Continuously Non-malleable Codes with Split-State Refresh

Antonio Faonio[1(✉)], Jesper Buus Nielsen[2], Mark Simkin[2], and Daniele Venturi[3]

[1] IMDEA Software Institute, Madrid, Spain
antonio.faonio@imdea.org
[2] Aarhus University, Aarhus, Denmark
[3] Sapienza University of Rome, Rome, Italy

Abstract. Non-malleable codes for the split-state model allow to encode a message into two parts, such that arbitrary independent tampering on each part, and subsequent decoding of the corresponding modified codeword, yields either the same as the original message, or a completely unrelated value. Continuously non-malleable codes further allow to tolerate an unbounded (polynomial) number of tampering attempts, until a decoding error happens. The drawback is that, after an error happens, the system must self-destruct and stop working, otherwise generic attacks become possible.

In this paper we propose a solution to this limitation, by leveraging a split-state refreshing procedure. Namely, whenever a decoding error happens, the two parts of an encoding can be locally refreshed (i.e., without any interaction), which allows to avoid the self-destruct mechanism. An additional feature of our security model is that it captures directly security against continual leakage attacks. We give an abstract framework for building such codes in the common reference string model, and provide a concrete instantiation based on the external Diffie-Hellman assumption.

Finally, we explore applications in which our notion turns out to be essential. The first application is a signature scheme tolerating an arbitrary polynomial number of split-state tampering attempts, without requiring a self-destruct capability, and in a model where refreshing of the memory happens only after an invalid output is produced. This circumvents an impossibility result from a recent work by Fuijisaki and Xagawa (Asiacrypt 2016). The second application is a compiler for tamper-resilient RAM programs. In comparison to other tamper-resilient compilers, ours has several advantages, among which the fact that, for the first time, it does not rely on the self-destruct feature.

Keywords: Non-malleable codes · Tamper-resilient cryptography

1 Introduction

Tampering attacks are subtle attacks that undermine the security of cryptographic implementations by exploiting physical phenomena that allow to modify the underlying secrets. Indeed, a long line of works (see, e.g., [3,4,16,18])

B. Preneel and F. Vercauteren (Eds.): ACNS 2018, LNCS 10892, pp. 121–139, 2018.
https://doi.org/10.1007/978-3-319-93387-0_7

has established that black-box interaction with a tampered implementation can potentially expose the entire content of the secret memory. Given this state of affairs, protecting cryptographic schemes against tampering attacks has become an important goal for modern cryptographers.

An elegant solution to the threat of tampering attacks against the memory comes from the notion of non-malleable codes (NMCs), put forward by Dziembowski et al. [10]. Intuitively, a non-malleable encoding $(\mathsf{Encode}, \mathsf{Decode})$ allows to encode a value M into a codeword $C \leftarrow\!\!\$\ \mathsf{Encode}(M)$, with the guarantee that a modified codeword $\widetilde{C} = f(C)$ w.r.t. a tampering function $f \in \mathcal{F}$, when decoded, yields either M itself, or a completely unrelated value. An important parameter for characterizing the security guarantee offered by NMCs is the class of modifications $\mathcal{F}$ that are supported by the scheme. Since non-malleability is impossible to obtain for arbitrary (albeit efficient) modifications,[1] research on NMCs has focused on constructing such schemes in somewhat restricted, yet interesting, models. One such model that has been the focus of intensive research (see, e.g., [1,2,12,17]) is the split-state model, where the codeword C consists of two parts (C_0, C_1) that can be modified independently (yet arbitrarily). This setting is also the focus of this paper.

Unfortunately, standard NMCs protect only against a single tampering attack,[2] To overcome this limitation, Faust *et al.* [12] introduced *continuously* non-malleable codes (CNMCs for short), where the attacker can tamper for an unbounded (polynomial) number of times with the codeword, until a decoding error happens which triggers the self-destruction of the device. As argued in [12], the self-destruct capability is necessary, as each decoding error might be used to signal one bit of information about the target codeword.

Another desirable feature of non-malleable codes is their ability to additionally tolerate leakage attacks, by which the adversary can obtain partial information on the codeword while performing a tampering attack. Note that in the split-state model this means that the adversary can leak independently from the two parts C_0 and C_1. All previous constructions of leakage-resilient NMCs either achieve security in the so-called bounded-leakage model [1,12,17], where the total amount of leakage (from each part) is upper-bounded by a value ℓ that is a parameter of the scheme, or only satisfy non-continuous non-malleability [11].

Our Contributions. We introduce a new form of CNMCs (dubbed R-CNMCs) that include a split-state algorithm for refreshing a valid codeword. The refresh procedure is invoked either after a decoding error happens, or in order to amplify resilience to leakage, and takes place *directly* on the memory and without the

[1] As it can be seen by considering the tampering function that first decodes the codeword, flips one bit of the message, and then encodes the result.

[2] When using NMCs to obtain security against memory tampering, one can still obtain security against continuous attacks by enforcing a re-encoding of the secret key after each invocation; however, this comes with several disadvantages [11], among which the fact that the encoding process is considerably more complex than the decoding process.

need of a central unit. Our new model has a number of attractive features, which we emphasize below.

- It captures security in the so-called noisy-leakage model, where between each refresh the adversary can leak an arbitrary (yet independent) amount of information on the two parts C_0, C_1, as long as the leakage does not reveal (information-theoretically) more than ℓ bits of information. Importantly, this restriction is well-known to better capture realistic leakage attacks.
- It avoids the need for the self-destruct capability in some applications. Besides mitigating simple denial-of-service attacks, this feature is useful in situations where a device (storing an encoding of the secret state) is not in the hands of the adversary (e.g., because it has been infected by a malware), as it still allows to (non-interactively) refresh the secret state and continue to safely use the device in the wild.

Our first contribution is an abstract framework for constructing R-CNMCs, which we are able to instantiate under the external Diffie-Hellman assumption. This constitutes the first NMC that achieves at the same time continuous non-malleability and security under continual noisy leakage, in the split-state model (assuming an untamperable common reference string).

Next, we explore applications of R-CNMCs. As second contribution, we show how to construct a split-state[3] signature scheme resilient to continuous (non-persistent) tampering and leakage attacks, without relying on the self-destruct capability, and where the memory content is refreshed in case a decoding error is triggered. Interestingly, Fujisaki and Xagawa [13] recently showed that such a notion is impossible to achieve for standard (i.e., non split-state) signature schemes, even if the self-destruct capability is available; hence, our approach can be interpreted as a possible way to circumvent the impossibility result in [13].

Our third contribution consists of two generic compilers for protecting random access machine (RAM) computations against tampering attacks. Here, we build on the important work of Dachman-Soled *et al.* [7], who showed how to compile any RAM to be resilient to continual tampering and leakage attacks, by relying both on an update and a self-destruct mechanism. We refer the reader to Sect. 5 for further details on our RAM compilers. Below, we highlight the main technical ideas behind our code construction.

Code Construction. The starting point of our code construction is the recent work of Faonio and Nielsen [11]. The scheme built in [11] follows a template that originates in the work of Liu and Lysyanskaya [17], in which the left side of the encoding stores the secret key sk of a PKE scheme, whereas the right side of the encoding stores a ciphertext c, encrypting the encoded message M, plus a non-interactive zero-knowledge (NIZK) argument that proves knowledge of the secret key under the label c; the PKE scheme is chosen to be a continual-leakage resilient storage friendly PKE (CLRS friendly PKE for short) scheme

[3] This means that the signing key is made of two shares that are stored in two separate parts of the memory, and need to be combined upon signing.

(see Dodis *et al.* [9]), whereas the NIZK is chosen to be a malleable NIZK argument of knowledge (see Chase *et al.* [5]). Such a code was shown to admit a split-state refresh procedure, and, at the same time, to achieve *bounded-time* non-malleability.

The NM code of [11] does not satisfy security against continuous attacks. In fact, an attacker can create two valid codewords (C_0, C_1) and (C_0, C_1') such that $\mathsf{Decode}(C_0, C_1) \neq \mathsf{Decode}(C_0, C_1')$. Given this, the adversary can tamper the left side to C_0 and the right side to either C_1 or C_1' according to the bits of the right side of the target encoding. In a non-persistent model, the adversary can leak all the bits of C_1 without activating the self-destruct mechanism. More in general, for any R-CNMC it should be hard to find two valid codewords (C_0, C_1) and (C_0, C_1') such that $\mathsf{Decode}(C_0, C_1) \neq \mathsf{Decode}(C_0, C_1')$. This property, which we call "message uniqueness", was originally defined in [12].[4]

Going back to the described code construction, an attacker can sample a secret key sk and create two ciphertexts, c_0 for M and c' for M', where $M \neq M'$, together with the corresponding honestly computed NIZKs, and thus break message uniqueness. We fix this issue by further binding the right and the left side of an encoding. To do so, while still be able to refresh the two parts independently, we keep untouched the structure of the right side of the codeword, but we change the message that it carries. Specifically, the ciphertext c in our code encrypts the message M concatenated with the randomness r for a commitment γ that is stored in the left side of the codeword together with the secret key for the PKE scheme. Observe that "message uniqueness" is now guaranteed by the binding property of the commitment scheme. Our construction additionally includes another NIZK for proving knowledge of the committed value under the label sk, in order to further link together the left and the right side of the codeword.

Proof Strategy. Although our construction shares similarities with previous work, our proof techniques diverge significantly from the ones in [11,12]. The main trick of [12] is to show that given one half of the codeword it is possible to fully simulate the view of the adversary in the tampering experiment, until a decoding error happens. To catch when a decoding error happens, [12] carries on two independent simulators in an interleaved fashion; as they prove, a decoding error happens exactly when the outputs of the two simulations diverge. The main obstacle they faced is how to succinctly compute the index where the two simulations diverge so that they can reduce to the security of the inner leakage-resilient scheme storage (see Daví *et al.* [8]) they rely on. To solve this, [12] employs an elegant dichotomic search-by-hash strategy over the partial views produced by the two simulators. At this point the experiment can terminate,

[4] Faust *et al.* also consider "codeword uniqueness", where the fact that $\mathsf{Decode}(C_0, C_1) \neq \mathsf{Decode}(C_0, C_1')$ is not required. However, this flavor of uniquness only allows to rule-out so-called *super* continuous non-malleability, where one asks that not only the decoded value, but the entire modified codeword, be independent of the message. It is easy to see that no R-CNMC can satisfy "codeword uniqueness", as for instance C_1' could be obtained as a valid refresh of C_1.

and thanks to a specific property of the leakage-resilient storage scheme, the simulator can "extract" the view.

Unfortunately, we cannot generalize the above proof strategy to multiple rounds. In fact, the specific property of the leakage-resilient storage scheme they make use of is inherently one shot. Specifically, the property allows the adversary to get an half of the leakage-resilient codeword. However, to allow this the adversary must lose leakage oracle access to the other half of the codeword. In our case, we would need to repeat the above trick again and again, after a decoding error and a subsequent refresh of the target encoding happens; however, once we ask for an entire half of the codeword, even if we refreshed the codeword, we cannot regain access to the leakage oracles[5]. We give a solution to this problem by relying on a simple information-theoretic observation.

Let (X_0, X_1) be two random variables, and consider a process that interleaves the computation of a sequence of leakage functions $g^1, g^2, g^3, \ldots$ from X_0 and from X_1. The process continues until, for some index $i \in \mathbb{N}$, we have that $g^i(X_0) \neq g^i(X_1)$. We claim that $\bar{g}^i(X_0) := g^1(X_0), g^2(X_0), \cdots, g^{i-1}(X_0)$ do not reveal more information about X_0 than what X_1 and the index i already reveal. To see this, consider $\widetilde{\mathbb{H}}_\infty(X_0 \mid \bar{g}^i(X_0))$ to be the average conditional min-entropy of X_0, which is, roughly speaking, the amount (in average) of the uncertainty of X_0 given $\bar{g}^i(X_0)$ as side information. Now, since $\bar{g}^i(X_0)$ and $\bar{g}^i(X_1)$ are exactly the same random variables we can derive[6]:

$$\widetilde{\mathbb{H}}_\infty(X_0 \mid \bar{g}^i(X_0)) = \widetilde{\mathbb{H}}_\infty(X_0 \mid \bar{g}^i(X_1)) \geq \widetilde{\mathbb{H}}_\infty(X_0 \mid X_1,\ i).$$

The above observation implies that the size of the view of the adversary, although much larger than the leakage bound, does reveal only little information.

We can already give a different proof of security for the scheme in [12] where the reduction to the inner-product leakage-resilient storage loses only a factor $O(\kappa)$ in the leakage bound (instead of $O(\kappa \log \kappa)$). Briefly, the idea is to carries on two independent simulators in an interleaved fashion (as in [12]) and, at each invocation, outputting first the hashes[7] of the simulated tampered codeword, then, if the hashes match, leak the full simulated tampered codeword avoiding, in this way, the dichotomic search-by-hash strategy. The information-theoretic observation above guarantees that only the last hashes (which will be different) reveals information. The latter implies that the amount of leakage is bounded by $O(\kappa)$.

2 Preliminaries and Building Blocks

We introduce the cryptographic primitives on which we build. For space reasons, standard notation and formal definitions are deferred to the full version of the paper.

[5] In particular, this property does not hold for a CLRS friendly PKE scheme.

[6] In the last equation, we also use that the output of a function is at most as informative as the input.

[7] By collision resistance of the hash function, if the two hashes match then the simulated tampered codewords are the same for both the simulators.

Oracle Machines. Given a pair of strings $X = (X_0, X_1) \in (\{0,1\}^*)^2$ define the oracle $\mathcal{O}_\infty(X)$ to be the *split-state leakage oracle* that takes as input tuples of the form (β, g), where $\beta \in \{0,1\}$ is an index and g is a function described as a circuit, and outputs $g(X_\beta)$. An adversary A with oracle access to $\mathcal{O}_\infty(X)$ is called *ℓ-valid*, for some $\ell \in \mathbb{N}$, if for all $\beta \in \{0,1\}$ the concatenation of the leakage functions sent by A is an ℓ-leaky function of X_β (i.e., the total amount of leakage does not reduce the entropy of X_β by too much).

Given two PPT interactive algorithms A and B we write $(y_\mathsf{A}; y_\mathsf{B}) \leftarrow (\mathsf{A}(x_\mathsf{A}) \leftrightarrows \mathsf{B}(x_\mathsf{B}))$ to denote the execution of algorithm A (with input x_A) and algorithm B (with input x_B). The string y_A (resp. y_B) is the output of A (resp. B) at the end of such interaction. In particular, we write $\mathsf{A} \leftrightarrows \mathcal{O}_\infty(X)$ to denote A having oracle access to the leakage oracle with input X. Moreover, we write $\mathsf{A} \leftrightarrows \mathsf{B}, \mathsf{C}$ to denote A interacting in an interleaved fashion both with B and with C.

Non-interactive Zero-Knowledge. Let $\mathcal{R} \subseteq \{0,1\}^* \times \{0,1\}^*$ be an NP-relation; the language associated with $\mathcal{R}$ is $\mathcal{L}_\mathcal{R} := \{x : \exists w \text{ s.t. } (x,w) \in \mathcal{R}\}$. We typically assume that given a pair (x,w) it is possible to efficiently verify whether $(x,w) \in \mathcal{R}$ or not. Roughly, a non-interactive argument (NIA) for an NP-relation $\mathcal{R}$ allows to create non-interactive proofs for statements $x \in \mathcal{L}$, when additionally given a valid witness w corresponding to x. More formally, a NIA $\mathcal{NIA} := (\mathsf{CRSGen}, \mathsf{Prove}, \mathsf{Ver})$ for $\mathcal{R}$, with label space Λ, is a tuple of PPT algorithms specified as follows (1) The (randomized) initialization algorithm CRSGen takes as input the security parameter 1^κ, and creates a common reference string (CRS) $\omega \in \{0,1\}^*$; (2) The (randomized) prover algorithm Prove takes as input the CRS ω, a label $\lambda \in \Lambda$, and a pair (x,w) such that $(x,w) \in \mathcal{R}$, and produces a proof $\pi \leftarrow\!\!\$\ \mathsf{Prove}^\lambda(\omega, x, w)$; (3) The (deterministic) verifier algorithm Ver takes as input the CRS ω, a label $\lambda \in \Lambda$, and a pair (x,π), and outputs a decision bit $\mathsf{Ver}^\lambda(\omega, x, \pi)$.

Completeness means that for all CRSs ω output by $\mathsf{CRSGen}(1^\kappa)$, for all labels $\lambda \in \Lambda$, and for all pairs $(x,w) \in \mathcal{R}$, we have that $\mathsf{Ver}^\lambda(\omega, x, \mathsf{Prove}^\lambda(\omega, x, w)) = 1$ with all but a negligible probability. As for security, we require the following properties.

- **Adaptive multi-theorem zero-knowledge:** Honestly computed proofs do not reveal anything beyond the validity of the statement, and, as such, can be simulated given only the statement itself.
- **Φ-Malleable label simulation extractability:** Our construction will be based on a so-called label-malleable NIA, parametrized by a set of label transformations Φ, where for any $\phi \in \Phi$, the co-domain of ϕ is a subset of Λ. For such NIAs, given a proof π under some label $\lambda \in \Lambda$, one can efficiently generate new proofs π' for the same statement under a different label $\phi(\lambda)$, for any $\phi \in \Phi$ (without knowing a witness); this is formalized via an additional (randomized) label-derivation algorithm LEval, which takes as input the CRS ω, a transformation $\phi \in \Phi$, a label $\lambda \in \Lambda$, and a pair (x,π), and outputs a new proof π'. The property we need intuitively says that a NIA satisfies knowledge soundness, except that labels are malleable w.r.t. Φ. More in details,

there exists a knowledge extractor K that for any adversary which can query polynomially many simulated proofs of false statements and then it produces a tuple (x, λ, π) where π is a valid NIZK proof for (x, λ) can extract either (1) the witness for x or (2) a transformation $\phi \in \Lambda$ which maps $\phi(\lambda') = \lambda$ and (x, λ') was precedently queried by the adversary.
- **Label derivation privacy:** It is hard to distinguish a fresh proof for some statement x (with witness w) under label λ, from a proof re-randomized using algorithm LEval w.r.t. some function $\phi \in \Phi$; moreover, the latter should hold even if (x, w, λ, ϕ) are chosen adversarially (possibly depending on the CRS).

Public-Key Encryption. A public-key encryption (PKE) scheme is a tuple of algorithms $\mathcal{PKE} = (\mathsf{Setup}, \mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ with the usual syntax. We will require two additional algorithms, the first one to re-randomize a given ciphertext, and the second one for re-randomizing the secret key (without changing the corresponding public key). More formally: The (randomized) algorithm $\mathsf{UpdateC}$ takes as input a ciphertext c, and outputs a new ciphertext c'; The (randomized) algorithm $\mathsf{UpdateS}$ takes as input a secret key sk, and outputs a new secret key sk'.

As for security, we require the following properties.

- **CLRS friendly PKE security:** This property is essentially a strengthening of semantic security, where the adversary can additionally observe noisy independent leakages from $S_0 = sk$ and $S_1 = c$ (c is the challenge ciphertext).
- **Ciphertext-update privacy:** The distributions of fresh and updated ciphertexts are the same.
- **Secret-key-update privacy:** The distributions of fresh and updated keys are the same.

Additionally, we make use of a (non-interactive) commitment scheme $\mathcal{COM} = (\mathsf{CRSGen}, \mathsf{Commit})$ with statistical hiding and computationally binding and of an authenticated encryption scheme $\mathcal{SKE} := (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$. This notions are standard therefore we defer the definitions to the full version of the paper.

3 Non-malleability with Refresh

A coding scheme in the CRS model is a tuple of polynomial-time algorithms $\mathcal{CS} = (\mathsf{Init}, \mathsf{Encode}, \mathsf{Decode})$ with the following syntax: (1) The (randomized) initialization algorithm Init takes as input the security parameter 1^κ, and outputs a CRS $\omega \in \{0,1\}^*$; (2) The (randomized) encoding algorithm Encode takes as input the CRS ω and a message $M \in \mathcal{M}$, and outputs a codeword $C \in \mathcal{C}$; (3) The (deterministic) decoding algorithm Decode takes as input the CRS ω and a codeword $C \in \mathcal{C}$, and outputs a value $M \in \mathcal{M} \cup \{\perp\}$ (where $\perp$ denotes an invalid codeword). A coding scheme is correct if for all ω output by $\mathsf{Init}(1^\kappa)$, and any $M \in \mathcal{M}$, we have $\mathbb{P}[\mathsf{Decode}(\omega, \mathsf{Encode}(\omega, M)) = M] = 1$, where the probability is taken over the randomness of the encoding algorithm.

We consider coding schemes with an efficient *refreshing algorithm*. Specifically, for a coding scheme $\mathcal{CS}$ we assume there exists a randomized algorithm Rfrsh that, upon input the CRS ω and a codeword $C \in \mathcal{C}$, outputs a codeword $C' \in \mathcal{C}$. For correctness we require that for all ω output by $\mathsf{Init}(1^\kappa)$, we have $\mathbb{P}[\mathsf{Decode}(\omega, \mathsf{Rfrsh}(\omega, C)) = \mathsf{Decode}(\omega, C)] = 1$, where the probability is over the randomness used by the encoding and refreshing algorithms.

Split-State Model. In this paper we are interested in coding schemes in the split-state model, where a codeword consists of two parts that can be refreshed independently and without the need of any interaction. More precisely, given a codeword $C := (C_0, C_1)$, the refresh procedure $\mathsf{Rfrsh}(\omega, (\beta, C_\beta))$, for $\beta \in \{0, 1\}$, takes as input either the left or the right part of the codeword, and updates it. Sometimes we also write $\mathsf{Rfrsh}(\omega, C)$ as a shorthand for the algorithm that independently executes $\mathsf{Rfrsh}(\omega, (0, C_0))$ and $\mathsf{Rfrsh}(\omega, (1, C_1))$.

Correctness here means that for all ω output by $\mathsf{Init}(1^\kappa)$, for all $C \in \mathcal{C}$, and for any $\beta \in \{0, 1\}$, if we let $C' = (C'_0, C'_1)$ be such that $C'_\beta \leftarrow\!\!\$\ \mathsf{Rfrsh}(\omega, (\beta, C_\beta))$ and $C'_{1-\beta} = C_{1-\beta}$, then $\mathbb{P}[\mathsf{Decode}(\omega, C') = \mathsf{Decode}(\omega, C)] = 1$.

3.1 The Definition

We give the security definition for continuously non-malleable codes with split-state refresh (R-CNMCs for short). Our notion compares two experiments, which we denote by **Tamper** and **SimTamper** (cf. Fig. 1). Intuitively, in the experiment **Tamper** we consider an adversary continuously tampering with, and leaking from, a target encoding $C = (C_0, C_1)$ of a message $M \in \mathcal{M}$ (the message can be chosen adaptively, depending on the CRS). For each tampering attempt (f_0, f_1), the adversary gets to see the output $\widetilde{M}$ of the decoding corresponding to the modified codeword $\widetilde{C} = (f_0(C_0), f_1(C_1))$. Tampering is non-persistent, meaning that each tampering function is applied to the original codeword C, until, eventually, a decoding error happens; at this point the adversary is allowed to make one extra tampering query (f_0^*, f_1^*), and, if the corresponding tampered codeword $\widetilde{C}^*$ is valid and is not an encoding of the original message M, it receives a refresh of $\widetilde{C}^*$ (otherwise the adversary receives M or $\perp$). After that, the target encoding C is refreshed, and the adversary can start tampering with, and leaking from, the refreshed codeword. (Below we explain why this extra feature is useful.)

In the experiment **SimTamper**, we consider a simulator $\mathsf{S} = (\mathsf{S}_0, \mathsf{S}_1)$, where S_0 outputs a simulated CRS, while S_1's goal is to simulate the view of the adversary in the real experiment; the simulator S_1, in faking a tampering query (f_0, f_1), is allowed to output a special value $\diamond$, signaling that (it believes) the adversary did not change the encoded message, in which case the experiment replaces $\diamond$ with M; We stress that the simulator S is stateful; in particular algorithms $\mathsf{S}_0, \mathsf{S}_1$ implicitly share a state.

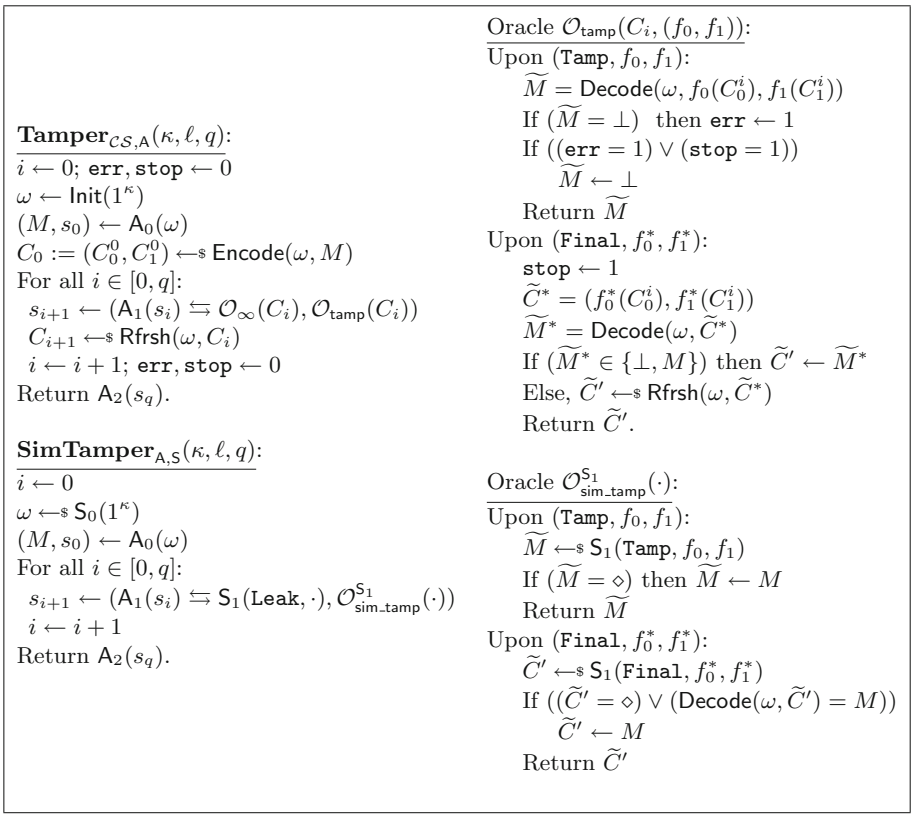

Fig. 1. Experiments defining continuously non-malleable codes with split-state refresh.

Definition 1 (Continuous non-malleability with split-state refresh). *For $\kappa \in \mathbb{N}$, let $\ell = \ell(\kappa)$ be a parameter. We say that a coding scheme $\mathcal{CS}$ is an ℓ-leakage-resilient and continuously non-malleable code with split-state refresh (R-CNMC for short) if for all adversaries $\mathsf{A} := (\mathsf{A}_0, \mathsf{A}_1, \mathsf{A}_2)$, where A_0 and A_2 are PPT algorithms and A_1 is an ℓ-valid deterministic polynomial-time algorithm, there exists a PPT simulator $\mathsf{S} = (\mathsf{S}_0, \mathsf{S}_1)$ and a negligible function $\nu : \mathbb{N} \to [0,1]$ such that, for any polynomial $q(\kappa)$, the following holds:*

$$\left|\mathbb{P}\left[\mathbf{Tamper}_{\mathcal{CS},\mathsf{A}}(\kappa,\ell,q) = 1\right] - \mathbb{P}\left[\mathbf{SimTamper}_{\mathsf{A},\mathsf{S}}(\kappa,\ell,q) = 1\right]\right| \leq \nu(\kappa),$$

where the experiments $\mathbf{Tamper}_{\mathcal{CS},\mathsf{A}}(\kappa,\ell,q)$ and $\mathbf{SimTamper}_{\mathsf{A},\mathsf{S}}(\kappa,\ell,q)$ are defined in Fig. 1.

We give some intuitions on why the extra tampering query is meaningful. First, observe that for (standard) continuously non-malleable codes, the notion of non-persistent tampering is strictly stronger than the notion of persistent tampering. This is because the effect of any sequence of persistent tampering functions

$f^1, f^2, f^3, \cdots$ can be simulated in the non-persistent setting by the sequence of tampering functions $f^1, f^2 \circ f^1, f^3 \circ f_2 \circ f_1, \cdots$. For R-CNMCs, instead, we cannot simulate persistent tampering, as in such a setting the refreshing procedure can be invoked on invalid codewords. The extra tampering query in our definition allows for some flavor of persistent tampering, in that the adversary gets to see a refresh of the tampered codeword, as long as the codeword is valid[8]. Unfortunately, it is impossible to further generalize our definition to handle the refreshing of invalid codewords.[9]

As additional remark, we notice that in the **Tamper** security game the adversary does not have a "direct" access to a refresh oracle (namely, an oracle that, under request of the adversary, would refresh the codeword). We skim this extra detail to not overload the (already heavy) notation. However, the choice comes without loss of any generality. In fact, we can simulate an adversary that makes explicit call to a refreshing oracle by an adversary stop, and return its state (this would indeed trigger a refresh in the experiment), and restart again in the next iteration of the **Tamper** experiment.

4 Code Construction

Let $\mathcal{PKE} = (\mathsf{Setup}, \mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{UpdateC}, \mathsf{UpdateS})$ be a CLRS friendly PKE scheme, with secret-key space $\mathcal{SK}$. We assume there exists an efficient polynomial-time function PK that maps a secret key to the corresponding public key. Let $\mathcal{COM} = (\mathsf{CRSGen}, \mathsf{Commit})$ be a commitment scheme in the CRS model. Consider the following NP-relations, parametrized by the PKE and the commitment scheme, respectively:

$$\mathcal{R}_0 := \{(pk, sk) : \ pk = \mathsf{PK}(sk), sk \in \mathcal{SK}\},$$
$$\mathcal{R}_1 := \{((\omega, \gamma), (M, r)) : \ \gamma = \mathsf{Commit}(\omega, M; r)\}.$$

Let Φ_0 and Φ_1 be two sets of label transformations defined below:

$$\Phi_0 := \{\phi : \ \exists pk, sk \text{ s.t. } (\forall m, r)\ \mathsf{Dec}(sk, \phi(\mathsf{Enc}(pk, m; r))) = m, pk = \mathsf{PK}(sk)\}$$
$$\Phi_1 := \{\phi : \ (\forall sk)\ \mathsf{PK}(sk) = \mathsf{PK}(\phi(sk))\}.$$

[8] A sequence of persistent tampering functions $f^1, f^2, \cdots, f^q$ followed by a refreshing (on tampered codeword) can be simulated in the non-persistent setting by the sequence of concatenation of tampering functions (as described above) and then invoking a final tampering query with tampering function set to $f^1 \circ f^2 \circ ... \circ f^q$.

[9] This can be seen by the following attack. Consider an attacker that computes offline a valid codeword $(\overline{C}_0, \overline{C}_1)$, and then makes two extra tampering queries (in two subsequent rounds, say, i and $i+1$) such that the first query overwrites (C_0^i, C_1^i) with $(C_0^i, \overline{C}_1)$, and the second query overwrites (C_0^{i+1}, C_1^{i+1}) with $(\overline{C}_0, C_1^{i+1})$; by combining the refreshed codewords obtained as output, the adversary gets a refresh of the original codeword, which cannot be simulated in the ideal experiment (recall that the refresh algorithm updates the two shares independently).

Notice that $\mathcal{R}_0, \mathcal{R}_1, \Phi_0$ and Φ_1 are implicitly parametrized by the public parameters $\rho \in \{0,1\}^*$ of the PKE scheme. Finally, let $\mathcal{U}^0$ and $\mathcal{U}_1$ be the following sets of label transformations:

$$\mathcal{U}^0 := \{\mathsf{UpdateC}(\ \cdot\ ; r_u) :\ r_u \in \{0,1\}^*\}$$
$$\mathcal{U}^1 := \{\mathsf{UpdateS}(\ \cdot\ ; r_u) :\ r_u \in \{0,1\}^*\}\,.$$

It is easy to verify that $\mathcal{U}_\beta \subseteq \Phi_\beta$, for $\beta \in \{0,1\}$. In fact, for $\beta = 0$, by the correctness of the PKE scheme, there exists sk such that $\mathbb{P}[\mathsf{Dec}(sk, \mathsf{UpdateC}(\mathsf{Enc}(pk, m))) = m] = 1$ and $pk = \mathsf{PK}(sk)$; similarly, for $\beta = 1$, again by correctness of the PKE scheme, for any $sk' \leftarrow\!\!\$\ \mathsf{UpdateS}(pk, sk)$ we have that $\mathsf{PK}(sk) = \mathsf{PK}(sk')$.

Scheme Description. Let $\mathcal{NIA}_0 = (\mathsf{CRSGen}_0, \mathsf{Prove}_0, \mathsf{Vrfy}_0, \mathsf{LEval}_0)$ and $\mathcal{NIA}_1 = (\mathsf{CRSGen}_1, \mathsf{Prove}_1, \mathsf{Vrfy}_1, \mathsf{LEval}_1)$ be NIAs for the above defined relations $\mathcal{R}_0$ and $\mathcal{R}_1$. Our code $\mathcal{CS} = (\mathsf{Init}, \mathsf{Encode}, \mathsf{Decode})$ works as follows.

- $\underline{\mathsf{Init}(1^\kappa)}$: For $\beta \in \{0,1\}$, sample $\omega_\beta \leftarrow\!\!\$\ \mathsf{CRSGen}_\beta(1^\kappa)$, $\omega \leftarrow \mathsf{CRSGen}(1^\kappa)$, and $\rho \leftarrow \mathsf{Setup}(1^\kappa)$. Return $\overline{\omega} = (\omega_0, \omega_1, \omega, \rho)$.
- $\underline{\mathsf{Encode}(\overline{\omega}, M)}$: Parse $\overline{\omega} := (\omega_0, \omega_1, \omega, \rho)$, sample $(pk, sk) \leftarrow\!\!\$\ \mathsf{KGen}(\rho)$, and $r \leftarrow\!\!\$\ \{0,1\}^*$. Compute $c \leftarrow\!\!\$\ \mathsf{Enc}(pk, M||r)$, $\gamma = \mathsf{Commit}(\omega, M; r)$, and $\pi_0 \leftarrow\!\!\$\ \mathsf{Prove}_0^c(\omega_0, pk, sk)$, and $\pi_1 \leftarrow\!\!\$\ \mathsf{Prove}_1^{sk}(\omega_1, (\omega, \gamma), (M, r))$. Set $C_0 := (pk, c, \pi_0)$ and $C_1 := (sk, \gamma, \pi_1)$, and return $C := (C_0, C_1)$.
- $\underline{\mathsf{Decode}(\overline{\omega}, C)}$: Parse $\overline{\omega} := (\omega_0, \omega_1, \omega, \rho)$ and $C := (C_0, C_1)$, where $C_1 := (sk, \gamma, \pi_1)$ and $C_0 = (pk, c, \pi_0)$. Compute $M||r := \mathsf{Dec}(sk, c)$, and if the following conditions hold return M else return $\perp$:
  - I. Left check: $\mathsf{Ver}_0^c(\omega_0, pk, \pi_0) = 1$.
  - II. Right check: $\mathsf{Ver}_1^{sk}(\omega_1, (\omega, \gamma), \pi_1) = 1$.
  - III. Cross check: $\mathsf{Commit}(\omega, M; r) = \gamma$.
- $\underline{\mathsf{Rfrsh}(\overline{\omega}, (\beta, C_\beta))}$: Parse $\overline{\omega} := (\omega_0, \omega_1, \omega, \rho), C_0 := (pk, c, \pi_0)$, and $C_1 := (sk, \gamma, \pi_1)$. Hence:
  - For $\beta = 0$, pick $r^0_{\mathrm{upd}} \leftarrow\!\!\$\ \{0,1\}^*$, let $c' := \mathsf{UpdateC}(c; r^0_{\mathrm{upd}})$ and $\pi'_0 \leftarrow\!\!\$\ \mathsf{LEval}_0(\omega_0, \mathsf{UpdateC}(\cdot; r^0_{\mathrm{upd}}), (pk, c, \pi_0))$, and return $C'_0 := (pk, c', \pi'_0)$.
  - For $\beta = 1$, pick $r^1_{\mathrm{upd}} \leftarrow\!\!\$\ \{0,1\}^*$, let $sk' := \mathsf{UpdateS}(sk; r^1_{\mathrm{upd}})$, and $\pi'_1 \leftarrow\!\!\$\ \mathsf{LEval}_1(\omega_1, \mathsf{UpdateS}(\cdot; r^1_{\mathrm{upd}}), ((\gamma, \omega), sk, \pi_1))$, and return $C'_1 := (\gamma, sk', \pi'_1)$.

We show the following result. In the full version we provide a concrete instantiation of our code, based on fairly standard computational assumptions.

Theorem 1. *Let $\mathcal{PKE}$ be a PKE scheme with message space $\mathcal{M}_{\mathrm{pke}}$ and public-key space $\mathcal{PK}$, let $\mathcal{COM}$ be a commitment scheme with message space $\mathcal{M}$, and let $\mathcal{NIA}_0$ (resp. $\mathcal{NIA}_1$) be a NIA w.r.t. the relations $\mathcal{R}_0$ (resp. $\mathcal{R}_1$). Define $\mu(\kappa) := \log|\mathcal{M}|, \mu_{\mathrm{pke}}(\kappa) := \log|\mathcal{M}_{\mathrm{pke}}|$, and $\delta(\kappa) := \log|\mathcal{PK}|$.*

For any $\ell \in \mathbb{N}$, assuming that $\mathcal{PKE}$ is an $(\ell + 3\mu + 2\kappa + \max\{\delta, \mu_{\mathrm{pke}}\})$-noisy CLRS-friendly PKE scheme, that $\mathcal{COM}$ is a non-interactive statistically binding commitment scheme, and that $\mathcal{NIA}_0$ (resp. $\mathcal{NIA}_1$) satisfies adaptive multi-theorem zero-knowledge, Φ_0-malleable (resp. Φ_1-malleable) label simulation

extractability, and label derivation privacy, then the coding scheme $\mathcal{CS}$ described above is an ℓ-leakage-resilient continuously non-malleable code with split-state refresh.

Proof Intuition. The proof of the above theorem is quite involved. We provide some highlights here. We defer the formal proof to the full version of the paper. Consider a simulator $(\mathsf{S}_0, \mathsf{S}_1)$, where S_0 simulates a fake CRS $\overline{\omega} = (\omega_0, \omega_1, \omega, \rho)$ by additionally sampling the corresponding zero-knowledge and extraction trapdoors for the NIAs (which are then passed to S_1). At the core of our simulation strategy are two algorithms T^0 and T^1, whose goal is essentially to emulate the outcome of the real tampering experiment, with the important difference that T^0 is only given the left part of a (simulated) codeword C_0 and the left tampering function f_0, whereas T^1 is given (C_1, f_1).

The simulator S_1 then works as follows. Initially, it samples a fresh encoding (C_0, C_1) of 0^μ. More in details, the fresh encoding comes from the (computationally close) distribution where the proofs π_0 and π_1 are simulated proofs. At the beginning of each round, it runs a simulated refresh procedure in which the ciphertext c is updated via $\mathsf{UpdateC}$ (and the simulated proof π_0 is re-computed using fresh randomness), and similarly the secret key sk is updated via $\mathsf{UpdateS}$ (and the simulated proof π_1 is re-computed using fresh randomness). Hence, for each tampering query (f_0, f_1), the simulator S_1 runs $\widetilde{M}_0 := \mathsf{T}^0(C_0, f_0), \widetilde{M}_1 := \mathsf{T}^1(C_1, f_1)$, and it returns $\widetilde{M}_0$ as long as $\perp \neq \widetilde{M}_0 = \widetilde{M}_1 \neq \perp$ (and $\perp$ otherwise). The extra tampering query (f_0^*, f_1^*) is simulated similarly, based on the outcome of the tampering simulators $(\mathsf{T}^0, \mathsf{T}^1)$. We briefly describe the tampering simulators T^0 and T^1:

- Algorithm T^0 lets $f_0(C_0) := (\widetilde{pk}, \widetilde{c}, \widetilde{\pi}_0)$. If the proof $\widetilde{\pi}_0$ does not verify, it returns $\perp$. Else, if $(\widetilde{pk}, \widetilde{c}, \widetilde{\pi}_0) = (pk, c, \pi_0)$, it returns $\diamond$. Else, it extracts the proof $\widetilde{\pi}_0$, this leads to two possible outcomes[10]:
  (a) The extractor outputs a secret key $\widehat{sk}$ which is used to decrypt $\widetilde{c}$, and the tampering simulator returns the corresponding plaintext $\widetilde{M}$.
  (b) The extractor outputs a transformation ϕ which maps the label of the simulated proof π_0, namely the encryption of 0^μ, to $\widetilde{c}$. In this case the tampering function f_0 has modified the original ciphertext c to the mauled ciphertext $\widetilde{c}$ which is an encryption of the same message, so we can safely output $\diamond$.
- Algorithm T^1 lets $f_1(C_1) := (\widetilde{\gamma}, \widetilde{sk}, \widetilde{\pi}_1)$. If the proof $\widetilde{\pi}_1$ does not verify, it returns $\perp$. Else, if $(\widetilde{\gamma}, \widetilde{sk}, \widetilde{\pi}_1) = (\gamma, sk, \pi_1)$, it returns $\diamond$. Else, it extracts the proof $\widetilde{\pi}_1$, again, this leads to two possible outcomes:
  (a) the extractor outputs the committed message $\widetilde{M}$ (along with the randomness of the commitment), so the tampering simulator can simply return $\widetilde{M}$.

[10] The above description is simplified, in that extraction could potentially fail, however, this happens only with negligible probability when the proof verifies correctly.

(b) The extractor outputs a transformation ϕ which maps the label of the simulated proof π_1, namely the original secret key sk, to the mauled secret key $\widetilde{sk}$. In this case, the mauled proof $\widetilde{\pi^1}$ must be a valid proof which instance is the original commitment, so, again, we can safely output $\diamond$.

To show that the above simulator indeed works, we use a hybrid argument where we incrementally change the distribution of the ideal tampering experiment until we reach the distribution of the real tampering experiment. Each step introduces a negligible error, thanks to the security properties of the underlying building blocks. Perhaps, the most interesting step is the one where we switch the ciphertext c from an encryption of zero to an encryption of the real message (to which we always have to append the randomness of the commitment); in order to show that this change is unnoticeable, we rely on the CLRS storage friendly security of the PKE scheme. In particular, this step of the proof is based on the following observations:

- The reduction can perfectly emulate the distribution of the CRS $\overline{\omega}$, and of all the elements $(pk, \pi_0, \gamma, \pi_1)$, except for (c, sk). However, by outputting $(0^\mu||r, M||r)$ as challenge plaintexts—where $r \in \{0,1\}^*$ is the randomness for the commitment—the reduction can obtain independent leakages from C_0 and C_1 with the right distribution.
- Refresh of codewords can also be emulated by exploiting the fact that the reduction is allowed to update the challenge secret key and ciphertext.
- The reduction can answer tampering queries from the adversary by using T^0 and T^1 as leakage functions. The main obstacle is to ensure that T^0 and T^1 are ℓ-leaky, where $\ell \in \mathbb{N}$ is the leakage bound tolerated by the PKE scheme. Luckily, by using carefully the information-theoretic argument explained in the Introduction, we can show that this is indeed the case, which allows simulation to go through. In particular, between each refresh the reduction needs to interleave the executions of T^0 and T^1 until their outputs diverge. So let q be the number of tampering queries that the simulator performs until triggering a decoding error. The leakage that the reduction needs to perform during this stage (namely, between two consecutive refresh) is $\mathsf{T}^0(C_0, f_0^0), \mathsf{T}^1(C_1, f_1^0), \ldots, \mathsf{T}^0(C_0, f_0^q), \mathsf{T}^1(C_1, f_1^q)$ where $(f_0^0, f_1^0), \ldots, (f_0^q, f_1^q)$ is the list of tampering functions applied. By the information-theoretic argument:

$$\begin{aligned}&\widetilde{\mathbb{H}}_\infty(C_0 \mid \mathsf{T}^0(C_0, f_0^0), \ldots, \mathsf{T}^0(C_0, f_0^q))\\ &= \widetilde{\mathbb{H}}_\infty(C_0 \mid \mathsf{T}^1(C_1, f_1^0), \ldots, \mathsf{T}^0(C_1, f_1^{q-1}), \mathsf{T}^0(C_0, f_0^q)).\end{aligned}$$

In fact, the outputs of the $\mathsf{T}^0(C_0, f_0^i)$ and $\mathsf{T}^0(C_0, f_0^i)$ is exactly the same when $i < q$. Moreover:

$$\begin{aligned}&\widetilde{\mathbb{H}}_\infty(C_0 \mid \mathsf{T}^1(C_1, f_1^0), \ldots, \mathsf{T}^0(C_1, f_1^{q-1}), \mathsf{T}^0(C_0, f_0^q))\\ &\geq \widetilde{\mathbb{H}}_\infty(C_0 \mid C_1, q, \mathsf{T}^0(C_0, f_0^q)).\end{aligned}$$

Because the output of a function cannot be more informative than the inputs of the function itself. Lastly, we can notice that C_1 gives little information about C_0 and that q and $\mathsf{T}^0(C_0, f_0^q)$ can decrease the min-entropy of C_0 of at most their size which is $O(\kappa)$. The reduction, therefore, is a valid adversary against for the CLRS storage-friendly security experiment of the PKE.

Remark 1 (On the refresh procedures). The notion of split-state refresh does not imply that a refreshed codeword is indistinguishable from a freshly sampled one. And indeed the codeword of our CNMC-R is not, as the public key pk (resp. the commitment γ) do not change after the refresh algorithms are executed. However, the latter is not required for our proof, as the only thing that matters is that the information about the target codeword that the adversary gathers before a refresh takes place will not be useful after the refresh. Put differently, the adversary could potentially leak the entire values pk and γ, but this information would not be useful for breaking the security of the scheme.

5 Applications

Tamper-Resilient Signatures Without Self-destruct. Consider a signature scheme $\mathcal{SS}$. We would like to protect $\mathcal{SS}$ against tampering attacks with the memory, storing the signing key sk. As observed originally by Gennaro *et al.* [14], however, without further assumptions, this goal is too ambitious. Their attack can be circumvented by either assuming the self-destruct capability, or a key-update mechanism.

Interestingly, Fujisaki and Xagawa [13] observed that, whenever the key-update mechanism is invoked only after an invalid output is generated, the goal of constructing tamper-resilient signature is impossible, even assuming the self-destruct capability. The idea behind the attack is to generate two valid pairs of independent signing/verification keys, and thus to overwrite the original secret key with either of the two sampled signing keys in order to signal one bit of the original key. Note that such an attack never generates invalid signatures, thus rendering both the self-destruct capability and a key-update mechanism useless.

In the full version of the paper we show that it is possible to avoid self-destruct and obtain tamper-resilient signatures against arbitrary attacks in the split-state model.

RAM Compilers. Consider a RAM machine, where both the data and the program to be executed are stored in the random access memory. Such a RAM program is modeled as a tuple consisting of a CPU and its memory. At each clock cycle the CPU fetches a memory location and performs some computation. We focus on read-only RAM programs that do not change the content of the memory after the computation is performed. More in details, a read-only RAM program $\Lambda = (\Pi, \mathcal{D})$ consists of a next instruction function Π, a state state stored in a non-tamperable but non-persistent register, and some database $\mathcal{D}$. The next

instruction function Π takes as input the current state state and input inp, and outputs an instruction I and a new state state'. The initial state is set to $(\mathsf{start}, \star)$.

A RAM compiler is a tuple of algorithms $\Sigma = (\mathsf{Setup}, \mathsf{CompMem}, \mathsf{CompNext})$. Algorithm Setup takes as input the security parameter 1^κ, and outputs an untamperable CRS ω. The memory compiler $\mathsf{CompMem}$ takes as input the CRS ω, and a database $\mathcal{D}$, and outputs a database $\widehat{\mathcal{D}}$ along with an initial internal state state. The next instruction function Π is compiled to $\widehat{\Pi}$ using $\mathsf{CompNext}$ and the CRS. To define security, we compare two experiments (cf. Fig. 2). The real experiment features an adversary A that is allowed, via the interface doNext, to execute RAM programs on chosen inputs *step-by-step*; upon input x, oracle $\mathsf{doNext}(x)$ outputs the result of a single step of the computation, as well as the memory location that is accessed during that step. Additionally, adversary A can also apply tampering attacks that are parametrized by two families of functions $\mathcal{F}_{\mathsf{mem}}$ and $\mathcal{F}_{\mathsf{bus}}$, where: (1) Each function $f \in \mathcal{F}_{\mathsf{mem}}$ is applied to the compiled memory. (2) Each function $f \in \mathcal{F}_{\mathsf{bus}}$ is applied to the data in transit on the bus.

The ideal experiment features a simulator S that is allowed, via the interface $\mathsf{Execute}$, to execute RAM programs on chosen inputs in one g;. Upon input x, oracle $\mathsf{Execute}(x)$ outputs the result of the entire computation and the list of all the memory locations that were accessed during that computation. Briefly, a RAM compiler is tamper-resilient if for all possible logics Π, and all efficient adversaries A, there exists a simulator S such that the real and ideal experiment are computationally indistinguishable. A formal definition follows.

Definition 2 (Tamper simulatability). *A compiler* $\Sigma = (\mathsf{Setup}, \mathsf{CompMem}, \mathsf{CompNext})$ *is tamper simulatable w.r.t.* $(\mathcal{F}_{\mathsf{bus}}, \mathcal{F}_{\mathsf{mem}})$ *if for every next instruction function* Π*, and for every PPT adversary* A*, there exists a PPT simulator* S *and a negligible function* $\nu : \mathbb{N} \to [0,1]$ *such that, for all PPT distinguishers* D *and any database* $\mathcal{D}$*, we have that:*

$$\left|\mathbb{P}\left[\mathsf{D}(\mathbf{TamperExec}^{\mathcal{F}_{\mathsf{bus}},\mathcal{F}_{\mathsf{mem}}}_{\mathsf{A},\Sigma,\Lambda}(\kappa)) = 1\right] - \mathbb{P}\left[\mathsf{D}(\mathbf{IdealExec}_{\mathsf{S},\Lambda}(\kappa)) = 1\right]\right| \leq \mathrm{negl}(\kappa)$$

with $\Lambda := (\Pi, \mathcal{D})$*, and where the experiments* $\mathbf{TamperExec}^{\mathcal{F}_{\mathsf{bus}},\mathcal{F}_{\mathsf{mem}}}_{\mathsf{A},\Sigma,\Lambda}$ *and* $\mathbf{IdealExec}_{\mathsf{S},\Lambda}(\kappa)$ *are defined in Fig. 2.*

We propose two compilers for protecting arbitrary RAM computations against tampering attacks.

First Compiler. The first compiler achieves security in a model where only non-persistent tampering on the buses is allowed. The compiler encodes a random secret key k for an authenticated encryption scheme using a R-CNMC; let (K_0, K_1) be the corresponding codeword. Then, the compiler encrypts each data block in the original memory $\mathcal{D}$, along with its index, under the key k; let $\mathcal{E}$ be the encrypted memory. The encoded memory is made of two parts $\mathcal{D}_0 := (K_0, \mathcal{E})$ and $\mathcal{D}_1 := (K_1, \mathcal{E})$. When fetching the location j, the compiled RAM program first reads and decodes (K_0, K_1), and stores k in the untamperable register; then, it loads $\mathcal{E}[j]$ from both $\mathcal{D}_0$ and $\mathcal{D}_1$ and checks that, indeed, they are the same

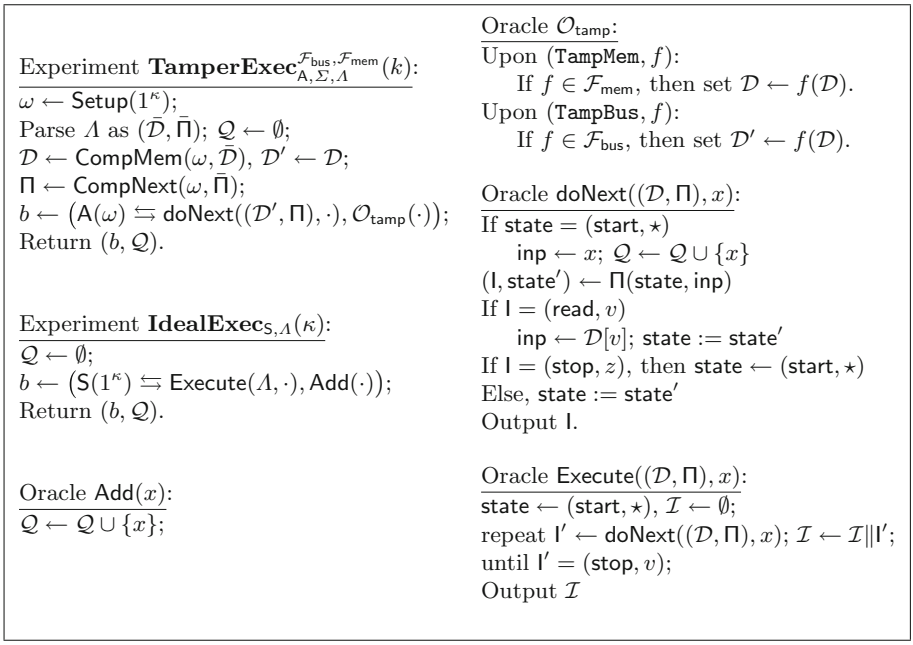

Fig. 2. Experiments defining security of a RAM compiler.

ciphertext, which is then decrypted.[11] If an error happens, the compiled RAM invokes the refresh mechanism.

The reason behind the redundant encoding of $\mathcal{E}$ can be explained using the information-theoretic observation described in the introduction of the paper. In fact, the mauled ciphertexts from $\mathcal{D}_0$ (resp. $\mathcal{D}_1$) can be arbitrary functions of the non-malleable encoding K_0 (resp. K_1). However, as long as the mauled ciphertexts from $\mathcal{D}_0$ are equal to the mauled ciphertexts from $\mathcal{D}_1$, the amount of information they carry about K_0 is bounded by the amount of information that K_1 reveals about K_0. If the two ciphertexts are not equal, some information about K_0 may be leaked, but in this case the codeword is refreshed and the leaked information becomes useless.

In the full version of the paper we prove the following theorem and give the details of the construction.

Theorem 2 (Informal). *Let $n, \kappa \in \mathbb{N}$ be parameters. Assume there exists a coding scheme that is $\mathrm{poly}(\kappa, \log n)$-leakage-resilient R-CNMC and assume there exists an authenticated encryption scheme with ciphertext space of length at least $\mathrm{poly}(\log n)$. Then there exists a tamper-resilient RAM compiler w.r.t. $(\mathcal{F}_{\mathsf{bus}}, \emptyset)$ for RAM programs with database of length n, where $\mathcal{F}_{\mathsf{bus}}$ is the family of split-state tampering functions.*

[11] The compiled RAM program additionally needs to check that the encrypted index is equal to j, in order to avoid shuffling attacks.

Tamper-Resilient for Persistent Tampering. The above compiler is not secure against adversaries that can tamper persistently with the memory. In fact, such attackers can "copy-paste" the value K_0 (resp. K_1) in a part of the memory $\mathcal{D}_0$ (resp. $\mathcal{D}_1$) that is not refreshed, and restore these values at a later point, bypassing the refreshing procedure.

To partially overcome this problem we assume that, once a decoding error is triggered, the system can switch in a *safe mode* where the communication between CPU and memory is tamper free. While in safe mode, the system will perform a consistency check. To minimize the dependency on the assumption we constraint the consistency check to be succinct, meaning that its complexity depends only on the security parameter and not on the size of the RAM program. Finally, if the consistency check passes, the refresh procedure will be executed otherwise the self-destruct is triggered. In the full version of the paper we prove the following theorem and give the details of the construction.

Theorem 3 (Informal). *Let $n, \kappa \in \mathbb{N}$ be parameters. Assume there exists a coding scheme that is $\mathrm{poly}(\kappa, \log n)$-leakage-resilient R-CNMC and assume there exists an authenticated encryption scheme with ciphertext space of length at least $\mathrm{poly}(\log n)$. Moreover, assume the system can switch in safe mode for $\mathrm{poly}(\kappa)$ number of operations and self destruct, then there exists a tamper-resilient RAM compiler w.r.t. $(\mathcal{F}_{\mathsf{bus}}, \mathcal{F}_{\mathsf{mem}})$ for RAM programs with database of length n, where both $\mathcal{F}_{\mathsf{bus}}$ and $\mathcal{F}_{\mathsf{mem}}$ are the family of split-state tampering functions.*

The Compiler of [7]. In order to better compare our RAM compilers with previous work, we first describe the compiler of Dachman-Soled *et al.* [7] in some details. The starting point is a RAM program $\Lambda = (\Pi, \mathcal{D})$ that is previously compiled using an Oblivious RAM [15], and later encoded using a (split-state) locally-updatable and locally-decodable non-malleable code (LULD-NMC)[12]. In particular, one first samples a random key k for an authenticated encryption scheme, encrypts all the locations $\mathcal{D}[i]$ block by block, and finally computes a Merkle tree of the encrypted blocks. A non-malleable encoding (K_0, K_1) of k together with the root of the Merkle tree is computed and the resulting codeword is composed of (K_0, K_1), the encrypted memory $\mathcal{D}'$, and the merkle tree T. Since the encoded memory $\mathcal{D}'$ is encrypted block-by-block, it is possible to locally decode it and update it using $\Omega(\log n)$ operations,[13] where n is the number of blocks in $\mathcal{D}$.

The security model in [7] is a flavour of the standard 2-split-state model tampering model, where the adversary can choose tampering functions $f = (f_1, f_2)$. Tampering function f_1 is any tampering function supported by the underlying 2-split-state NMC that was used to compute (K_0, K_1) and the function can depend on the encrypted memory blocks $\mathcal{D}'$, and the merkle tree T. Tampering function f_2 enables the adversary to tamper with the memory and the merkle tree, but the function *does not* depend on the codeword (K_0, K_1).

[12] The compiler, more generally, can be instantiated with any kind of (standard) NMC, for concreteness we consider only the instantiation based on split-state NMC.

[13] In a subsequent work, Dachman-Soled et al. [6] showed that, in order to have security against "reset attacks", the overhead of $\Omega(\log n)$ is necessary.

Comparison. Finally, let us review the main differences between our RAM compilers and the one by Dachman-Soled *et al.* [7]. First, the compiler of [7] can handle very general RAM programs that can also write on the memory. Our compilers, instead, are specifically tuned for RAMs that make only read operations (recall that we want to avoid write-back operations); this feature allows us to exploit the non-interactive refresh procedure of the underlying R-CNMC. The read-only model is strictly weaker than the model that is considered in [7] and reset attacks cannot exist in our model. This enables us to avoid the use of a Merkle tree and obtain a construction similar to the one given in [7], thus reducing the overhead from $\Omega(\log n)$ to $O(1)$.

Second, the compiler of [7] only achieves security in a variant of the regular split-state model (as described above), whereas both our compilers are secure in the standard split-state model. On the downside, we require an untamperable CRS, which is not needed in [7].

Third, we do not aim to hide the access pattern of the RAM machine. Notice that the latter can be avoided using ORAMs (as done in [7]). However, we think of this as an orthogonal problem. In fact, in some cases, ORAMs could be, more efficiently, replaced by constant-time implementations, or by fixed-pattern ones (for example when hardening cryptographic primitives).

Lastly, our first compiler is the first RAM compiler that achieves security against continuous attacks without relying on the self-destruct capability. This feature allows us also to tolerate non-malicious hardware faults that may affect the data of the bus accidentally, while at the same time maintaining security against malicious tampering attacks. We notice that a similar property could be achieved in the scheme of [7] by applying a layer of error-correcting code over the non-malleable encoding. This allows to transparently correct the hardware faults as long as these faults are supported by the capability of the error correcting code and otherwise self destruct. On the other hand, our compiler cannot correct such hardware faults, but it can detect them (without any bound on their nature) and trigger a refresh before safely continuing the computations.

References

1. Aggarwal, D., Dodis, Y., Kazana, T., Obremski, M.: Non-malleable reductions and applications. In: STOC, pp. 459–468 (2015)
2. Aggarwal, D., Dodis, Y., Lovett, S.: Non-malleable codes from additive combinatorics. In: STOC, pp. 774–783 (2014)
3. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Kaliski, B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 513–525. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0052259
4. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-69053-0_4

5. Chase, M., Kohlweiss, M., Lysyanskaya, A., Meiklejohn, S.: Malleable proof systems and applications. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 281–300. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29011-4_18
6. Dachman-Soled, D., Kulkarni, M., Shahverdi, A.: Tight upper and lower bounds for leakage-resilient, locally decodable and updatable non-malleable codes. In: Fehr, S. (ed.) PKC 2017. LNCS, vol. 10174, pp. 310–332. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54365-8_13
7. Dachman-Soled, D., Liu, F.-H., Shi, E., Zhou, H.-S.: Locally decodable and updatable non-malleable codes and their applications. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015. LNCS, vol. 9014, pp. 427–450. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46494-6_18
8. Davì, F., Dziembowski, S., Venturi, D.: Leakage-resilient storage. In: Garay, J.A., De Prisco, R. (eds.) SCN 2010. LNCS, vol. 6280, pp. 121–137. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15317-4_9
9. Dodis, Y., Lewko, A.B., Waters, B., Wichs, D.: Storing secrets on continually leaky devices. In: FOCS, pp. 688–697 (2011)
10. Dziembowski, S., Pietrzak, K., Wichs, D.: Non-malleable codes. In: Innovations in Computer Science, pp. 434–452 (2010)
11. Faonio, A., Nielsen, J.B.: Non-malleable codes with split-state refresh. In: Fehr, S. (ed.) PKC 2017. LNCS, vol. 10174, pp. 279–309. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54365-8_12
12. Faust, S., Mukherjee, P., Nielsen, J.B., Venturi, D.: Continuous non-malleable codes. In: Lindell, Y. (ed.) TCC 2014. LNCS, vol. 8349, pp. 465–488. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54242-8_20
13. Fujisaki, E., Xagawa, K.: Public-key cryptosystems resilient to continuous tampering and leakage of arbitrary functions. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10031, pp. 908–938. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53887-6_33
14. Gennaro, R., Lysyanskaya, A., Malkin, T., Micali, S., Rabin, T.: Algorithmic tamper-proof (ATP) security: theoretical foundations for security against hardware tampering. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 258–277. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24638-1_15
15. Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: STOC (1987)
16. Govindavajhala, S., Appel, A.W.: Using memory errors to attack a virtual machine. In: IEEE Symposium on Security and Privacy, pp. 154–165 (2003)
17. Liu, F.-H., Lysyanskaya, A.: Tamper and leakage resilience in the split-state model. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 517–532. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_30
18. Otto, M.: Fault attacks and countermeasures. Ph.D. thesis, University of Paderborn, Germany (2006)