (2L + 1)t_{\mathsf{E}}$, since $\ell \leq L$.

When A outputs its forgery c, she proceeds as for a normal decryption query except that she does not return anything to A. Instead, if at the end of the verification $h = \tilde{h}$ and $c \notin \mathcal{S}$, B outputs 1; otherwise 0.

Thus, in total B does q_E queries to FC, $q_D + 1$ to FC^{-1} and runs in time bounded by $t + (q_E + q_D + 1)[t_{\mathsf{H}} + (2L + 1)t_{\mathsf{E}}] = t_1$.

Bounding $|\Pr[E_0] - \Pr[E_1]|$. If the oracles B has access to are implemented by $(\mathsf{FC}_k, \mathsf{FC}_k^{-1})$, B is correctly simulating Game 0 for A; otherwise, Game 1. Since B is $(q_E, q_D + 1, t_1)$-adversary, and FC is a $(q_E, q_D + 1, t_1, \epsilon_{\mathsf{CR}})$-PRFP secure forkcipher, then

$$|\Pr[E_0] - \Pr[E_1]| = |\Pr[1 \leftarrow \mathsf{B}^{\mathsf{FC},\mathsf{FC}^{-1}}] - \Pr[1 \leftarrow \mathsf{B}^{\mathsf{f},\mathsf{f}^{-1}}]| \leq \epsilon_{\mathsf{PRFP}}.$$

Game 2

It is Game 2, where we replace the second permutation f_1, with $\mathsf{f}_1' = \mathsf{f}_1 \circ \mathsf{f}_0$. Since f_1 is a random permutation, it is impossible to distinguish f_1' from, f_1. Now, we are exactly in the situation for DTE2 after we have replaced the tweakable blockcipher with a random permutation.

B.6 nAE-Security of ForkDTE 1 and 2

This follows from the fact that nonce-misuse security definition (Def. 15) implies nAE-security (Def. 14) since the latter definition is the first with an additional requirement: the adversary is not allowed to repeat the nonce n in different encryption queries.

C Modifying ForkDTE to Accomodate Associated Data

To modify ForkDTE to accommodate associated data, it is enough to replace $h = \mathsf{H}_s(n\|m)$, with $h = \mathsf{H}_s(n, a, m)$. We need to be careful (and not to use directly $h = \mathsf{H}_s(n\|a\|m)$ because it would be easy to find "forgeries" (for example using $a = (a_0, a_1), m = m_0$, and $a' = a_0, m' = (a_1, m_0)$).

D Algorithms

Algorithm 3 The PSV leakage-resilient iv-based (Def. 18 encryption scheme [31]. E^* is a strongly protected implementation of E.

- Gen:
  - $k \xleftarrow{\$} \mathcal{K}$
  - $p_A, p_B \xleftarrow{\$} \{0,1\}^N$ (p_A, p_B are public parameters)
- $\mathsf{Enc}_k(\mathsf{iv}, m)$:
  - Parse $m = (m_1, m_2, \ldots, m_\ell)$ in N-bit blocks
  - $k_1 = \mathsf{E}^*_k(\mathsf{iv})$ first ephemeral key k_1 generation
  - $y_1 = \mathsf{E}_{k_1}(p_B)$ first pseudorandom block y_1 generation
  - $c_1 = y_1 \oplus m_1$ first ciphertext block c_1 generation
  - For $i = 2, \ldots \ell$
    - $k_i = \mathsf{E}_{k_{i-1}}(p_A)$ i^{th} ephemeral key k_i generation
    - $y_i = \mathsf{E}_{k_i}(p_B)$ i^{th} pseudorandom block y_i generation
    - $c_i = \pi_{|m_i|}(y_i) \oplus m_i$ i^{th} ciphertext block c_i generation
  - Return $c = (c_1, \ldots, c_\ell)$
- $\mathsf{Dec}_k(\mathsf{iv}, c)$:
  - Parse $c = (c_1, c_2, \ldots, c_\ell)$ in N-bit blocks
  - $k_1 = \mathsf{E}^*_k(\mathsf{iv})$
  - $y_1 = \mathsf{E}_{k_1}(p_B)$
  - $m_1 = y_1 \oplus c_1$
  - For $i = 2, \ldots \ell$
    - $k_i = \mathsf{E}_{k_{i-1}}(p_A)$
    - $y_i = \mathsf{E}_{k_i}(p_B)$
    - $m_i = \pi_{|c_i|}(y_i) \oplus c_i$
  - Return $m = (m_1, \ldots, m_\ell)$

E Additional figures

Algorithm 4 ForkTMAC, a sUF−L2-secure MAC based on a forkcipher.

We use a hash function H whose output is $2n$-bit long.

– Gen:
- $k \xleftarrow{\$} \mathcal{K}$
- $s \xleftarrow{\$} \mathcal{HK}$ (s is a public parameter)

– $\mathsf{Mac}_k(m)$:
- $h = \mathsf{H}_s(m)$ // digest
- Parse h in n -bit blocks ($h = h_1 \| h_2$)
- $\tau = \mathsf{FC}_k^{h_2}(h_1, 0)$ // tag
- Return τ

– $\mathsf{Vrfy}_k(m, \tau)$:
- $h = \mathsf{H}_s(m)$
- Parse h in n -bit blocks ($h = h_1 \| h_2$)
- $v = \mathsf{FC}_k^{h_2}(h_1, 1)$
- $\tilde{v} = \mathsf{FC}_k^{-1,h_2}(\tau, 0, \mathsf{o})$
- If $v = \tilde{v}$ Return $\top$, Else Return $\bot$

Algorithm 5 HBC2 [14], a sUF−L2 MAC.

We use a hash function H whose output is n-bit long, and a BC $\mathsf{E} : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$

– Gen:
- $k \xleftarrow{\$} \mathcal{K}$
- $s \xleftarrow{\$} \mathcal{HK}$ (s is a public parameter)

– $\mathsf{Mac}_k(m)$:
- $h = \mathsf{H}_s(m)$ // digest
- $\tau = \mathsf{E}_k(h)$ // tag
- Return τ

– $\mathsf{Vrfy}_k(m, \tau)$:
- $h = \mathsf{H}_s(m)$
- $\tilde{h} = \mathsf{E}_k^{-1}(\tau)$
- If $h = \tilde{h}$ Return $\top$, Else Return $\bot$

Algorithm 6 HTBC [9], a sUF−L2 MAC.

We use a hash function H whose output is $2n$-bit long, and a TBC $\mathsf{E} : \mathcal{K} \times \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$

– Gen:
  - $k \xleftarrow{\$} \mathcal{K}$
  - $s \xleftarrow{\$} \mathcal{HK}$ (s is a public parameter)

– $\mathsf{Mac}_k(m)$:
  - $h = \mathsf{H}_s(m)$ // digest
  - Parse h in n -bit blocks ($h = h_1 \| h_2$)
  - $\tau = \mathsf{E}_k^{h_2}(h_1)$ // tag
  - Return τ

– $\mathsf{Vrfy}_k(m, \tau)$:
  - $h = \mathsf{H}_s(m)$
  - Parse h in n -bit blocks ($h = h_1 \| h_2$)
  - $\tilde{h}_1 = \mathsf{E}_{k}^{-1,h_2}(\tau)$
  - If $h_1 = \tilde{h}_1$ Return $\top$, Else Return $\perp$

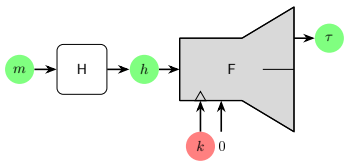

Fig. 5. The tag-generation of ForkMAC - Alg. 1.

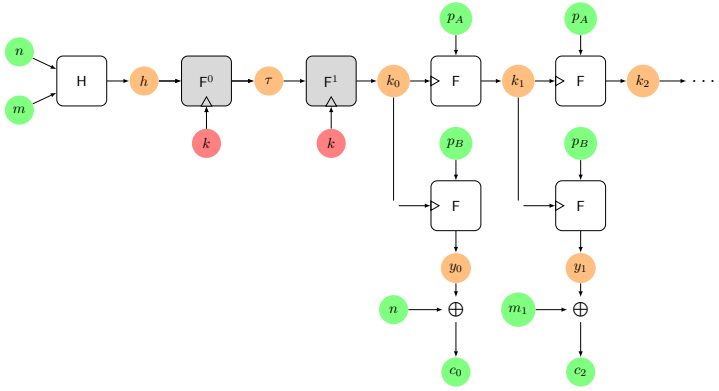

Fig. 6. The encryption scheme of DTE2 [14] - Alg. 2. From k_0 we have PSV [31].

The Add, and Fresh algorithms for the sU−L2 experiment.

Algorithm Add$((x, sel, y), \mathcal{C})$:
  If $sel = 0$
    If $(x, \cdot, \cdot) \in \mathcal{C}$
      If $(x, \mathsf{gu}, y_1) \in \mathcal{C}$
        Return $\mathcal{C} \leftarrow (\mathcal{C} \setminus \{(x, \mathsf{gu}, y_1)\}) \cup \{(x, y, y_1)\}$
      Else Return $\mathcal{C}$
    If $(\cdot, y, \cdot) \in \mathcal{C}$
      If $(\mathsf{gu}, y, y_1) \in \mathcal{C}$
        Return $\mathcal{C} \leftarrow (\mathcal{C} \setminus \{(\mathsf{gu}, y, y_1)\}) \cup \{(x, y, y_1)\}$
      Else Return $\mathcal{C}$
    Else Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(x, y, \mathsf{gu})\}$
  If $sel = 1$
    If $(x, \cdot, \cdot) \in \mathcal{C}$
      If $(x, y_0, \mathsf{gu}) \in \mathcal{C}$
        Return $\mathcal{C} \leftarrow (\mathcal{C} \setminus \{(x, y_0, \mathsf{gu})\}) \cup \{(x, y_0, y)\}$
      Else Return $\mathcal{C}$
    Else If $(\cdot, \cdot, y) \in \mathcal{C}$
      If $(\mathsf{gu}, y_0, y) \in \mathcal{C}$
        Return $\mathcal{C} \leftarrow (\mathcal{C} \setminus \{(\mathsf{gu}, y_0, y)\}) \cup \{(x, y_0, y)\}$
      Else Return $\mathcal{C}$
    Else Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(x, \mathsf{gu}, y)\}$
  If $sel = \mathsf{b}$
    $y = (y_0, y_1)$
    If $(x, \cdot, \cdot) \in \mathcal{C}$
      Return $\mathcal{C} \leftarrow (\mathcal{C} \setminus \{(x, \cdot, \cdot)\}) \cup \{(x, y_0, y_1)\}$
    Else If $(\cdot, y_0, \cdot) \in \mathcal{C}$
      Return $\mathcal{C} \leftarrow (\mathcal{C} \setminus \{(\cdot, y_0, \cdot)\}) \cup \{(x, y_0, y_1)\}$
    Else Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(x, y_0, y_1)\}$

Algorithm Fresh$((x, sel, z), \mathcal{C})$:
  If $sel = 0$
    If $(x, z, \cdot) \in \mathcal{C}$
      Return 0
  If $sel = 1$
    If $(x, \cdot, z) \in \mathcal{C}$
      Return 0
  If $sel = \mathsf{o}$
    If $(\cdot, x, z) \in \mathcal{C}$
      Return 0
  Return 1

Table 5. The Add, and Fresh algorithms for the strong unpredictability with leakage in evaluation and inversion experiment (Tab. 2).

The AddI algorithm for the $\mathsf{sU\text{-}L2}$ experiment.

Algorithm $\mathsf{AddI}((x, sel, sel', y), \mathcal{C})$:

If $sel = 0$
  If $sel' = \mathsf{i}$
    If $(\cdot, x, \cdot) \in \mathcal{C}$
      If $(\mathsf{gu}, x, y_1) \in \mathcal{C}$
        $\mathcal{C} \leftarrow \mathcal{C} \setminus \{(\mathsf{gu}, x, y_1)\}$
        Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(y, x, y_1)\}$
      Else Return $\mathcal{C}$
    If $(y, \cdot, \cdot) \in \mathcal{C}$
      If $(y, \mathsf{gu}, y_1) \in \mathcal{C}$
        $\mathcal{C} \leftarrow \mathcal{C} \setminus \{(y, \mathsf{gu}, y_1)\}$
        Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(y, x, y_1)\}$
      Else Return $\mathcal{C}$
    Else Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(y, x, \mathsf{gu})\}$
  If $sel' = \mathsf{o}$
    If $(\cdot, x, \cdot) \in \mathcal{C}$
      If $(z, x, \mathsf{gu}) \in \mathcal{C}$
        $\mathcal{C} \leftarrow \mathcal{C} \setminus \{(z, x, \mathsf{gu})\}$
        Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(z, x, y)\}$
      Else Return $\mathcal{C}$
    If $(\cdot, \cdot, y) \in \mathcal{C}$
      If $(z, \mathsf{gu}, y) \in \mathcal{C}$
        $\mathcal{C} \leftarrow \mathcal{C} \setminus \{(z, \mathsf{gu}, y)\}$
        Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(z, x, y)\}$
      Else Return $\mathcal{C}$
    Else Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(\mathsf{gu}, x, y)\}$
  If $sel = \mathsf{b}$
    $y = (z, y_1)$
    If $(z, \cdot, \cdot) \in \mathcal{C}$
      $\mathcal{C} \leftarrow \mathcal{C} \setminus \{(z, \cdot, \cdot)\}$
      Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(z, x, y_1)\}$
    If $(\cdot, x, \cdot) \in \mathcal{C}$
      $\mathcal{C} \leftarrow \mathcal{C} \setminus \{(\cdot, x, \cdot)\}$
      Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(z, x, y_1)\}$
    Else Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(z, x, y_1)\}$

If $sel = 1$
  If $sel' = \mathsf{i}$
    If $(\cdot, \cdot, x) \in \mathcal{C}$
      If $(\mathsf{gu}, y_0, x) \in \mathcal{C}$
        $\mathcal{C} \leftarrow \mathcal{C} \setminus \{(\mathsf{gu}, y_0, x)\}$
        Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(y, y_0, x)\}$
      Else Return $\mathcal{C}$
    If $(y, \cdot, \cdot) \in \mathcal{C}$
      If $(y, y_0, \mathsf{gu}) \in \mathcal{C}$
        $\mathcal{C} \leftarrow \mathcal{C} \setminus \{(y, y_0, \mathsf{gu})\}$
        Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(y, y_0, x)\}$
      Else Return $\mathcal{C}$
    Else Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(y, \mathsf{gu}, x)\}$
  If $sel' = \mathsf{o}$
    If $(\cdot, \cdot, x) \in \mathcal{C}$
      If $(z, \mathsf{gu}, x) \in \mathcal{C}$
        $\mathcal{C} \leftarrow \mathcal{C} \setminus \{(z, \mathsf{gu}, x)\}$
        Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(z, y, x)\}$
      Else Return $\mathcal{C}$
    If $(\cdot, y, \cdot) \in \mathcal{C}$
      If $(z, y, \mathsf{gu}) \in \mathcal{C}$
        $\mathcal{C} \leftarrow \mathcal{C} \setminus \{(z, y, \mathsf{gu})\}$
        Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(z, y, x)\}$
      Else Return $\mathcal{C}$
    Else Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(\mathsf{gu}, y, x)\}$
  If $sel = \mathsf{b}$
    $y = (z, y_0)$
    If $(z, \cdot, \cdot) \in \mathcal{C}$
      $\mathcal{C} \leftarrow \mathcal{C} \setminus \{(z, \cdot, \cdot)\}$
      Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(z, y_0, y)\}$
    If $(\cdot, \cdot, x) \in \mathcal{C}$
      Return $\mathcal{C} \leftarrow \mathcal{C} \setminus \{(\cdot, \cdot, x)\}$
      Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(z, y_0, x)\}$
    Else Return $\mathcal{C} \leftarrow \mathcal{C} \cup \{(z, y_0, x)\}$

Table 6. The AddI algorithm for the strong unpredictability with leakage in evaluation and inversion experiment (Tab. 2).

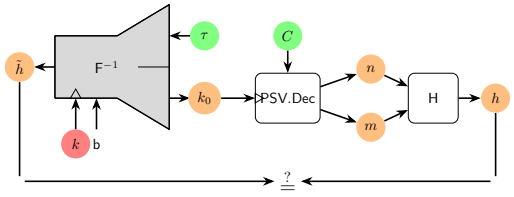

Fig. 7. The decryption of $\mathsf{ForkDTE1}$ - Alg. 2.

Algorithm 7 The ForkDTE1, ForkDTE2,where we have replaced the PSV encryption with the encryption of FEDT [18]. E is a leak-free forkcipher and FC is a forkcipher.

– Gen:
  - $k \xleftarrow{\$} \mathcal{K}$ DTE 2 ForkDTE1 and 2 ForkDTE1 ForkDTE2
  - $s \xleftarrow{\$} \mathcal{HK}$
  - $p_A, p_B \xleftarrow{\$} \{0,1\}^N$ (s, p_A, p_B are public parameters)

– $\mathsf{Enc}_k(n, m)$:
  - $h = \mathsf{H}_s(n \| m)$ digest
  - $\tau = \mathsf{E}_k^0(h)$ tag
  - $k_0 = \mathsf{E}_k^1(\tau)$ generate the first ephemeral key
  - $(\tau, k_0) = \mathsf{FC}_k(h, \mathsf{b})$ tag and generate the first ephemeral key
  - Parse $m = (m_1, m_2, \ldots, m_\ell)$ in N-bit blocks ...and encrypt
  - $(k_1, k_2) = \mathsf{E}_{k_0}(0^n, \mathsf{b})$
  - For $i = 3, 5, 7, \ldots, 2l-1$ create a tree of random value
    - $a = (i-1)/2$
    - $(k_i, k_{i+1}) = \mathsf{E}_{k_a}(0^n, \mathsf{b})$
  - $c_0 = k_{l-1} \oplus n$ encrypt the nonce
  - For $i = 1, \ldots l$ encrypt the message
    - $c_i = \pi_{|m_i|}(k_{l-1+i}) \oplus m_i$
  - $C = (c_0, c_1, \ldots, c_\ell)$
  - Return $c = (\tau, C)$

– $\mathsf{Dec}_k(c)$:
  - Parse $c = (\tau, C)$ with $|\tau| = N$
  - Parse $C = (c_0, c_1, c_2, \ldots, c_\ell)$ in N-bit blocks
  - $k_0 = \mathsf{E}_k^{-1,1}(\tau)$ Recovering the first ephemeral key
  - $(\tilde{h}, k_0) = \mathsf{FC}_k^{-1}(\tau, 0, \mathsf{b})$ Recovering the first ephemeral key and check value
  - $k_0 = \mathsf{FC}_k^{-1}(\tau, 0, \mathsf{o})$ Recovering the first ephemeral key
  - $(k_1, k_2) = \mathsf{E}_{k_0}(0^n, \mathsf{b})$
  - For $i = 3, 5, 7, \ldots, 2l-1$
    - $a = (i-1)/2$
    - $(k_i, k_{i+1}) = \mathsf{E}_{k_a}(0^n, \mathsf{b})$
  - $n = k_{l-1} \oplus c_0$
  - For $i = 1, \ldots l$
    - $m_i = \pi_{|c_i|}(k_{l-1+i}) \oplus c_i$
  - $(n, m) = (n, (m_1, ..., m_\ell))$
  - $h = \mathsf{H}_s(n \| m)$
  - $\tilde{h} = \mathsf{FC}_k^{-1,0}(\tau)$ check value
  - If $h = \tilde{h}$ Return m; Else Return $\perp$
  - If $h = \tilde{h}$ Return m; Else Return $\perp$
  - $\tilde{k_0} = \mathsf{FC}_k(h, 1)$ check value
  - If $k_0 = \tilde{k_0}$ Return m; Else Return $\perp$ check value

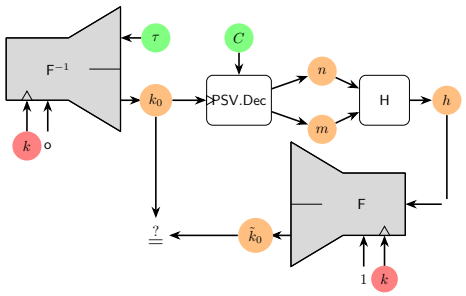

Fig. 8. The decryption of ForkDTE2 - Alg. 2.